Rapport de projet - Fourmis V2

1. Modification apporté (ou pas)

1.0 Loi de demeter

Nous avons essayé de donner à nos classes le moins d'information possible, ils n'ont que les informations nécessaires. Cela nous a conduits à faire des choix, comme par exemple celui de privé au nœud de savoir les fourmis présente sur elle ou de privé le nœud de savoir dans quel graph il est.

1.1 Reine et colonie

Nous sommes partie du fait qu'une reine représentait une colonie entière. La reine a été renommée "*Anthill*".

Chaque fourmi provient d'une reine. Les fourmis reconnaissent donc les fourmis d'autres colonies à l'aide de cet attribut *AntHill* "colony" qui ne sera pas la même. Ce qui nous permet par la suite de pouvoir différencier les phéromones en fonction de leur provenance (de leur colonie).

1.2 Inversion de dépendance

Dans le reçu du premier rapport, l'idée d'avoir un classe Nœud abstraite qui gère le graphe et qui pourrait être sous-type a été proposée. Alors nous avons donc décidé de créer une classe Abstraite Node et une classe Node2D, ce qui nous permet de créer spécialement des méthodes pour la 2D (ou tout autre dimension). La création d'une classe abstraite Node va nous permettre par la suite (s'il le faut) de pouvoir créer des nœuds en n dimension (ou n est un entier supérieur à 0).

1.3 Forte dépendance

Le rapport énonce le faite que dans la v1 l'enum STATE à une forte dépendance avec la classe Node, mais cela à était modifier et nous avons donc créé un enum State global pour ne plus avoir cette forte dépendance.

1.4 <<Abstract>>

Nous avons eu une correction sur le fait que l'indication <<Abstract>> n'était pas nécessaire. Nous trouvons que cela aide beaucoup pour la lecture des diagrammes, nous l'avons donc laissé.

2. Précisions

2.1 Patron créateur (GRASP)

ControlAnt à la responsabilité de créateur de la majorité des classes : Graph, AntHill, *SaveIteration*.

AntHill (reine/colony) crée les soldats et les ouvrières.

Le Graphe crée les Noeuds.

Les fourmis ouvrières créent de la phéromone sur son chemin retour après avoir trouvé de la nourriture.

2.2 Patron contrôleur

ControlAnt est le contrôleur de notre application, il permet de gérer les différentes itérations de l'application et il contrôle toutes les instances de celle-ci. Il peut donc contrôler l'entièreté de l'application et déléguer le travail.

2.3 Liskov Substitution Principle

Les méthodes de fourmis sont toutes génériques excepté de move() qui varie selon le sous-type, c'est pour cela qu'il sera en abstract. Cependant toutes les méthodes de la classe abstraite Noeud sont générique car nous avons pas eu la nécessité d'en créer de spécial pour sa sous-classe Noeud2D, nous aurions pu créer (getX et getY mais pas extrêmement nécessaire).

2.4 Patron Expert en information

La classe *Graphe* a l'information sur l'emplacement des noeuds,

Les *Noeuds* ont l'information : s'il possède un obstacle ou non, leurs nœuds voisins.

Les Fourmis ont l'information sur quel nœud ils sont.

3. Choix de conception

3.1. Phéromones et capacité de collecte

Les ouvrières ont différentes capacités de collecte selon la colonie d'où ils proviennent.

Ils relâchent aussi des phéromones différents selon leur colonie. C'est pourquoi nous avons fait un objet "Pheromone", qui sera différencié à l'aide de l'attribut colonie. Un nœud possède une liste de phéromones et provient d'une colonie.

3.2. Nourriture

Nous avons décidé de simplement représenter la nourriture sous forme de variable présente dans un nœud. Mais si le cahier des charges demandait de faire différent type de nourriture, nous nous serions pencher plus sur une création d'une classe "Nourriture".

3.3 Ouvrière

Le cahier des charges nous demande d'ajouter une fourmis ouvrière, la conception actuel avec la classe abstraite fourmis nous facilite grandement la tâche à l'implémentation de cet catégorie ouvrière (Polymorphisme énoncé dans le premier rapport) L'ouvrière est nécessaire afin de créer et de déposer les phéromones sur les noeuds. **(Patron Créateur)**

3.4. Comportements des ouvrières

Quand une ouvrière a de la nourriture, l'ouvrière repasse par les nœuds de l'historique qu'elle a précédemment remplis avant de trouver de la nourriture (elle supprime 1 à 1 ceux retourner) , alors que si elle en a pas, elle suit les nœuds avec le plus de phéromones (avec un algorithme donné), cependant si le noeud adjacent a de la nourriture, il sera alors privilégié. (Si plusieurs noeud voisins avec nourriture, prend celui qui en a le plus)

Son état passe de l'un à l'autre en regardant si la variable "foodCollected" est différente de 0 ou non.

Si il y a un obstacle pendant le retour à la fourmilière, l'ouvrière reprendra un mouvement aléatoire afin de retrouver sa fourmilière.

3.5. Réécriture du compareTo (exclusif à Java)

Dans une partie de l'algorithme de move() de la fourmis ouvrière, l'algorithme doit effectuer un trie croissant des noeuds en fonction de leur quantité de phéromone, et pour cela nous avons utiliser l'import Collections que java possède, et donc nous avons dû modifier la fonction compareTo pour avoir un trie croissant voulu (la fonction compareTo est appelé dans Collection).

3.6. Création du .csv

Pour respecter le **Patron invention pure** (GRASP) cité dans le rapport de la v1, nous avons décidé de créer un classe spécialement pour faire la partie enregistrement des itérations dans un fichier .csv sous le nom de *SaveIteration*.

Rapport de projet - Fourmis V1

Explication du diagram de conception

classe **ControlFourmis :**

- Permet de faire le lien entre l'interface "AntFacadeController" et les autres classes.
- ControlFourmis est l'**expert en information** *(GRASP)*
- *ControlFourmis est le* ***créateur*** *(GRASP),* car il vas instancié : la reine, les soldats, le graphe, la nourriture

classe **Fourmis :**

- Une fourmi peut être une reine ou un soldat (pour l'instant)

- Une fourmi est sur un noeud et ne peut se déplacer que sur les voisins de ce noeud là (noeud.getFreeVoisins())

- On peut obtenir la positions d'une fourmi et établir sa positions (getPositions et setPositions)

- La reine ne peut se déplacer (Override de la méthode move(), elle peut néanmoins créé des soldats (createSoldiers)

classe **Noeud & Graphe :**

- La position des obstacles passe par la classe Graphe.

- Un nœud possède une liste de ses voisins. (getVoisins()), mais aussi une liste des ses voisins accessible, c'est à dire où getNoeudState() renvoie STATE.FREE.

- Un graphe possède une liste de noeuds

- Un nœud est occupé lorsque une reine (STATE.ANTHILL) ou un obstacle (STATE.OBSTACLE) est sur celle-ci.

- Un graphe peut créer une colonie (donc place une Reine sur une case)

- Si il n'y a plus de graphe , il n'y a plus de nœuds. Donc la relation Graphe vers Noeud est "complète", c'est donc une relation d'agrégation.

Principe GRASP & SOLID

5.a PRINCIPE GRASP

Patron Créateur & **Contrôleur** : La classe *ControlFourmis* est nécessaire afin de pouvoir créer les instances de classe à l'aide de l'interface *AntFacadeController*, c'est-à-dire que *ControlFourmis* est le créateur.

Patron Expert en information : La classe *Graphe* a l'information sur l'emplacement des noeuds,

Les *Noeuds* ont l'information : s'il possède un obstacle ou non, leurs nœuds voisins.

Les fourmis ont l'information sur quel nœud ils sont.

Patron Polymorphisme : la classe Fourmis contiendra la méthode move() qui permettra de déplacer les fourmis, mais elle sera définie dans les sous classes de Fourmis.(Par ex : le déplacement de la reine sera mis à null car elle ne peut pas se déplacer). Cela nous permet d'anticiper l'ajout de nouveaux types de fourmis sans pour autant ajouter un switch dans le code qui deviendra conséquent, ce qui implique alors un **faible couplage**.

Patron invention pure : utilisation future pour la V2 (création du fichier des itérations des fourmis).

5.b PRINCIPE SOLID

Single Responsibility

- Le graphe a pour responsabilité de s'occuper des nœuds
  .
- ControlFourmi ne répond pas à ce principe, mais cela est dû au fait que cette classe est la classe créatrice (Principe **GRASP**).

- La classe nœud a pour responsabilité de permettre l'emplacement d'un obstacle ou d'une fourmilière à l'aide de l'énumération STATE.

- Les fourmis ont seulement l'information de leur noeud (position), leur statut (nourriture ou non) et doivent "naviguer".

Open-Closed

- Ouvert à l'extension : On peut ajouter de nouvelle type de fourmis (le type ouvrière sera ajouté par la suite grâce à ce principe), de même pour l'énumération STATE, nous pourrons ajouter un nouveau statut comme par exemple, le fait de savoir si il y a une fourmis ouvrière ou non.

Liskov Substitution Principle

- On est pas impacté par ce principe, donc pas d'utilisation de ce principe

Interface Segregation Principle

- Pas trop utilisé car cela compliquerait pour rien la conception (sur-ingénierie)

Dependency Inversion

- Un graphe est composé de Noeud, peu importe leur composition ou leur nombre. Pas spécialement de tableau 2d (rectangulaire)

- Nous dépendons aussi de l'interface *AntFacadeController* (+Interface Segregation Principle)

Choix de conception

6.0. Convention anglais/français

Nous somme partie de base en français, mais nous avons étais conseiller d'écrire les variables/fonctions en anglais, alors nous avons dû faire du franglais.

6.1. Fourmi sur un nœud...

Nous avons décidé que c'est la fourmi qui connait le nœud où elle se trouve et **non** le nœud qui connaît les fourmis sur elle.
La fourmi a un attribut -position de type Noeud.

6.2. Collection Java et voisins..

Nous avons décidé de partir sur une ArrayList (tableau 1d avec des index et sans limite. Cela nous permet de ne pas se restreindre à un tableau 2d et ainsi avoir une structure qui puissent changer. Ainsi chaque nœud (sommet) aura une liste de voisins. Les listes de voisins seront instanciées dans le constructeur du Graphe, après avoir instancié tous les nœuds dont nous avions besoin.

6.3. Statut des noeuds..

Les noeuds ont 3 statut différents :

- Libre (FREE)

- Obstacle (OBSTACLE)

- Fourmilière (ANTHILL)

Ce qui permet de différencier les nœuds et nous permet de par exemple ne pas permettre au soldat de passer par un nœud avec un obstacle ou de ne pas placer un obstacle sur une fourmilière.