

Module Java Avancé

AntHill



Sommaire

I. Introduction	2
II. Cahier des charges	3
III. Choix des techniques.....	7
IV. L'application.....	8
a) Exemple d'une carte.....	8
V. Analyse	9
a) Diagramme de cas d'utilisations	9
b) Diagrammes de séquences	10
Ajout d'un ordre.....	10
Lancement du simulator.....	11
VI. Conception UML.....	12
a) Les couches	12
b) Couche simulator	13
Couche simulator - Structure de données	13
Couche simulator - Les tâches	15
Couche simulator - Gestion de la fourmilière.....	16
Couche simulator - Factory	16
Couche simulator - Chargement d'une fourmilière	17
Couche simulator - Affichage.....	18
Couche simulator - Relation gotA-simulator.....	19
Couche simulator - Son.....	21
Couche simulator - Exceptions.....	21
Couche simulator - Thread principal	22
Couche simulator - Point d'entrée	22
c) La couche util	23
d) La couche gota	24
VII. Aspects techniques.....	25
VIII. JAVADOC.....	27
IX. Ant	28
X. Problèmes rencontrés	29
XI. Améliorations possibles.....	30
XII. Conclusion.....	31
XIII. Webographie	32

I. Introduction

Dans le cadre du module Java Avancé de M1 Informatique, un projet de simulateur de fourmilière a été donné comme application des cours vus pendant le premier semestre.

Pour réaliser cette application, une bibliothèque graphique a été fournie par le corps enseignant et notre tâche a donc été de créer une application qui simule la vie d'une fourmilière en utilisant la bibliothèque pour l'affichage. L'application est divisée en deux parties, un « dieu » des fourmis et un simulateur qui régit les interactions entre la fourmilière et les fourmis guidées par le dieu.

Ce rapport présente le résultat de notre travail. Il se découpe en plusieurs parties, la première consiste à présenter le cahier des charges, les choix techniques et logiciels que nous avons mis en place. Ensuite, nous présenterons l'architecture choisie. Enfin, nous conclurons sur l'enrichissement personnel que nous a apporté ce projet et les améliorations qui pourraient être effectuées par la suite.

II. Cahier des charges

(Reçu fin octobre 2007)

Le but de ce projet est de réaliser un simulateur de fourmilière. Ce simulateur régit l'évolution d'une fourmilière en faisant respecter des règles du jeu. Outre la description de la fourmilière en elle-même, le simulateur fait intervenir des fourmis ainsi que des insectes nuisibles aux fourmis. Le comportement des fourmis est dicté par un Dieu des fourmis, qui joue avec la fourmilière au fil de la simulation. Après un réglage initial de paramètres, la fourmilière devra être autonome dans son évolution jusqu'à la mort de la reine.

Bibliothèque graphique

Le projet est à réaliser avec la bibliothèque graphique lawrence (disponible à <http://gforgeigm.univ-mlv.fr/projects/lawrence>). Elle doit prendre en charge toute la partie graphique de la manière suivante:

- elle permet l'ouverture d'une fenêtre représentant une grille dont chaque case est remplie par une ou plusieurs icônes (image bitmap ou rendu vectoriel svg) ;*
- chaque icône est représentée par un objet d'un certain type E et l'application modifie l'affichage d'une case en indiquant la liste de ses icônes sous forme de Collection<E>.*

Un exemple de jeu de chasse à la grenouille est disponible sur le site web de lawrence.

Il n'est pas autorisé de gérer l'affichage graphique par un autre moyen, ni de modifier ou recopier le code du packaging, ni de placer ses propres classes dans le packaging fr.umlv.lawrence.

Éléments du jeu

La fourmilière

La fourmilière contient les éléments suivants, répartis dans des cases, c'est-à-dire des zones géographiques repérées par leurs coordonnées :

- une zone de ponte, possiblement constituée de plusieurs cases contiguës, où sera initialement placée la reine de la fourmilière ;*
- des zones prédéfinies où de la nourriture apparaîtra en quantité fixée et se remplissant périodiquement ;*
- des obstacles (e.g. paroi, etc.) ; rien d'autre ne peut se trouver sur une case contenant un obstacle ;*
- des zones où l'on peut passer (e.g. galerie, chemin, etc.) ;*
- des fourmis ;*
- des insectes.*

La fourmilière est paramétrée à sa création par un ensemble de valeurs (emplacements, caractéristiques, périodicité et rendement des ressources, etc.) spécifiées sous la forme d'un fichier de configuration lu par le simulateur.

Les fourmis

Les fourmis ont deux caractéristiques essentielles qui sont leur nombre de points de vie et la capacité de leur jabot social [1] . Les fourmis ont les capacités suivantes :

- *se déplacer [2] ;*
- *ingurgiter une quantité de nourriture à condition que ladite quantité soit présente dans la même case que la fourmi et que son jabot social puisse accueillir cette quantité*
- *régurgiter tout ou partie de la nourriture contenue dans son jabot social.*

La reine des fourmis a la particularité de pouvoir créer des fourmis si elle est située dans sa zone de ponte. Le principe général est qu'une reine peut créer une fourmi à partir du contenu de son jabot social. Les fourmis comme la reine agissent conformément à la volonté de GotA, dieu des fourmis (voir plus loin). Les fourmis comme leur reine perdent des points de vie au contact des insectes, et meurent lorsqu'elles n'ont plus de point de vie.

[1] La trophallaxie est un mode de transfert de nourriture utilisé, entre autres, par les fourmis. Les fourmis possèdent deux estomacs. Lorsque l'une d'elles ingurgite de la nourriture, la majeure partie de celle-ci est stockée dans le second estomac, le jabot social, appelé également estomac social. La trophallaxie consiste en une régurgitation de la nourriture pré-digérée contenue dans ce dernier afin de nourrir d'autres fourmis. On considère dans ce projet que la fourmi n'a pas besoin de s'alimenter pour survivre. Elle ne possède donc qu'un seul estomac -- le jabot social -- qui lui sert à apporter à la reine de la nourriture.

[2] Tout déplacement d'une fourmi ou d'un insecte se fait en demandant à la fourmi ou à l'insecte de se déplacer d'une case vers une des quatre cases immédiatement adjacentes (au dessus, en dessous, à gauche ou à droite -- pas en diagonale). Si un obstacle empêche le déplacement, aucune erreur ne sera signalée mais le déplacement n'aura pas lieu. Dans le cas particulier d'un insecte, le déplacement n'est possible que si la case de départ ne contient pas de fourmi. En la présence d'au moins une fourmi, l'insecte ne peut pas se mouvoir.

Les insectes

Les insectes nuisent à la fourmilière. Les insectes n'ont pas besoin de se nourrir et sont immortels. Un insecte inflige des dégats (en nombre de points de vie) aux fourmis qui se trouvent dans la même case que lui ainsi que dans les huit cases adjacentes. Un insecte peut se déplacer [2] à condition qu'aucune fourmi en vie ne se trouve dans la même case que lui (tant que des fourmis en vie sont dans sa case, il est contraint d'y rester).

C'est le simulateur et non GotA qui dicte leur comportement aux insectes.

Le Dieu des fourmis, GotA

GotA, dieu des fourmis (God of the Ants), est un Dieu omniscient qui connaît la fourmilière, sait où sont les insectes, et commande leurs actions aux fourmis. Il doit néanmoins respecter les règles du jeu imposées par le simulateur.

En pratique, GotA interagit de deux manières avec le reste du jeu. D'une part, il dispose d'un ensemble de méthodes qui sont appelées par le simulateur pour lui faire savoir ce qui se passe dans la fourmilière :

- *GotA est prévenu lorsque de la nourriture apparaît, sans savoir où ni s'il s'agit de nourriture régurgitée ;*
- *GotA est prévenu lorsqu'une fourmi ou la reine est blessée ou morte.*

D'autre part, GotA peut consulter l'état de la fourmilière à chaque instant, et accéder à chacune des fourmis auxquelles il dicte leurs actions. Plus particulièrement :

- *GotA peut obtenir le contenu de toute case de la fourmilière ;*
- *GotA sait localiser chaque fourmi et connaît leurs points de vie ;*
- *GotA peut connaître certaines caractéristiques de la fourmilière comme le nombre d'insectes, le coût en nourriture de la création d'une fourmi par la reine, etc.*

Le simulateur

Le simulateur est le maître du jeu, qui régit les interactions entre une fourmilière, des fourmis guidées par leur dieu GotA, et des insectes. Le principe est que le simulateur "laisse" jouer GotA avec ses fourmis dans la fourmilière, en le tenant au courant de certaines évolutions, tout en vérifiant qu'il ne triche pas. C'est le simulateur qui fait bouger les insectes.

Après avoir chargé une fourmilière, des insectes, des fourmis et le dieu GotA, le simulateur établit les paramètres régissant le fonctionnement de la fourmilière comme le nombre d'insectes, les quantité et périodicité d'apparition de la nourriture, etc. Le simulateur démarre ensuite le GotA par l'appel à sa méthode `play()` avec en argument un objet permettant d'interagir avec la fourmilière. Au fil de la simulation, le simulateur appelle des méthodes de GotA pour lui indiquer les évolutions de la fourmilière.

Chaque ordre donné par GotA à une fourmi prend un certain temps qui est paramétrable par le simulateur via un fichier de configuration : pendant cet intervalle de temps, la fourmi est "indisponible" (occupée à la réalisation de son action) et tout nouvel ordre donné par GotA "bloque" ce dernier jusqu'à la fin de la durée nécessaire à la réalisation de l'ordre précédent. Néanmoins, pendant qu'une fourmi exécute un ordre, GotA peut donner des ordres aux autres fourmis ou traiter les informations transmises par le simulateur. À titre indicatif, voici des exemples de "durée" d'une action :

- *une fourmi prend 100 millisecondes pour se déplacer ;*
- *une fourmi prend 200 millisecondes pour manger et 100 millisecondes pour régurgiter ;*
- *une reine prend 1 seconde pour pondre.*

Principe du jeu et travail demandé

Vous devez principalement réaliser 3 choses pour ce projet :

- *le jeu en lui-même, c'est-à-dire le simulateur gérant l'évolution de la fourmilière et ses acteurs avec son interface graphique utilisant la bibliothèque lawrence ;*
- *un ensemble de classes et d'interfaces visibles et utilisables par GotA pour lui permettre de jouer avec ses fourmis et d'influer sur l'évolution de la fourmilière ;*
- *un GotA particulier qui joue avec votre jeu.*

Rendu

Le projet (simulateur, fourmilière, insectes, fourmis et votre GotA) est à rendre par mail à l'enseignant de cours et aux chargés de TD au plus tard le 7 janvier 2008. Le format de rendu est une archive au format zip (tout tar.gz, rar, 7z et autre ne sera pas ouvert) contenant :

- *un répertoire src contenant les sources du projets et les éventuelles ressources (images, sons, etc.) à recopier à côté des classes ;*
- *un répertoire classes vide, qui contiendra les classes compilées (ne les incluez pas lors de votre livraison)*
- *un répertoire docs contenant un manuel de l'utilisateur (user.pdf) et un manuel qui explique votre architecture (dev.pdf) au format PDF ;*
- *un répertoire lib contenant les éventuelles bibliothèques dont a besoin votre projet pour fonctionner (dont celles de lawrence)*
- *un jar exécutable antHill.jar qui fonctionne avec java -jar antHill.jar et qui possède donc une directive Class-Path correcte dans son manifest ; il est interdit de mettre les classes de lawrence et/ou de batik dans antHill.jar ;*
- *un build.xml qui permet de*
 - *compiler les sources (target compile)*
 - *créer le jar exécutable (target jar)*
 - *générer la javadoc dans docs/api (target javadoc)*
 - *nettoyer le projet pour qu'il ne reste plus que les éléments demandés (target clean)*

Polythéisme et soutenance

Un peu avant la soutenance, nous vous confierons le projet d'un autre binôme (tel qu'il aura été rendu avec documentations, code sources, librairies, etc.) et il vous sera demandé de créer un dieu pour leur simulateur. Le but sera alors de créer un dieu qui triche et qui arrive à faire le maximum de choses interdites (planter le programme fait partie des choses interdites). Du point de vue du développeur du simulateur, veillez donc à ce que votre logiciel ne puisse pas planter, ait une architecture de sécurité, etc.

III. Choix des techniques



Ce projet étant réalisé en JAVA, nous avons choisi l'outil de développement NetBeans. Nous nous sommes également servis de son module UML.



Le logiciel nécessite la machine virtuelle java, appelée JVM. Elle doit être installée sur le poste, ce qui permettra d'interpréter le code java pour l'exécuter. La version 1.6 est requise pour l'application.

Tous les logiciels utilisés lors de ce projet sont libres. L'avantage d'utiliser ces logiciels, est bien sûr la gratuité, mais aussi la qualité. En effet, ces logiciels sont très souvent aussi puissants voire meilleurs que leurs équivalents en version propriétaire.

IV. L'application

a) Exemple d'une carte

Ci-dessous, un exemple d'une fourmilière générée par notre application. Cet exemple a pour but de donner une vision globale de l'application et de ses éléments pour une meilleure compréhension de la suite de ce rapport.

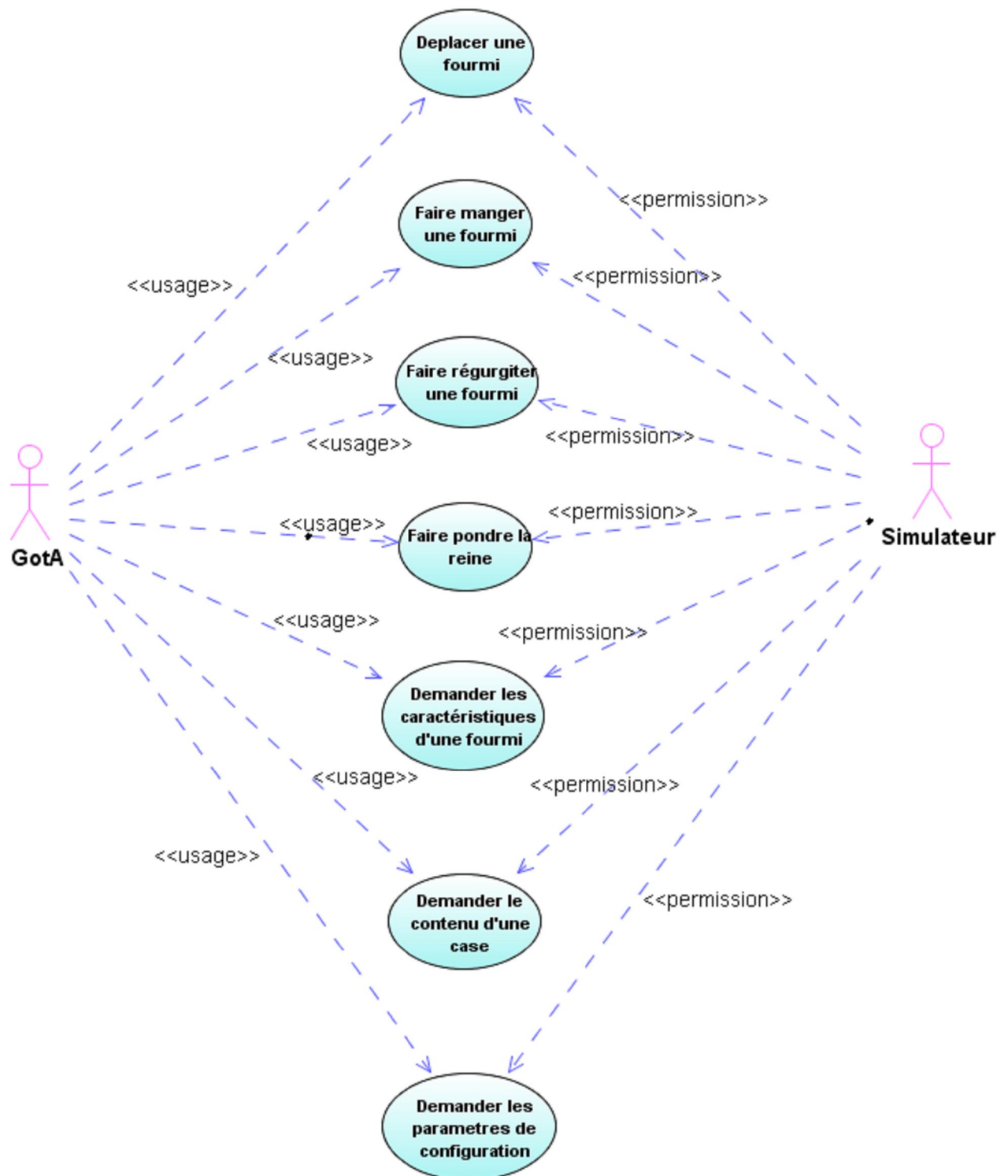


Le détail des éléments graphiques est précisé dans le manuel utilisateur fourni avec ce même rapport. Ce guide explique également la création d'une carte avec les paramètres de son choix et le chargement de celle-ci dans l'application.

V. Analyse

a) Diagramme de cas d'utilisations

Le diagramme suivant découle du cahier des charges. Il permet de mettre en évidence les droits que le dieu des fourmis possède sur le simulateur.

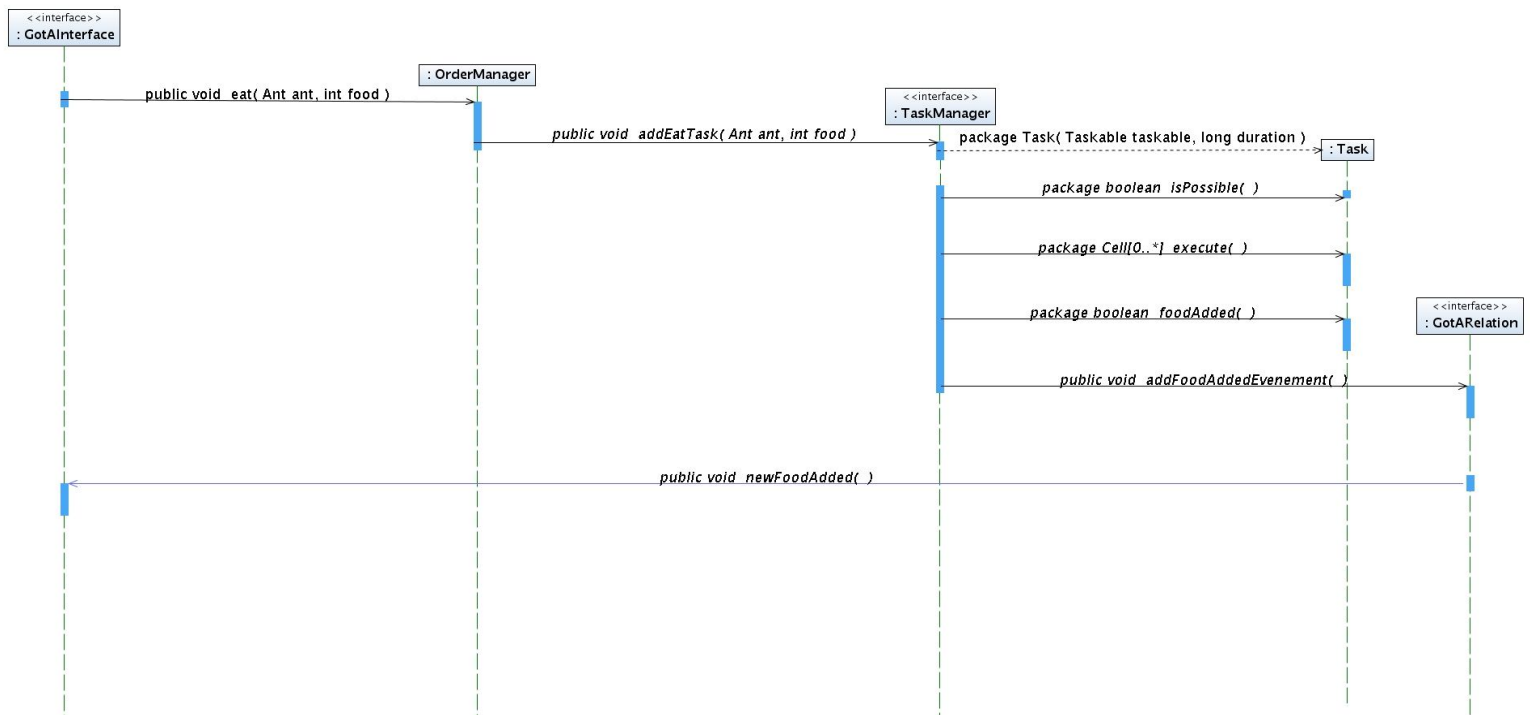


b) Diagrammes de séquences

Les diagrammes suivants ont simplement pour but de présenter une partie de la structure.
Les classes ainsi que les choix utilisés sont expliqués par la suite, dans la partie conception UML.

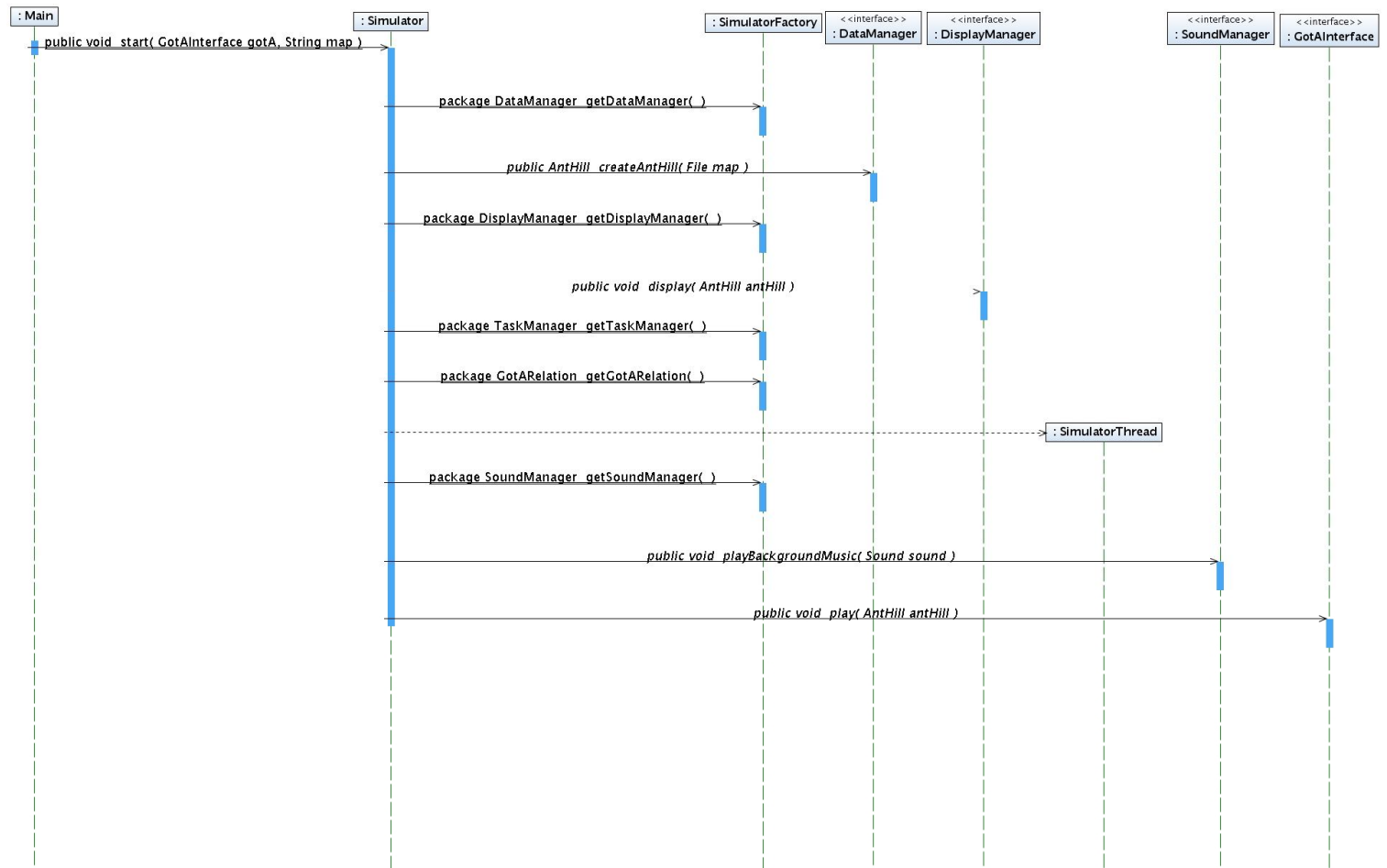
Ajout d'un ordre

Le diagramme de séquence suivant présente l'ajout d'un ordre par le **GotA**. Les ordres correspondent à une partie des cas d'utilisations précédents. Le **GotA** utilise **OrderManager** qui délègue l'action à **TaskManager**. C'est elle qui crée la tâche. Une fois la durée d'attente de la tâche épuisée, celle-ci est testée et exécutée. Si de la nourriture a été ajoutée par la tâche, le **GotA** est prévenu par l'intermédiaire de **GotARelation**. C'est le même principe qui est appliqué pour chacun des ordres disponibles dans **OrderManager**.



Lancement du simulator

Pour lancer le simulator, le **main** prend un **GotA** et une carte en paramètres et les envoie à la classe **Simulator**. C'est cette classe qui s'occupe de demander le chargement de la carte ainsi que de lancer les différents threads nécessaires au fonctionnement du simulateur. Pour finir elle lance le **GotA** en appelant sa méthode **play()**.



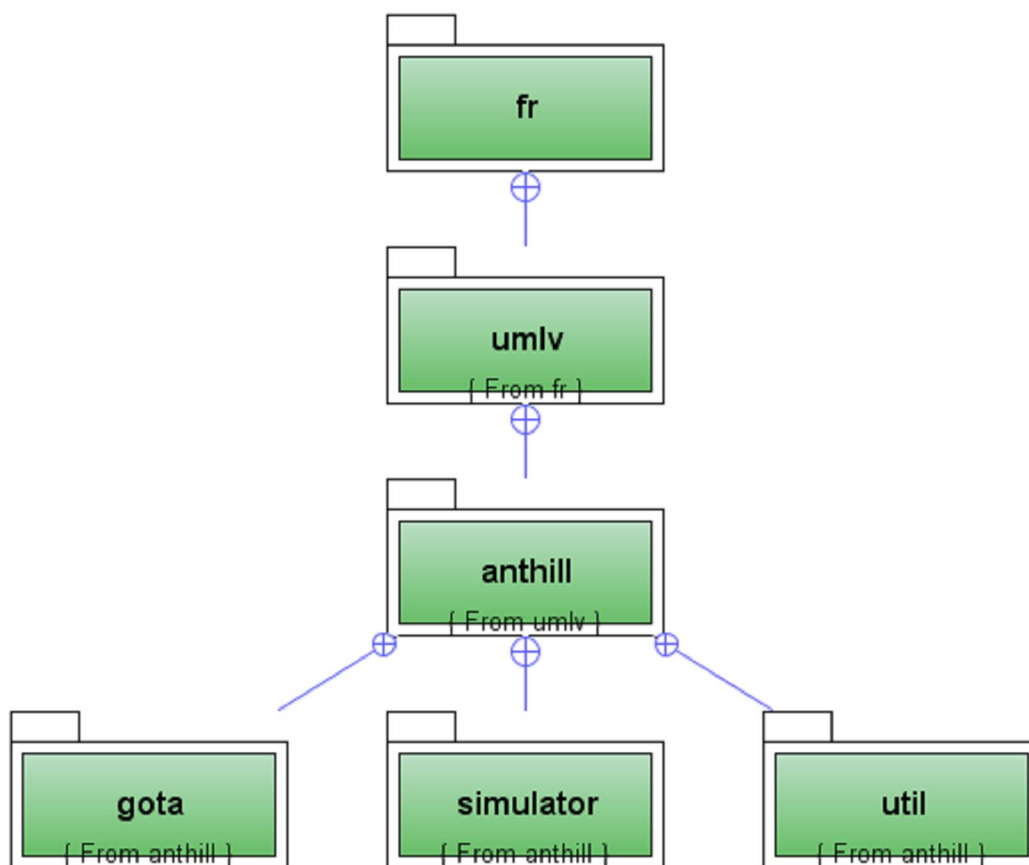
VI. Conception UML

La conception UML est une phase clef du développement d'un logiciel. Cette étape consiste à modéliser le fonctionnement du programme sous forme de schémas. On y définit la structure de données et toutes les méthodes qui seront nécessaires au bon fonctionnement du logiciel. Une fois la conception UML finalisée, on peut alors créer toutes les classes et ainsi commencer la partie codage. La qualité de la conception UML dépend de la qualité et de la précision du cahier des charges.



Les schémas UML suivants ont été réalisés avec l'outil UML de Netbeans.

a) Les couches



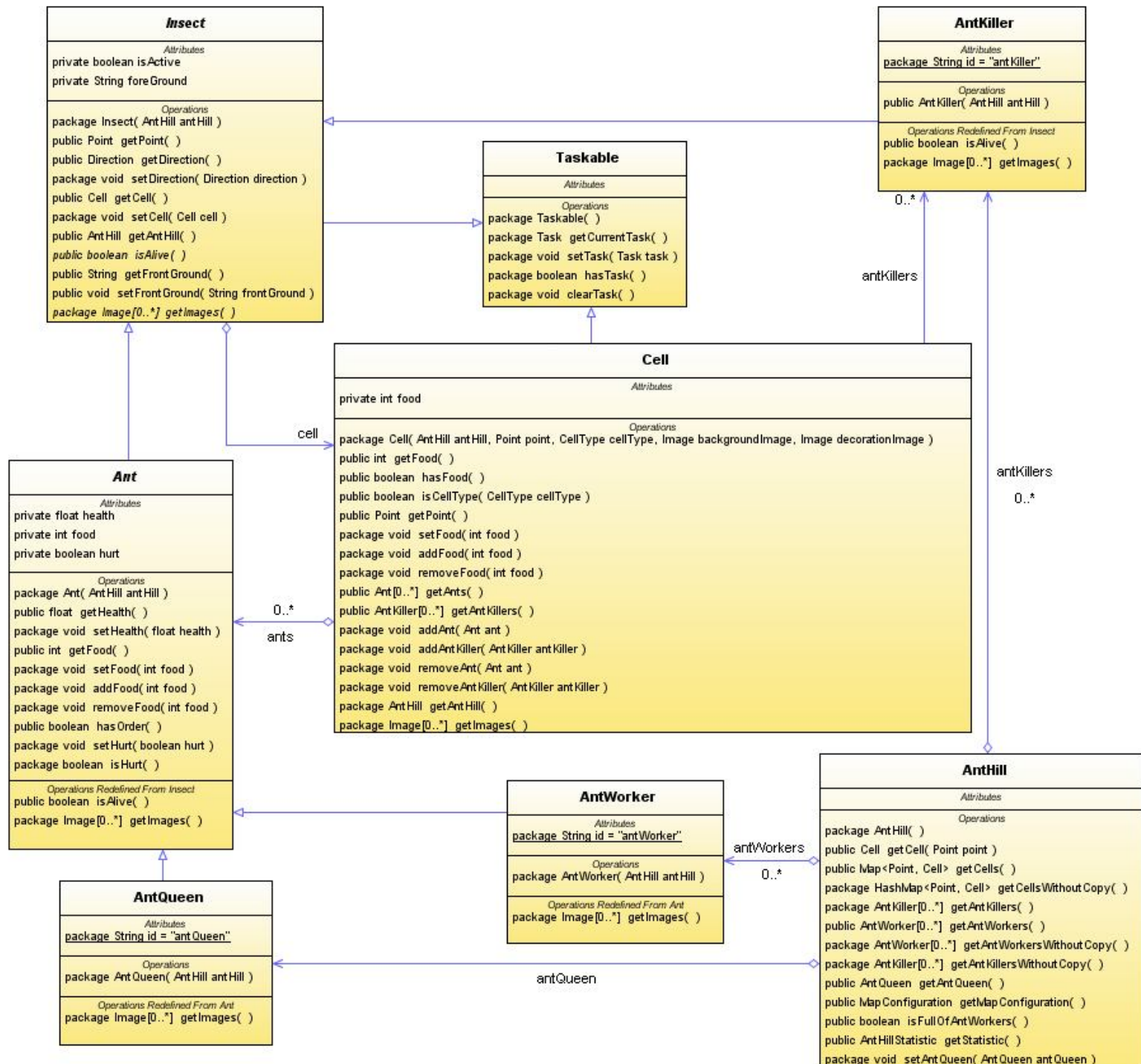
Le contenu des packages **gota**, **simulator** et **util** sont détaillés par la suite.

La couche **simulator** contient la quasi-totalité du projet. Nous avons choisi de centraliser toutes les classes dans ce package dans un souci de sécurité. En effet, une partie du sujet étant de pouvoir développer un dieu des fourmis (par la suite appelé gotA) qui triche avec le simulateur. La séparation des classes en plusieurs couches nous aurait obligé à déclarer ces classes publiques, les rendant accessible au gotA.

Toutes les relations ne sont pas représentées sur les diagrammes suivants, par souci de lisibilité.

b) Couche simulator

Couche simulator - Structure de données



Voici notre structure de données. C'est le centre de notre application. La classe **AntHill** est la classe mère de la structure. Cette classe contient une reine (**AntQueen**), une liste de travailleuses (**AntWorker**) et une liste de tueurs de fourmis (**AntKiller**). **AntHill** contient également une liste de cases (**Cell**) chacune associée à une position (**Util.Point**). **Insect** est la classe de base des êtres pouvant se mouvoir dans la fourmilière. C'est elle qui permet d'associer à un insecte une image, une case ou encore une direction.

Ant est la classe mère de **AntWorker** et **AntQueen**. Cela permet de définir les points de vie et de nourriture d'une fourmi. Ces niveaux ne sont pas définis dans **Insect** car les tueurs (**AntKiller**) sont immortels et n'ont pas besoin de se nourrir. Les cases et les insectes héritent de **Taskable**. Ceci permet de créer des tâches (**Task**) sur ces objets (voir partie suivante).

Nous avons choisi d'utiliser le polymorphisme pour notre structure. Cela permet de définir des méthodes pour le **GotA** comme par exemple **moveAnt(Ant ant)** (voir plus loin pour les méthodes possibles) qui lui permettra de déplacer des **AntWorker** et **AntQueen** du fait de l'héritage. De plus, cela permet, de bloquer la possibilité au **GotA** de déplacer un **AntKiller**, qui lui, n'hérite pas de **Ant**.

Un second avantage d'utiliser le polymorphisme est que le simulateur peut utiliser des méthodes génériques pour tous les insectes, comme un **moveInsect(Insect insect)**.

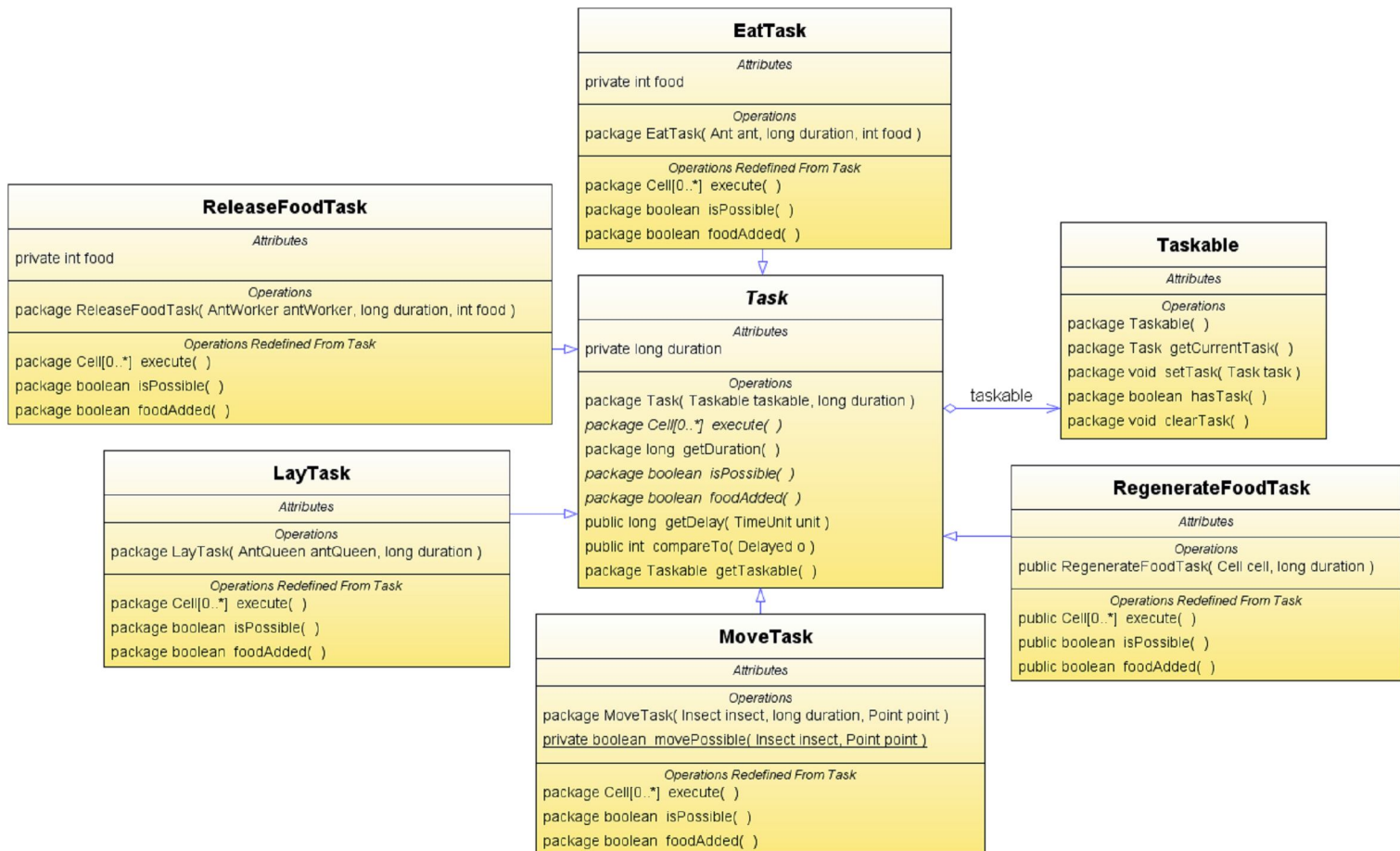
AntHillStatistic	MapConfiguration
<p><i>Attributes</i></p> <pre>private int nbAntWorkersCreate private int nbAntWorkersSimultaneous private int nbAntKillersSimultaneous private int nbAntWorkersCurrently private int nbAntKillersCurrently private long startGameTime private String endOfLine</pre> <p><i>Operations</i></p> <pre>package AntHillStatistic() public int getNbAntWorkersCreate() public void incrementNbAntWorkersCreate() public void decrementCurrentCountOfAntWorkers() public int getNbAntWorkersSimultaneous() public int getNbAntKillersSimultaneous() public void incrementNbAntKillersCreate() public void decrementCurrentCountOfAntKillers() public void startGameTime() public long getDuration() public String getStringStatistics()</pre>	<p><i>Attributes</i></p> <pre>private int antWorkersMaxCount private int movingDuration private int eatingDuration private int layingDuration private int releasingFoodDuration private int antWorkerHealthMax private int antQueenHealthMax private int regeneratingFoodDuration private int regeneratingFoodAmount private int foodNeededToLay private int antWorkerFoodMax private int antQueenFoodMax private int mapWidth private int mapHeight private int cellSize private String antHillName private int countRatioOfAntKillersPerAntWorkers private float healthLostPerMs</pre>

Une **AntHill** est associée à **AntHillStatistic** et **MapConfiguration**.

MapConfiguration contient les paramètres de configuration de la fourmilière. Tous les paramètres sont définis avec une valeur par défaut, mais l'utilisateur peut les personnaliser grâce à un fichier de configuration dont la structure est expliquée par la suite.

AntHillStatistic permet de connaître l'état courant de la fourmilière. Ces statistiques sont mises à jour en temps réel lors de la simulation. Cette classe permet de connaître le temps d'exécution de la simulation, le nombre de fourmis créé, etc. Elle est donc, par exemple, mise à jour à chaque ponte de la reine. Les statistiques sont affichables tout au long de la simulation en appuyant sur la touche « S ».

Couche simulator - Les tâches



Notre architecture principale s'articule autour d'un système de tâches centralisées utilisé par le simulateur ainsi que le **GotA**. Cette structure est nommée par le design pattern Commande. L'utilisation des tâches permet de factoriser le code en définissant à un seul endroit la manière de la réaliser. Ceci simplifie le debug et l'avantage le plus important est bien sur de pouvoir ajouter facilement une nouvelle tâche.

Une tâche possède une durée d'exécution. Il nous a alors fallu trouver un moyen de ne pas bloquer l'application pendant la durée d'attente de la tâche. Une solution est bien sûr de lancer chaque tâche dans un thread, ce qui est évidemment assez lourd. Nous avons alors choisi de créer un seul et unique Thread qui s'occupe de les exécuter (voir page suivante).

Sur le diagramme sont représentées les tâches possibles de l'application. Nous avons choisi de faire des tâches (**Task**) permettant d'effectuer des modifications sur un objet **Taskable** en définissant le temps d'attente avant d'effectuer celles-ci. **Insect** et **Cell** implémentent donc **Taskable**. Par exemple **MoveTask** permet de déplacer un **Insect** et **RegenerateFoodtask** de remettre de la nourriture sur une case. Le constructeur des tâches limite le type de **Taskable** fourni. Par exemple, il n'est pas possible de créer une **EatTask** avec un objet **Cell**.

Couche simulator - Gestion de la fourmilière

<<interface>>	
AntHillManager	
Attributes	
Operations	
public void	createAntKiller(AntHill antHill, Point point)
public void	createAntWorker(AntHill antHill, Point point)
public Cell	createCell(AntHill antHill, Point point, CellType cellType, Image backgroundImage, Image decorationImage)
public void	createQueen(AntHill antHill, Point point)
public void	moveAnt(Ant ant, Point point)
public void	moveAntKiller(AntKiller antKiller, Point point)
public void	removeAntKiller(AntKiller antKiller)
public void	removeAntWorker(AntWorker antWorker)

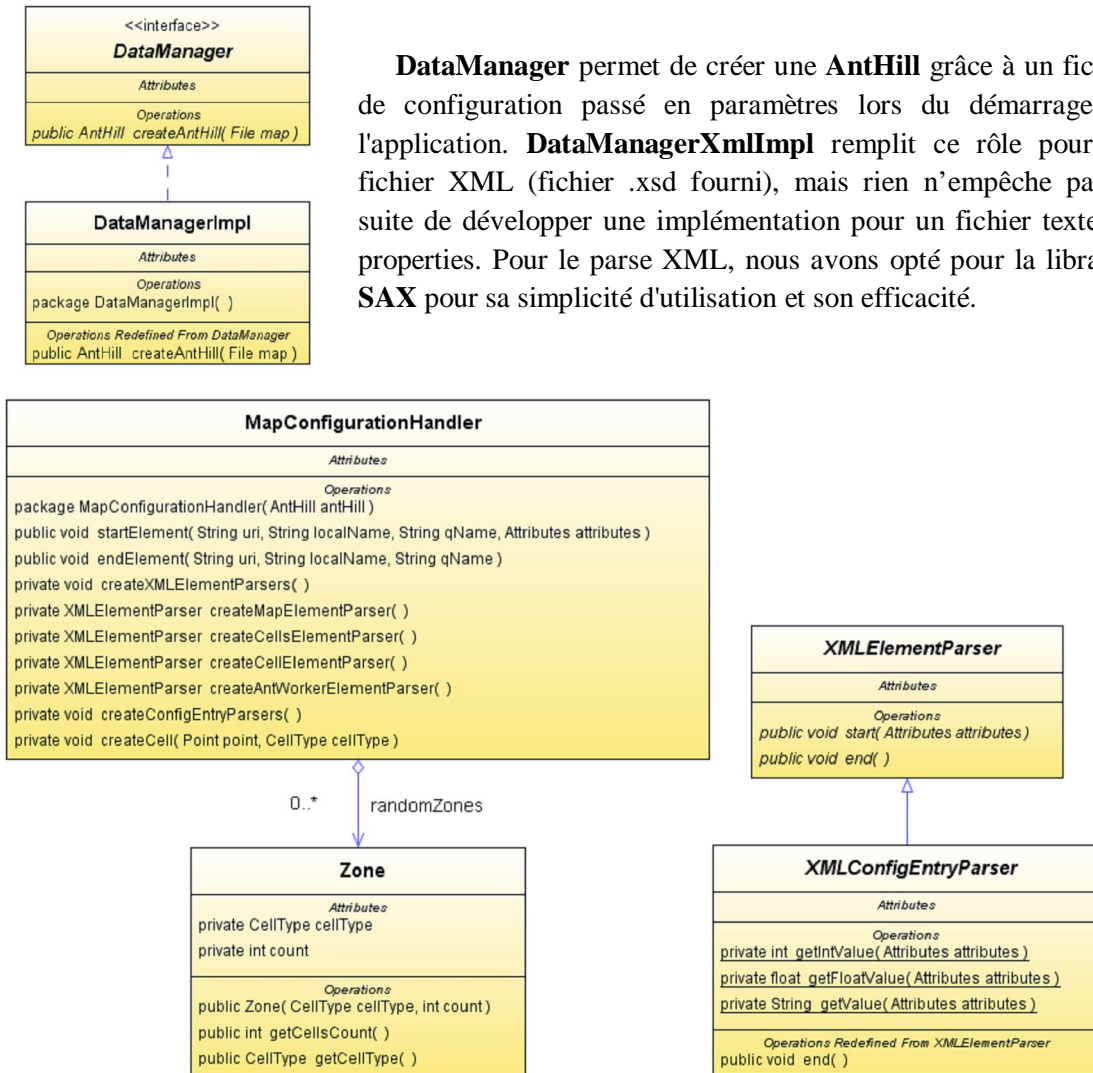
Cette classe est utilisée pour modifier la structure interne de la fourmilière ; par exemple, supprimer des insectes ou les déplacer d'une case à une autre. Elle fait également office de Factory d'**Insect**. C'est aussi dans cette classe que l'on met à jour les statistiques de la fourmilière lorsqu'une des méthodes est appelée. Cette classe est aussi utilisée lors du chargement d'une fourmilière pour initialiser les cases et leurs contenus (voir par la suite). Elle offre donc une couche entre la structure de données et le reste du package simulator, ce qui permet de tout centraliser à cet endroit.

Couche simulator - Factory

SimulatorFactory	
Attributes	
Operations	
private SimulatorFactory()	
public OrderManager	getOrderManager()
package DisplayManager	getDisplayManager()
package DataManager	getDataManager()
package TaskManager	getTaskManager()
package GotARelation	getGotARelation()
package SoundManager	getSoundManager()
package AntHillManager	getAntHillManager()

SimulatorFactory est une classe statique qui permet de récupérer les implémentations de chacune des interfaces du package simulator, notamment les managers qui offrent des services comme la gestion des tâches pour **TaskManager**. Lorsqu'une classe a besoin d'une gestion elle ne l'instancie par directement, mais demande à **SimulatorFactory** de lui retourner l'implémentation courante. Le changement d'une implémentation aura donc peu d'impact sur le code. Cette classe sera la seule impactée par le changement d'une implémentation. Cette classe est utilisée par la couche simulator mais aussi par **GotA**. En effet, c'est à partir de cette classe que le **GotA** récupère la gestion **OrderManager** lui permettant de donner des ordres aux fourmis.

Couche simulator - Chargement d'une fourmilière

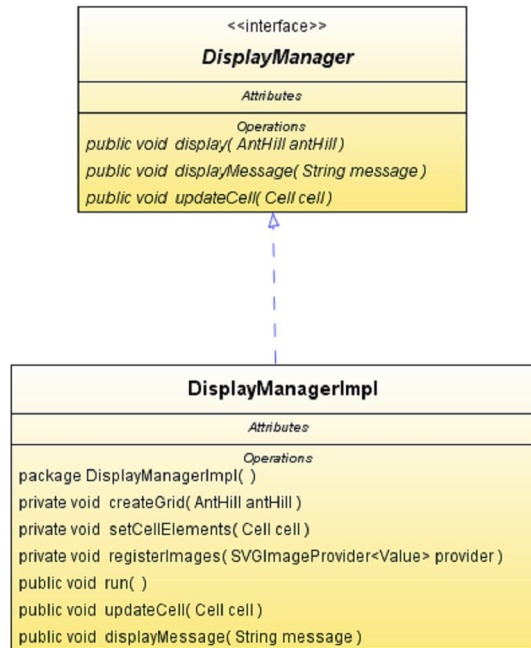


Nous avons conçu notre parse XML autour d'une architecture utilisant une factory statique de parseur d'éléments. Nous utilisons deux types de parseurs, un pour parser les options de configuration (**XMLConfigEntryParser**) et l'autre pour charger les éléments XML (**XMLElementParser**).

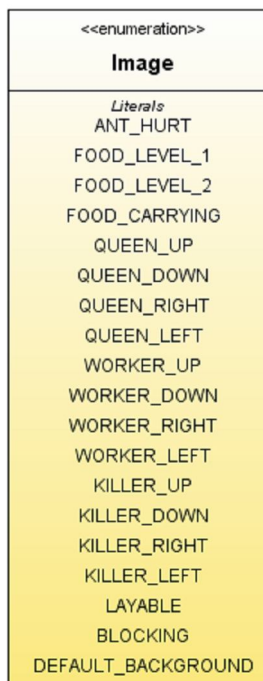
Notre handler SAX (**MapConfigurationHandler**) possède donc deux maps contenant un ensemble de parseurs ajouté lors de la construction du handler. Ces deux collections vont permettre au handler de déléguer dynamiquement le parse de chaque élément au parseur adéquat rencontré lors de l'analyse du fichier XML.

Lorsqu'un élément XML est rencontré, son parseur est appelé à partir de la map d'**XMLElementParser**. Si l'élément est une option de configuration, le parse est alors délégué par un deuxième parseur contenu dans la map d'**XMLConfigEntryParser** sinon il est directement analysé. Cela nous permet de gérer de manière transparente les différents éléments d'un fichier de configuration à partir de notre handler. Enfin cette architecture offre la possibilité d'ajouter des éléments dans le fichier de configuration en ayant simplement besoin de créer un nouveau parseur d'élément à partir de nos classes abstraites.

Couche simulator - Affichage



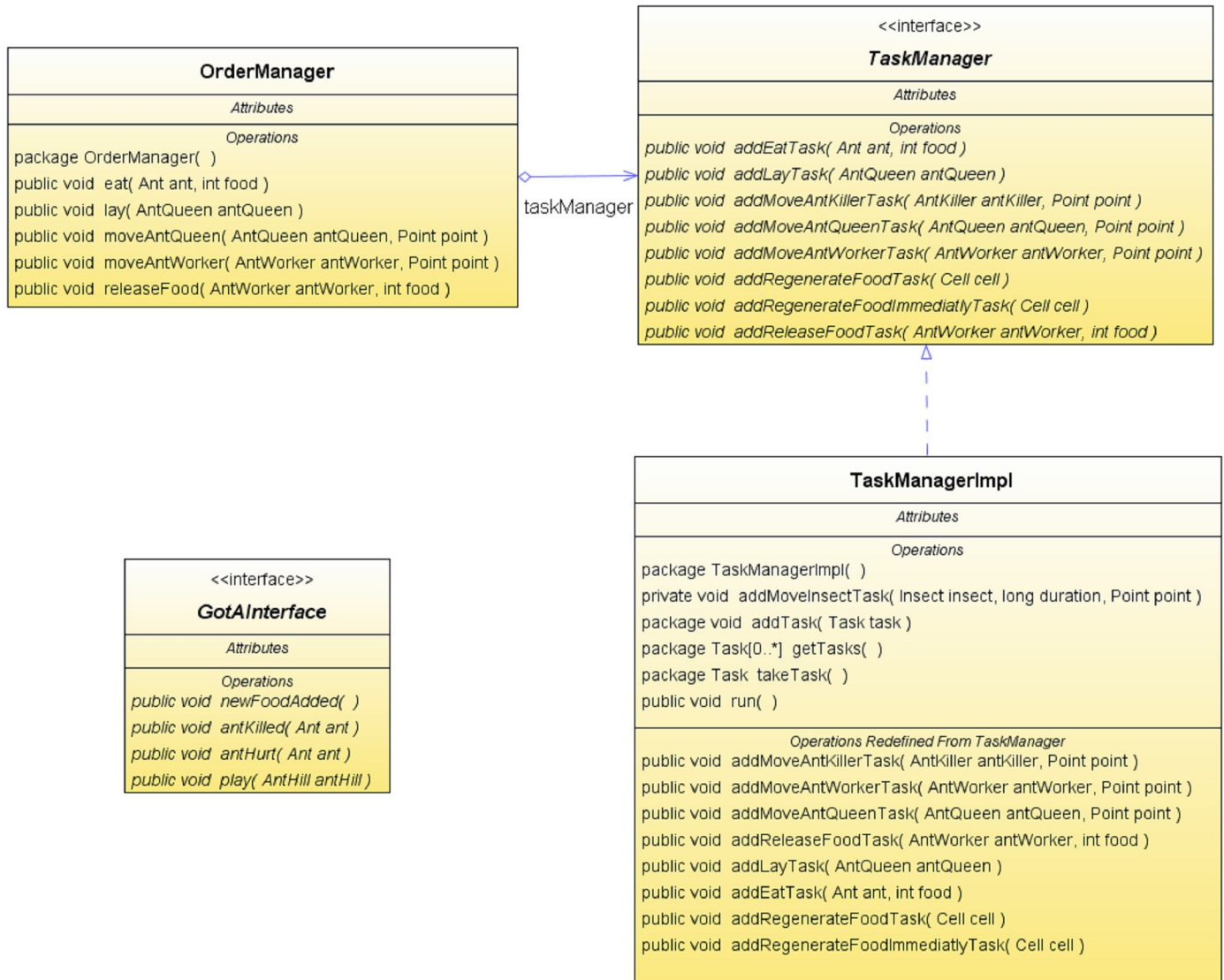
Nous avons souhaité la meilleure indépendance possible de notre package simulator avec le reste du code. Pour l’affichage nous utilisons la librairie **lawrence** mais nous voulions que l’affichage par une autre librairie nécessite le minimum de modifications. **DisplayManagerLawrenceImpl** est donc la seule classe qui interagit avec la librairie. On peut donc imaginer la création d’une nouvelle implémentation pour une autre bibliothèque graphique. L’utilisation d’une classe passerelle entre le package simulator et l’affichage réel permet de centraliser les demandes d’actualisation de l’affichage pour une case donnée. Un Thread est chargé d’actualiser l’affichage à une fréquence définie dans le fichier de configuration. Cela permet de réduire la consommation processeur.



Cette énumération permet de recenser toutes les images nécessaires par l’application pour l’affichage. Lors de l’affichage la méthode **getImages()** des cases(**Cell**) est appelée. Cette méthode retourne la liste des images lui correspondant (image de fond, caillou...) et ajoute également à cette liste le résultat de la méthode **getImages()** des **Insect** se trouvant sur cette case. Un insecte renvoie son image suivant la direction dans laquelle il se dirige et, à l’exception des tueurs, une image de nourriture si il en transporte et enfin un croix rouge si il perd de la vie.

Couche simulator - Relation gotA-simulator

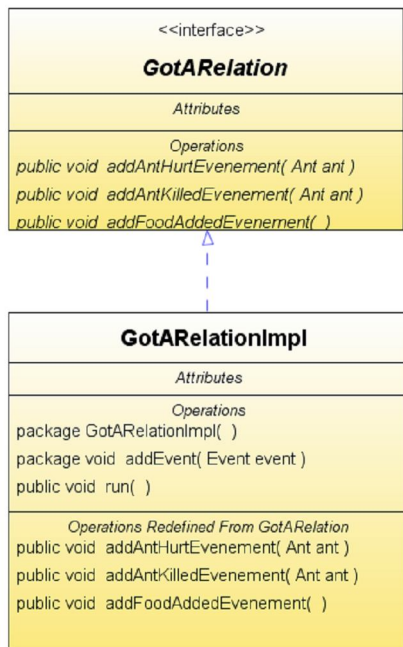
La gestion et l'exécution des tâches vues précédemment sont gérées par un **TaskManager**.



Pour permettre la meilleure indépendance possible entre le **gotA** et le package **simulator** nous avons mis en place deux parties. La première avec **OrderManager** qui est le seul moyen pour gotA d'effectuer des modifications sur la structure de données. La seconde avec la **GotARelation** qui est utilisée par la couche simulator pour informer le **GotA** des modifications de la structure.

Obliger **GotA** à utiliser **OrderManager** permet de contrôler les modifications qu'il souhaite apporter. **OrderManager** est une simple classe qui rend **public** certaines méthodes disponibles dans **TaskManager**. En effet **TaskManager** est de visibilité **package**. Grâce à ce système, le **GotA** ne peut pas (par exemple) demander la régénération de la nourriture sur une case. Ceci rend également transparent pour **GotA** la manière utilisée en interne pour traiter les ordres (en l'occurrence des tâches).

TaskManagerImpl ajoute toutes les tâches reçues dans une liste (**DelayQueue**) et un thread exécute les tâches lorsque le délai d'attente est terminé. L'exécution d'une tâche aurait pu se faire dans un Thread pour ne pas bloquer la lecture de la tâche suivante pendant l'exécution de la précédente. Nous n'avons pas retenu cette option, car le temps d'exécution d'une tâche est suffisamment court.

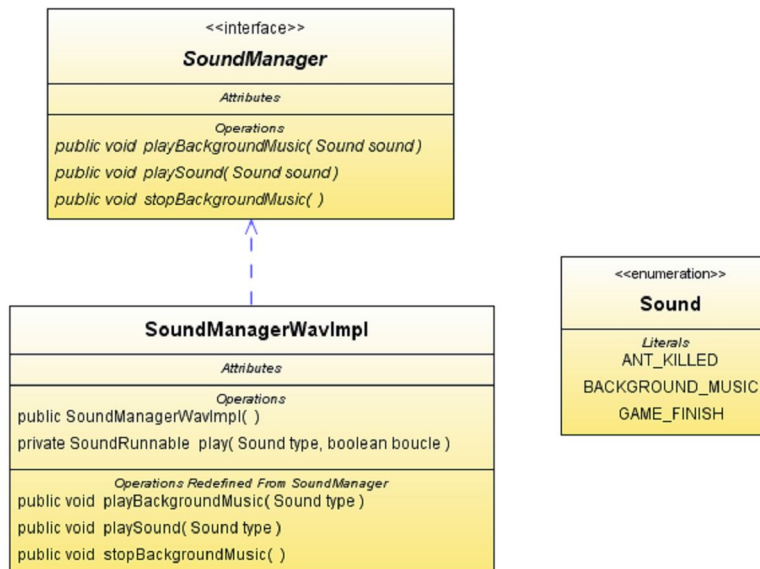


L'indépendance est effective dans les deux sens. La couche simulator prévient le **GotA** des modifications en passant par **GotARelation**. Cette classe stocke les informations dans une liste, et un thread les envoi au fur et à mesure au **GotA**. Nous avons fait ce choix pour éviter de bloquer la classe qui souhaite informer le **GotA** lorsque celui-ci traite l'information. En effet nous ne pouvons pas savoir le temps qui lui est nécessaire (il peut par exemple faire un **wait()** ou un **sleep()**).

Les classes du diagramme suivant sont des classes internes de **GotARelationImpl**. Elles lui permettent de gérer les différentes informations à retourner au **gotA**. Ces classes sont utilisées de la manière suivante : Une méthode de **GotARelation** est appelée (par exemple **addHurtEvenement(Ant antHurt)**) par une classe du package simulator. Un objet de la classe **Event** est alors créé et ajouté dans une **BlockingQueue**. Enfin, un thread permet lorsque un **Event** est ajouté dans cette liste de le récupérer et d'appeler sa méthode **execute()** qui appelle la bonne méthode du **GotA** pour l'informer de la modification. Ce principe ressemble fortement à notre gestion des **Task** à l'exception qu'ici l'événement n'a pas de temps d'attente avant d'être exécuté, mais dont sa durée d'exécution est inconnue car traité par le **GotA**. Nous aurions eu la possibilité de lancer un thread pour chaque appel aux méthodes de **GotA**, mais cette solution nécessite la création d'une trop grande quantité de Thread.



Couche simulator - Son



Nous avons décidé de mettre une musique de fond et de pouvoir (par exemple) associer un son lorsqu'une fourmi est touchée. **SoundManager** permet de jouer une musique ou un son. L'avantage d'avoir une interface a été vérifié dans ce cas. En effet nous avons fait une implémentation pour lire des fichiers Midi, mais le manque de ressources libre de qualité nous a poussé à créer une nouvelle implémentation. Nous en avons alors créé une pour lire les fichiers wave (.wav), ce qui est resté totalement transparent pour l'application. Nous avons juste eu besoin de changer l'instanciation de l'implémentation dans la **SimulatorFactory**. Pour lire un son, il suffit d'appeler une des deux méthodes **play...()** du **Manager** avec en paramètre une valeur de l'**enum Sound**. L'utilisation d'un **enum** permet d'associer à chaque valeur l'adresse physique des fichiers son. Il y a deux méthodes pour jouer un son, ce qui permet de dire explicitement à **SoundManager** lorsque l'on souhaite lire une musique de fond ou un bruitage. Dans le cas d'une musique de fond, celle-ci est jouée en boucle et peut également être arrêtée. Si une nouvelle musique de fond doit être jouée, l'ancienne est automatiquement arrêtée par le **Manager**.

Couche simulator - Exceptions

AnthillException
Attributes
Operations
package AnthillException(Throwable src, String msg) package AnthillException(String msg)

ConfigurationParseException
Attributes
Operations
package ConfigurationParseException(String message)

Ces classes correspondent aux deux types d'exceptions personnalisées dans le simulator. **AnthillException** est par exemple utilisée pour avertir le **gotA** lorsqu'il essaye de donner un ordre à une fourmi déjà morte.

ConfigurationParseException est utilisée lors du parse de la carte. Cette exception est générée lorsque le fichier de configuration défini par l'utilisateur n'est pas valide.

Les exceptions sont associées à un message qui offre plus de détails sur l'erreur.

Couche simulator - Thread principal

SimulatorThread
<i>Attributes</i>
private Thread otherThread[0..*]
<i>Operations</i>
package SimulatorThread(AntHill antHill, Thread otherThread[0..*])
private void startOtherThreads()
private void stopOtherThreads()
public void run()
private void moveAntKiller(AntKiller antKiller)
private void manageAntHurt(GotARelation gotARelation, AntHill antHill, Ant ant)
private void manageAntKillers(AntHill antHill)

Voici le thread principal du simulateur. C'est lui qui lance les autres threads de l'application et qui les arrête à la fin de la simulation. C'est également lui qui régit le déplacement des **AntKiller**, qui enlève les éventuels points de vie aux fourmis et qui donne l'ordre de régénérer les cases de nourriture lorsque cela est nécessaire.

Le **SimulatorThread** effectue des actions régulières que l'on peut résumer de manière séquentielle :

- Il ordonne à tous les tueurs sur la carte de se déplacer sauf si la case actuelle contient des fourmis
- Si un des tueurs est proche de la reine, il blesse la reine et dans le cas où celle-ci meure, le thread s'arrête et arrête tous ceux de l'application
- Si un des tueurs est proche d'une travailleuse, il la blesse et dans le cas où celle-ci meure, elle est retirée de la fourmilière et un bruitage est émis
- Il parcourt les cases de nourriture et vérifie si elles doivent régénérer de la nourriture.

Il prévient également, grâce au **GotARelation**, le **GotA** des événements suivant :

- l'apparition de nourriture dans la fourmilière
- la mort où la blessure d'une fourmi.

Couche simulator - Point d'entrée

Simulator
<i>Attributes</i>
<u>private boolean started = false</u>
<i>Operations</i>
private Simulator()
<u>public void start(GotAInterface gotA, String map)</u>

Cette classe est utilisée par le **main**. Elle vérifie les arguments et charge la carte passée en paramètre grâce au **DataManager**. Elle lance le thread principal avec les différents threads nécessaires à l'application et appelle la méthode **play** du GotA passé également en paramètre. Cette classe ne peut être appelée qu'une seule fois.

c) La couche util

Le package **util** est utilisé par **GotA** et la couche **simulator**. La classe **Point** permet de définir des coordonnées. La classe **Util** offre des méthodes sur des objets **Point** comme par exemple la distance entre deux points ou encore une liste aléatoire de points adjacents à un point donné. Ces méthodes sont principalement utilisées pour la génération aléatoire d'une carte.

Point
<i>Attributes</i>
private int x private int y private int hashCode
<i>Operations</i>
public Point(int x, int y) public Point(Point point) public int getX() public int getY() public boolean equals(Object o) public int hashCode() public String toString()

Util
<i>Attributes</i>
private Random random = new Random()
<i>Operations</i>
private Util() public Point getRandomPoint(Point point) public Point[0..*] getRandomAdjacentPoints(Point pointStart, int nb, int xMax, int yMax) public Point[0..*] getPointsPossibleAroundPoint(Point point, int xMax, int yMax, Point without[0..*]) public Point[0..*] getPointsPossibleAroundPoint(Point point) private boolean isPointValid(Point p, int xMax, int yMax) public T getRandomValue(T values[0..*]) public Point getBestPoint(Point origin, Point points[0..*]) public T getNextPoint(T value, T values[0..*]) public double distance(Point point1, Point point2)

ShortestPath
<i>Attributes</i>
<i>Operations</i>
private ShortestPath() private void addAdjacentCells(Point source, Map<Point, Boolean> map, Map<Point, Node> closedList, Map<Point, Node> openedList, Point endLocation) private Point bestNode(Map<Point, Node> openedList) private void addToClosedList(Point location, Map<Point, Node> openedList, Map<Point, Node> closedList) private void generatePath(Map<Point, Node> closedList, Point path[0..*], Node startNode, Point endLocation) public Point[0..*] calculateShortestPath(Map<Point, Boolean> map, Point startLocation, Point endLocation) public Point calculate(Map<Point, Boolean> map, Point startLocation, Point endLocation) private Point bestPointSimpliste(Map<Point, Boolean> map, Point points[0..*], Point origin, Point cible)

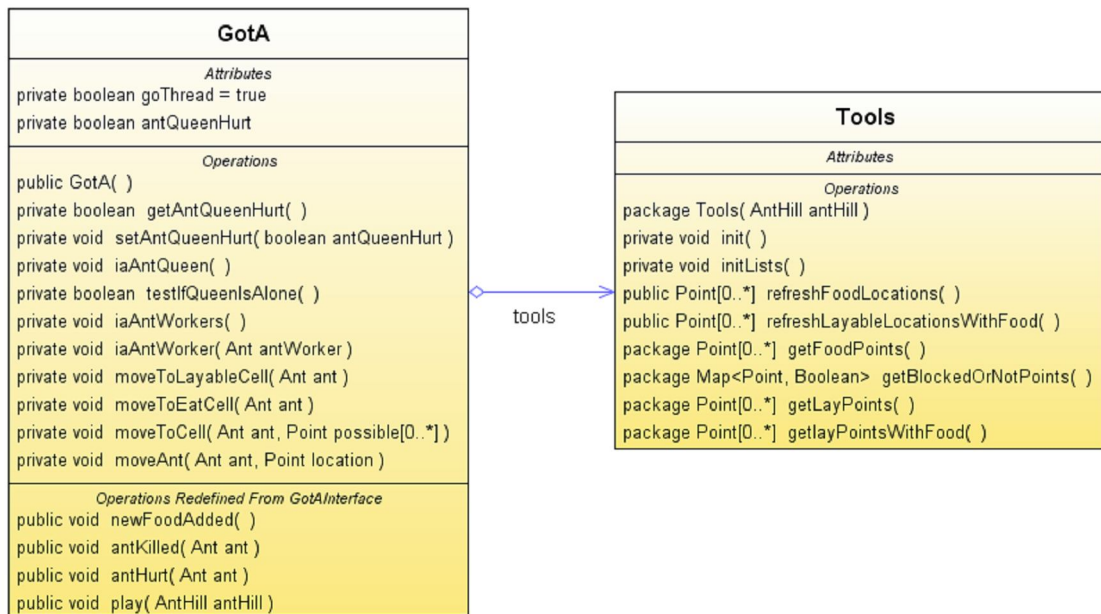
ShortestPath permet le calcul du plus court chemin entre deux points. L'algorithme utilisé est A-Star. Nous l'avons choisi pour sa vitesse d'exécution et son implémentation relativement simple. Comparé à d'autres algorithmes tel que Dijkstra, il ne garantit pas de trouver le chemin le plus court mais ceci n'était pas une priorité.

Il est possible de récupérer le parcours complet ou seulement la première case. Dans le cas de la demande de la première case on cherche un chemin à l'aide d'une approche naïve mais peu coûteuse en retournant la case la plus proche en terme de distance. Ceci permet de réduire le temps de calcul. Toutefois, cette méthode peut ne pas fonctionner notamment en présence d'obstacles. Dans ce cas, nous utilisons notre implémentation de l'algorithme A-Star. Cela réduit donc le temps de calcul, principalement sur des cartes avec peu d'obstacles.

Nous avons souhaité offrir la possibilité de retourner la première case d'un parcours au **GotA** pour faciliter son algorithme. En effet, comme demandé dans le sujet le **GotA** peut seulement déplacer une fourmi d'une seule case. Le calcul du chemin complet s'avère donc inutile, à moins qu'il ait une **map** avec en relation une fourmi et son chemin. Toutefois, si la fourmi est touchée par un insecte, la réaction logique du GotA est de déplacer la fourmi. Dans ce cas la suite du parcours peut être erronée. Il faut donc supprimer le parcours et en recréer un nouveau. Cette solution est donc au final plus coûteuse que de calculer à chaque fois la case la plus proche. La possibilité de récupérer la totalité du parcours reste néanmoins disponible.

d) La couche gota

Pour tester notre simulator nous avons créé un dieu des fourmis (**GotA**).



Notre **GotA** parcourt la liste des fourmis et leur assigne un ordre par l'intermédiaire de **OrderManager**. Un **GotA** a accès à toute la structure de données mais certaines méthodes ne lui sont pas disponibles. Dans notre cas, le **GotA** utilise une classe **Tools**. Cette classe prend en constructeur une **AntHill** et initialise plusieurs listes et maps qui seront ensuite utilisées par le **GotA**.

Ces listes et maps concernent par exemple les cases où de la nourriture a été détectée, les cases où la reine peut pondre, la map de toutes les cases avec un **boolean** pour savoir si elle contient un obstacle ou non (utilisé pour le parcours du plus court chemin). Certaines données sont bien sûr dupliquées dans plusieurs listes dans le but de ne pas les recréer à chaque fois. Cette façon se révèle plus performante. **GotA** peut, lorsqu'il le souhaite, actualiser les listes, notamment celles concernant les cases avec de la nourriture.

VII. Aspects techniques

Nous avons fait plusieurs choix pour réaliser l'application comme celui de gérer nous même les threads et de ne pas utiliser le package **lawrence**. Ce choix provient de la volonté d'apprendre à gérer les threads, ce qui n'a d'ailleurs pas été facile. De plus cela offre une grande indépendance avec la package **lawrence** en l'utilisant seulement pour l'affichage.

La librairie **lawrence** offre la possibilité d'utiliser des images vectorielles (*.svg). Nous avons choisi d'utiliser ces images car elles ont les mêmes avantages que les PNG et permettent, de plus, de ne pas perdre en qualité lors du zoom sur la carte.

La structure de données est composée d'objets entités avec pour seules méthodes des accesseurs mutateurs. En effet par définition un objet entité est un objet qui est non intelligent. Toutes les modifications que l'on souhaite apporter à la structure s'effectuent par l'intermédiaire de managers notamment par **AntHillManager**. Ceci permet d'avoir une classe « passerelle » entre la structure et le reste du code. L'avantage est par exemple, que nous avons seulement modifié **AntHillManager** lorsque nous avons souhaité ajouter des statistiques de création de fourmis. La structure définie au départ est donc restée inchangée.

La mise en place de Managers offre une grande flexibilité au code. En effet, les classes comme **DisplayManagerLawrenceImp** et **DataManagerXmlImpl** implémentent des interfaces utilisées dans le reste du code dont l'instanciation est récupérée par l'intermédiaire de **SimulatorFactory**. L'avantage est de pouvoir changer l'affichage ou encore le type de chargement de la carte en restant transparent par rapport au reste du code. Dans notre cas, la carte est chargée grâce à un fichier XML, mais on peut imaginer la charger à partir d'un fichier **properties** ou encore d'une base de données en créant simplement une nouvelle implémentation.

Le cœur du fonctionnement du projet se situe autour des tâches envoyées soit par le **GotA** ou bien le **Simulator** et centralisées dans une gestion réalisant ces tâches. La centralisation permet tout d'abord une maintenance plus aisée et dans le même temps une grande évolutivité. Evolution facile avec la possibilité d'implémenter de nouvelles tâches indépendamment de la gestion les réalisant.

Pour le stockage de données, notre choix s'est rapidement fait sur le langage de description XML pour plusieurs raisons :

- Un langage devenu à présent un standard utilisé dans de multiples applications et donc connu par le plus grand nombre d'utilisateurs/développeurs.
- Un langage adéquat avec notre faible volume de données, le problème de volume disque étant nul vu le peu de données nécessaires pour créer une fourmilière.
- Une technologie gérée nativement dans la librairie Java renforçant l'aspect indépendant de notre application (hors librairie graphique).

Notre application est basée sur une programmation multithread que l'on peut séparer ainsi :

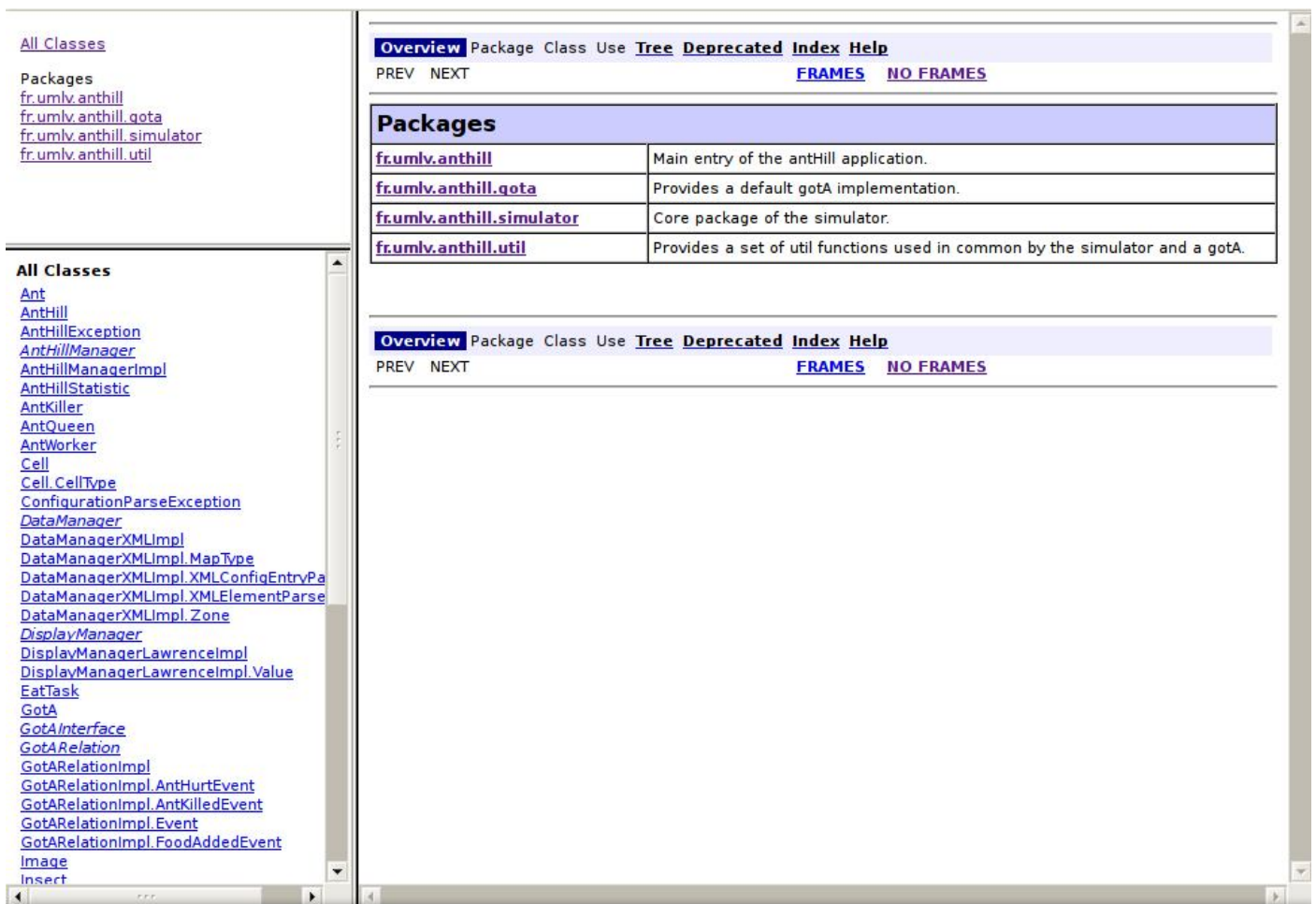
- Un thread principal (**SimulatorThread**) ayant la responsabilité de déplacer les **AntKiller** et blesser les **Ant**, de régénérer les cases possédant de la nourriture et de mettre à jour l'affichage.
- Un thread de gestion des tâches (**TaskManager**) qui a pour seul rôle de centraliser les tâches envoyées soit du **GotA**, soit du **Simulator**, et d'exécuter ces tâches si elle sont possibles.
- Un thread faisant la relation entre les événements de la fourmilière et prévenant le **GotA** de ceux-ci (**GotARelation**).
- Un thread ayant la responsabilité de lire des fichiers audio (*.wav)

Dans le cas de certains threads, nous avons commencé par utiliser des **sleep(1)** dans un **for(;;)** ce qui nous permettait par exemple de déplacer les tueurs toutes les millisecondes. Le problème de l'utilisation du **sleep()** est que l'attente est active. Nous avons donc utilisé des **Timer**, ce qui s'avère bien plus « propre ». En effet un **Timer** permet d'appeler une portion de code avec un intervalle choisi.

VIII. JAVADOC

Un projet doit être fourni avec une documentation. Elle a pour objectif de faciliter la reprise du code dans le futur. Elle peut être utile par la suite, lorsqu'une autre personne ou les auteurs eux-mêmes doivent modifier ou ajouter des fonctionnalités. Sans celle-ci, il est en effet difficile de reprendre un projet. En java, une documentation type est définie. Elle s'appelle la Javadoc et se présente sous forme de pages web. Une fois que la documentation a été écrite dans le code, l'outil javadoc permet de créer automatiquement les pages web. Ce dispositif offre l'avantage que toutes les JavaDocs sont conçues de la même manière. N'importe quel programmeur ayant déjà parcouru une Javadoc pourra rapidement trouver dans celle-ci l'information qu'il recherche. Pour que la documentation soit de qualité, des conventions sont à respecter lors de l'écriture des commentaires.

Voici une des pages de la Javadoc du projet.



The screenshot shows the Javadoc web interface for the AntHill project. The left sidebar contains a list of 'All Classes' and 'Packages'. The main content area displays the 'Packages' page, showing a table of packages and their descriptions.

Packages

Package	Description
fr.umlv.anthill	Main entry of the anthill application.
fr.umlv.anthill.gota	Provides a default gotA implementation.
fr.umlv.anthill.simulator	Core package of the simulator.
fr.umlv.anthill.util	Provides a set of util functions used in common by the simulator and a gotA.

IX. Ant

Un script ant est fourni et permet :

- La compilation des sources
 - ant compileCompilation des sources du dossier src dans le dossier classes.
- La création de antHill.jar
 - ant jarCette commande crée antHill.jar avec les **.class** du dossier classes, inclus les ressources dans antHill.jar, crée un dossier lib avec les librairies nécessaires et un dossier gota inclus par défaut dans le Class-Path du manifest . Tous les fichiers créés sont ajoutés dans un dossier dist.
- La génération de la javadoc
 - ant javadoc
- Le nettoyage du projet
 - ant clean

Le script ant est le fichier build.xml. Ce fichier utilise un fichier **.properties** nommé **build.properties** dans lequel est inscrit le nom du jar, du dossier source, etc.

X. Problèmes rencontrés

La partie d'analyse a subi de nombreuses modifications dues à certains éléments du cahier des charges pouvant être interprété de façons différentes.

Nous avons bien sûr rencontré des problèmes avec les threads. Nous nous sommes rendu compte qu'une application multithread stable est assez difficile à mettre en place. En effet, cela oblige à prendre des précautions notamment sur les variables pouvant être modifiées par plusieurs threads simultanément. Toute la difficulté de l'application multithread est de trouver ces endroits. En effet il est possible de lancer plusieurs dizaine de fois une application sans aucun problème et avoir un accès concurrent la fois suivante. L'arrêt des threads proprement à été également difficile car la documentation sur internet n'est pas forcément à jour (on voit encore beaucoup l'utilisation des méthodes dépréciées de la classe **Thread**). Malgré l'arrivée tardive du cours, il nous a permis de clarifier certains concepts sur les threads.

XI. Améliorations possibles

Nous avons respecté les consignes du cahier des charges mais il est possible d'imaginer les améliorations suivantes :

- Permettre aux fourmis de se déplacer en diagonale sur la carte
- Ajouter des contraintes sur les cases (tel que des cases ralentissant les fourmis ou les blessant)
- Des facteurs extérieurs à la fourmilière (tempêtes, tremblement de terre, jour/nuit)

XII. Conclusion

L'application est finie, fonctionnelle et répond au cahier des charges. La réalisation de celle-ci nous a permis d'appliquer les concepts vus pendant les cours et td du module Java Avancé ainsi que d'appliquer l'utilisation de design patterns vus dans le module de Génie Logiciel quand cela était utile et nécessaire.

L'évolution de notre projet à été constante et a bénéficiée d'une partie d'analyse où nous avons essayé de dégager le plus tôt possible une structure de données stable et une architecture évolutive.

Au niveau purement codage, la gestion du multithread est ce qui nous a posé le plus de problèmes avec la gestion des états invalides et des modifications concurrentes. Ce projet nous a permis de nous rendre compte des avantages et inconvénients de ce genre d'architecture.

Il a fallu garder en tête qu'une application externe (gotA) allait se greffer à notre simulateur et ainsi mettre en place une architecture indépendante et sécurisée par rapport à l'extérieur, sans fermer les possibilités offertes par nos interfaces.

Dans ce projet nous avons utilisé certains concepts vus en design pattern tels que les **factory** (dans le **DataManager**) ou **command** (le système des tâches et le **TaskManager**). Leur application a permis de factoriser notre code et d'augmenter la maintenabilité de celui-ci. L'emploi de ces designs s'est révélée particulièrement utile lors des séances de debuggage car ils permettent de centraliser les endroits « stratégiques » du code. L'application des patterns dans notre projet nous a donc permis de régler plus facilement certains problèmes.

XIII. Webographie

Nous avons principalement utilisé Internet et surtout la documentation de java pour mener à bien ce stage. Les adresses suivantes sont valides à la date du premier janvier 2008.

Documentation Java : <http://java.sun.com/javase/6/docs/api/>

Sites de programmation java : <http://java.developpez.com/>
<http://www.javafr.com/>

Pour utiliser le programme il est indispensable d'installer la machine virtuelle Java disponible à l'adresse suivante :

<http://www.java.com/fr/>

Nous avons utilisé NetBeans pour réaliser le programme ainsi que sont module UML. Cet outil libre et très performant peut être téléchargé à cette adresse :

<http://www.netbeans.info/downloads/index.php>

La musique de fond utilisée est libre et à été téléchargée sur le site suivant

<http://www.jamendo.com/fr/>