École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
4e année
2014 - 2015

Rapport de projet PIL

Développement d'un serveur de jeu pour l'initiation à la programmation

Encadrants
Nicolas MONMARCHE
nicolas.monmarche@univ-tours.fr
Sébastien AUPETIT
aupetit@univ-tours.fr
Carl ESSWEIN
carl.esswein@univ-tours.fr

Université François-Rabelais, Tours

Etudiants
Olivier BUREAU
olivier.bureau@etu.univ-tours.fr
Anthony DEMUYLDER
anthony.demuylder@etu.univ-tours.fr
Nicolas GOUGEON
nicolas.gougeon@etu.univ-tours.fr
Jonathan LEBONZEC
jonathan.lebonzec@etu.univ-tours.fr
Benjamin MAUBAN
benjamin.mauban@etu.univ-tours.fr
Juliette RICARD
juliette.ricard@etu.univ-tours.fr
Minghui ZHANG
minghui.zhang@etu.univ-tours.fr

DI4 2014 - 2015

Version du 31 mai 2015

Table des matières

Table des figures

Liste des tableaux

Introduction

Ce rapport a pour objectif de décrire les différentes étapes du projet "Développement d'un serveur de jeu pour l'initiation à la programmation". Le but final de ce projet étant la possibilité pour des personnes souhaitant s'entraîner au développement de bots (Intelligence Artificielle) le fassent par l'intermédiaire de l'application à développer.

Ce projet a été proposé par M. Nicolas Monmarché et est également supervisé par M. Carl Esswein et M. Sébastien Aupetit.
La Maitrîse d'Œuvre est représentée par le groupe 3 PIL 2015 de Polytech'Tours, composé de :

- Olivier Bureau
- Anthony Demuylder
- Nicolas Gougeon
- Jonathan Le Bonzec
- Benjamin Mauban
- Juliette Ricard
- Minghui Zhang

Ali Aalaoui a quitté le projet quelques semaines après le début de celui-ci. Il n'a donc pas participé au développement.

Contexte et Environnement du projet

Un jeu en ligne est un jeu vidéo sur lequel des joueurs du monde entier peuvent s'affronter sur une même partie de jeu.
Au cours des dernières années, le jeu en ligne s'est popularisé. On compte aujourd'hui des milliers de plateformes de jeu en ligne, chacune proposant des jeux avec des gameplays très différents.
Cependant tous ces jeux ont un point commun : seuls les joueurs réels sont invités à jouer. Mais depuis quelques temps, on voit apparaître des plateformes de jeu destinées spécialement aux bots : c'est l'ordinateur qui joue.
Le challenge est très intéressant car pour pouvoir jouer au jeu, il faut d'abord réfléchir à une intelligence artificielle qui sera utilisée par l'ordinateur pour survivre dans la partie, puis programmer le bot.

Vindinium et AIChallenge sont deux plateformes de ce type. Cependant, la première présente quelques défauts de conception dans les règles du jeu et la deuxième n'est pas utilisable de manière directe. C'est pourquoi nous cherchons à créer une plateforme de jeu qui prendrait les qualités qu'on retrouve dans chacun des 2 jeux.

Le projet consiste à créer un Serveur de Jeu qui puisse implémenter un jeu tour par tour avec plusieurs joueurs. Par exemple on doit être capable de jouer une partie de jeu d'échec ou d'un équivalent à AIChallenge si le jeu est implémenté. Un "joueur" joue en réalité par l'intermédiaire d'un bot : un programme tourne sur sa machine et est capable de choisir les actions à effectuer.

Objectifs

Il s'agit de créer un serveur de jeu sur lequel des bots pourront s'affronter dans des parties de jeu de type tour par tour. Le déploiement de cette plateforme permettra aux étudiants du département Informatique de s'initier à la programmation. En effet ils pourront élaborer une intelligence artificielle en programmant un bot dans le langage de leur choix. Ils pourront ensuite tester leur bot en le faisant jouer sur le serveur et disposeront d'un retour leur permettant d'améliorer leur programme.
Il faudra aussi rédiger une documentation et un exemple de bot pour que les utilisateurs puissent créer leur bot facilement.

Eléments techniques

4.1 Contraintes techniques

Pour réaliser ce projet, il nous a été imposé plusieurs contraintes au niveau de la conception et des outils de gestion que nous devions utiliser.

Au niveau du langage, c'est le Java qui nous a été imposé. Ce langage permet la portabilité du système grâce à la machine virtuelle Java. Il nous a aussi été fortement conseillé d'utiliser Maven (cf Annexe B) comme gestionnaire de dépendances.
Le protocole de communication qui nous a été imposé est le JSON (cf Annexe B). Celui-ci est utilisé pour formater tous les messages envoyés entre le client et le serveur. Il a l'avantage d'être nativement interprété. Sa particularité est que les informations sont accompagnées d'une étiquette permettant de les interpréter. Enfin, on nous a demandé d'utiliser SLF4J (cf Annexe B). C'est une API de framework qui permet de créer un historique des évènements apparus sur le système. Cela est très utile pour débugger ou repérer les anomalies du système.

Au niveau du jeu, Nicolas Monmarché nous a demandé de nous inspirer des règles du jeu des fourmis d'AI Challenge (cf Annexe B) en procédant à quelques modifications.
De plus, dans son fonctionnement, AI Challenge charge le code des bots et l'exécute sur son serveur afin de connaître leurs mouvements. Pour des raisons de sécurité il nous a été imposé d'envoyer les actions des bots via un client que chaque utilisateur devra créer. Ce fonctionnement est inspiré du jeu Vindinium (cf Annexe B) un autre jeu d'intelligence artificielle.

Toutes ces contraintes nous ont plus aidés que ralentis car elles conviennent à ce projet et à l'ensemble du système. De plus cela nous a évité d'utiliser des outils inappropriés qui nous auraient fait perdre du temps sur l'ensemble du projet.

4.2 Décisions du groupe

Ensuite, nous avons dû prendre quelques décisions concernant les outils de gestion de projet.

Premièrement, nous avons choisi d'adopter une méthode Agile durant la réalisation du projet. Cela nous a permis de créer les tâches sous forme de bot stories. Chaque bot story représente des fonctionnalités qu'un bot peut utiliser durant la partie à la manière des cas d'utilisation. C'est selon cette méthode que nous avons découpé notre projet en tâches correspondantes à chaque bot story. Nous détaillerons plus tard les résultats de notre méthode. Nous avons privilégié une méthode Agile par rapport à une méthode de cycle en V afin de pouvoir modifier certains paramètres en cours de projet. De plus la taille de notre projet n'était pas adaptée à un cycle en V.

Pour la gestion des versions, nous avons utilisé GitHub (cf Annexe B) via un dépôt privé ce qui nous a permis de travailler à plusieurs sur le projet sans générer trop de conflits. Pour l'attribution des tâches et le suivi du projet, c'est Trello (cf Annexe B) qui nous a été utile surtout dans le début du projet.

Au niveau du jeu des fourmis les règles nous ont été principalement dictées par l'existant d'AI Challenge. C'est un des points qui nous a posé problème au début du projet car nous nous concentrions sur le choix de ces règles et l'impact qu'elles auraient sur le développement (Que voit une fourmi sur le terrain ? Comment gère-t-on la nourriture ?). Grâce à une réunion avec les encadrants, nous avons décidé d'établir rapidement les règles du jeu pour pouvoir se concentrer sur autre chose. C'est pourquoi nous avons choisis de réutiliser les règles d'AI Challenge, jeu qui a déjà été testé et dont l'équilibre des règles a été éprouvé.

4.3 Spécifications

La rédaction des spécifications nous a demandé un temps important car énormément d'éléments nous étaient inconnus. Le résultat de cette partie est décrit dans le cahier de spécifications livré en annexe. Cependant, il y a eu beaucoup de modifications apportées durant le projet sur le diagramme de classe.

Tout d'abord, travailler sur le serveur de jeu nous a permis de mieux comprendre l'architecture de l'ensemble du projet et ainsi d'améliorer le diagramme de classe.

Au niveau des composants, nous avons pu supprimer des redondances en intégrant le serveur d'inscription au serveur de jeu. En effet dans le cahier de spécifications nous voulions développer un serveur d'inscription qui inscrirait dans les base de données les nouveaux joueurs. Or le serveur de jeu à lui aussi besoin d'avoir accès à la base de données pour la gestion des scores et l'identification des bots.

4.4 Découpage des tâches

Comme nous l'avons déjà évoqué, nous avons découpé les tâches selon des bots stories qui correspondent aux cas d'utilisations.
Cependant ce découpage n'a pas été le meilleur car l'estimation temporelle de ces bots stories était bien différente de la réalité. Cela s'explique par le fait que chaque bot story nécessite une partie structure décrite dans le diagramme de classe et une partie dite fonctionnelle. Par exemple la bot story foodspawn nécessite d'avoir implémenté la structure via la classe *AntGameObject* et la classe *AntFoodSpawn*.
De plus, la partie fonctionnelle est implémentée dans la classe *AntGame*. Nous avons donc préféré établir la structure de tous les objets dans une première tâche, les interactions des objets dans une seconde tâche. Nous avons donc validé plusieurs bot stories en un seul coup contrairement ce qui avait été prévu dans le diagramme de Gant. Le planning n'a pas pu être respecté car il n'était tout simplement pas adapté. Mais finalement, les fonctionnalités définies au début du projet sont toutes présentes et toutes les bot stories sont validées.

Dans les spécifications, nous avions mis en place un diagramme de séquence décrivant le déroulement d'un tour de jeu. Dans ce diagramme, soit tous les joueurs envoient leurs actions durant la même période soit les joueurs jouent les uns après les autres (comme dans un jeu d'échecs). Pour le jeu des fourmis, nous avons choisi la première option car c'est celle qui convient le mieux par rapport aux règles définies par AI Challenge. Le mode de jeu "l'un après l'autre" a finalement été abandonné en accord avec Mr Monmarche.

4.5 Ajouts par rapport au cahier de spécifications

Une nouvelle bot story à été ajoutée en cours de projet, il s'agit du visualiseur. Ce dernier permet à l'utilisateur de revoir le déroulement d'une partie. Cette application consiste en une console qui affiche les vision des fourmis via un alphabet de caractères. Cela nous a été utile pour débugger notre programme et s'assurer du bon fonctionnement général. Il se base sur celui déjà présent sur le site d'AI challenge.

Architecture finale

Dans cette partie nous allons présenter l'architecture actuelle du serveur de jeu des fourmis (serveur de jeu générique + implémentation du jeu des fourmis), discuter de ses points forts et de ses points faibles et tenter d'expliquer comment nous en sommes arrivés là en détaillant les choix techniques qui ont été faits. Cette partie offre un point de vue général sur l'application telle qu'elle est actuellement et est un bon point de départ pour ceux qui souhaite la reprendre pour l'améliorer.

5.1 Évolution de l'architecture

L'architecture du serveur de jeu a énormément évoluée depuis les spécifications. Nous avions eu beaucoup de mal à imaginer une telle structure, d'une part à cause de nos compétences Java qui étaient trop faibles et d'autre part parce que nous n'avions jamais programmé de serveur auparavant. Il y a certains points que nous avions plutôt bien anticipés : la rédaction d'un protocole, la création de threads pour écouter plusieurs bots simultanément, la création de threads pour faire tourner les jeux simultanément, un thread pour créer les jeux lorsque possible, etc.

Le diagramme de classe initial du serveur de jeu est disponible page 14 (aperçu) ou sur le lien disponible en Annexe B.

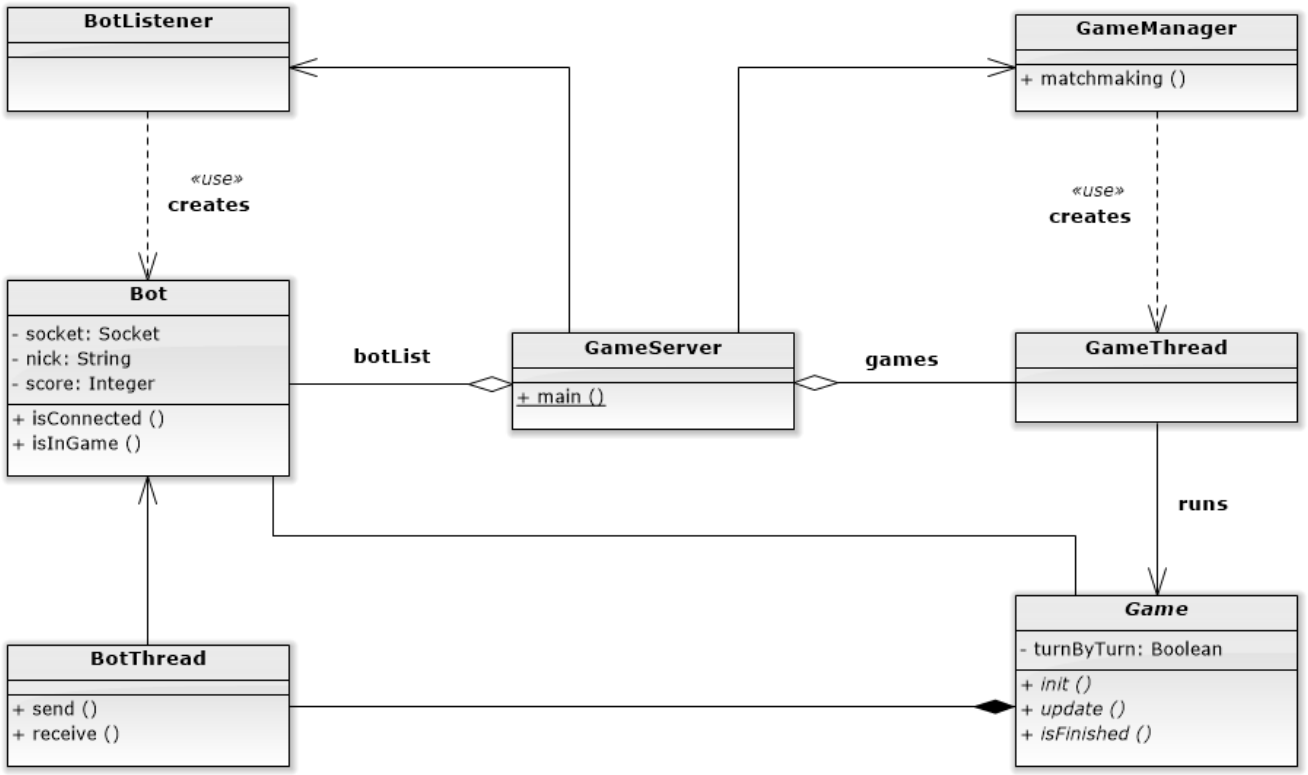
Pour rappel, nous avions une classe *Bot* qui contenait les données du Bot, une classe *BotThread* qui gère les communications avec le bot au cours d'une partie, une classe *BotListener* chargée d'écouter les bots se connectant sur le serveur, un classe *GameManager* qui crée des parties lorsque des bots sont disponibles pour jouer, une classe abstraite *Game* qui définie les propriétés et méthodes d'un jeu, une classe *GameThread* chargée de "piloter" un jeu (initialisation, réception des actions de jeu, mise à jour du plateau, etc...) et enfin une classe *GameServer* qui fait le lien entre tous ces composants.
On peut déjà remarquer que nous avions oublié une classe qui s'occupe de la communication avec la base de données, les informations des bots étant stockées dans une base de données MySQL (cf Annexe B).
D'une part, il n'y a aucun composant qui témoigne de la présence d'un mode entraînement.
D'autre part, il y a un point qui a été très mal anticipé et qui nous a forcé à revoir toute la structure. Il s'agit de l'absence de la notion de client. En effet, un bot qui n'est pas connecté ne peut pas jouer sur le serveur, mais il peut quand même communiquer avec le serveur. Car il faut bien que le client échange avec le serveur pour pouvoir se connecter. Or, notre classe *BotThread* qui gère la communication avec le bot implique que le bot soit connecté sur le serveur.
Nous avons donc réintroduit la notion de client :
"Un client est un noeud réseau qui envoie des requêtes au serveur, qu'il soit connecté avec un bot ou non."
À partir de maintenant, un bot qui n'est pas connecté est un client. Un client peut envoyer un message de connexion s'il souhaite se connecter avec un bot.
Il y a un autre point qui n'est pas clairement explicité dans le cahier de spécifications et qui nous a forcé à revoir lourdement la gestion des communications avec le client. Il s'agit de la façon dont le serveur de jeu intègre le protocole.
Au départ, nous avions une vision du serveur qu'on pourrait qualifier de "descendante". C'est à dire que c'était le serveur qui décidait à quel moment il écoutait le client et quel type de message il s'attendait à recevoir. Par exemple, lorsqu'un bot jouait à un jeu, le serveur n'écoutait pas le client tant que ce n'était

pas son tour, et lorsqu'il l'écoutait, il s'attendait à recevoir des actions de jeu uniquement. Cela semble intuitif, mais nous avions oublié que le serveur et le client communiquent sur la base d'un protocole et que le serveur est censé écouter en permanence les messages du client et lui répondre une erreur lorsque le message ne respecte pas le protocole, ou lorsque l'état actuel du bot ne lui permet pas d'envoyer ce type de message...
En bref, nous ne modélisions pas la façon dont fonctionne un serveur. Et cela allait nous poser de gros problèmes par la suite ! Nous avons alors revu le fonctionnement des communications sur le serveur et créé une architecture qu'on pourrait qualifier d'"ascendante".

Dans cette nouvelle architecture, le client et le serveur respectent à la lettre le protocole. Le client peut envoyer n'importe quel message à n'importe quel moment. Le serveur écoute ses clients en permanence. Lorsque le serveur reçoit un message, il va l'analyser, et va vérifier si le message a du sens / est pertinent au regard de l'état actuel du bot dans le serveur. Par exemple, si le client envoie des actions de jeu, le serveur va, dans un premier temps vérifier que le client est bien connecté en tant que bot, puis que le bot est bien en train de jouer à jeu, et enfin que le jeu est bien en attente d'actions de jeu pour ce bot.
Par ailleurs, cette vision du serveur nous a renforcé dans l'idée de supprimer le serveur de token. En effet, le design du serveur de token aurait été très similaire au serveur de jeu, et on aurait introduit une redondance entre le serveur de jeu et le serveur de token, en particulier au niveau de la gestion des communications avec le client, qui aurait été dupliquée.
La création d'un token peut désormais se faire via l'envoie d'un message de type "token" par le client. Par conséquent, le client d'inscription perd un peu de son utilité puisqu'en général on ne crée qu'un seul bot, mais pas tant que ça, car on ne créer en général qu'un seul bot pour jouer. Or si on ne fait pas attention en programmant son bot, on peut faire l'erreur de créer un nouveau bot à chaque fois qu'on lance le programme. De manière générale, il vaut mieux que le token soit codé en dur dans le code source du bot.

Au niveau des jeux, nous avons gardé le *GameManager*, le *GameThread* ainsi que la classe abstraite Game. La mécanique du *GameThread* reste la même que celle définie dans les spécifications. Cependant, nous avons enlevé la propriété 'turnByTurn' du jeu (qui avait été renommée *OneBotPerTurn* plus tard...), et donc la branche associée dans le diagramme d'activité du *GameThread* a disparue. Cette propriété avait pour but de différencier les jeux pour lesquels tous les bots jouent en même temps (comme dans le jeu des fourmis), de ceux où un seul bot joue à la fois (comme aux échecs par exemple).
Nous avons choisi de supprimer cette caractéristique car il s'agissait d'un cas particulier parmi d'autres. En effet, on peut aussi imaginer un jeu pour lequel deux bots ont le droit de jouer à chaque tour. Pour permettre cette éventualité, nous avons ajouté pour chaque bot, une propriété qui indique s'il peut jouer pour le tour actuel ou non. Ainsi, c'est l'implémentation de la classe *Game* qui décide qui doit jouer à quel moment. Le *GameThread* est programmé de telle sorte que le jeu est mis à jour lorsque tous les bots censés jouer pour le tour actuel ont joué. Par défaut, le jeu demandera à tous les bots de jouer à chaque tour.

FIGURE 5.1 – Aperçu du diagramme de classes initial



5.2 Serveur d'inscription

Lors de la phase de spécifications il était prévu de réaliser un serveur d'inscription et un client d'inscription indépendants du serveur de jeu.
Lors de la réalisation nous nous sommes aperçus qu'il était bien plus simple d'accéder à la base de donnée en passant directement par le serveur de jeu.

5.3 Serveur de jeu

La classe *TCPClientCommunicator* est peut-être la plus importante du serveur. Elle modélise la notion de client et propose des solutions pour communiquer avec celui-ci. Quand on jette un œil aux méthodes de cette classe, on peut voir les méthodes d'envoi et de réception de messages sont directement calquées sur le protocole. Cela donne une couche d'abstraction pour les communications avec le client au niveau du serveur.
À noter que la classe *FakeCommunicator* hérite de *TCPClientCommunicator* et modélise un faux client permettant d'implémenter un bot qui renvoie immédiatement des actions de jeu côté serveur. Ce composant a été introduit pour gérer le mode entraînement du jeu des fourmis. Il n'est absolument pas utilisable tel quel mais se doit d'être présent dans les composants liés au serveur de jeu.
Nous vous avouons que ce système a été assez mal modélisé dans le serveur de jeu car ce composant a été intégré trop tard dans l'architecture. Avec le recul, il aurait fallu créer une classe abstraite *Communicator* dont hériterait les classes *ClientCommunicator* et *FakeCommunicator*, chacune proposant des méthodes appropriées à chaque type de communication. En tout cas, l'idée de réutiliser cette classe pour créer des bots côté serveur était ingénieuse.

La classe chargée des interactions avec la base de données est *DBManager*, autrefois nommée *DBInterface* mais renommée à cause de la similarité avec le mot clé "interface" de Java. C'est un singleton. Toutes les requêtes SQL utilisées par le serveur sont préparées à la création de la classe. À noter que la génération de token ne se fait pas sur le serveur mais directement dans la base de données via des procédures MySQL.

La classe *BotGameInfo* est une classe très importante pour la compréhension du fonctionnement du serveur de jeu. Cette classe contient les informations propres au jeu qu'un bot est en train de jouer. C'est d'ailleurs cette classe qui indique si le bot peut jouer pour le tour actuel ou non.
Le tampon 'gamestateTimestampMs' permet de mémoriser l'instant auquel nous avons envoyé les actions de jeu au bot. Ainsi, nous savons dire si le bot a mis trop de temps à répondre ses actions de jeu. Auquel cas, la classe contient une propriété 'muted' qui permet de désactiver l'écoute du bot jusqu'à la fin du jeu. Un score de jeu permet de déterminer le gagnant d'une partie et de mettre à jour le score général du bot dans la base de données. On a aussi ajouté un identifiant de jeu pour un bot car cela peut être utile pour référencer le bot dans des fichiers de replay par exemple...
Le programmeur qui souhaite implémenter son jeu dans le serveur est invité à surcharger cette classe pour y ajouter des informations propres à son jeu (par exemple la liste des unités que contrôle chaque bot).

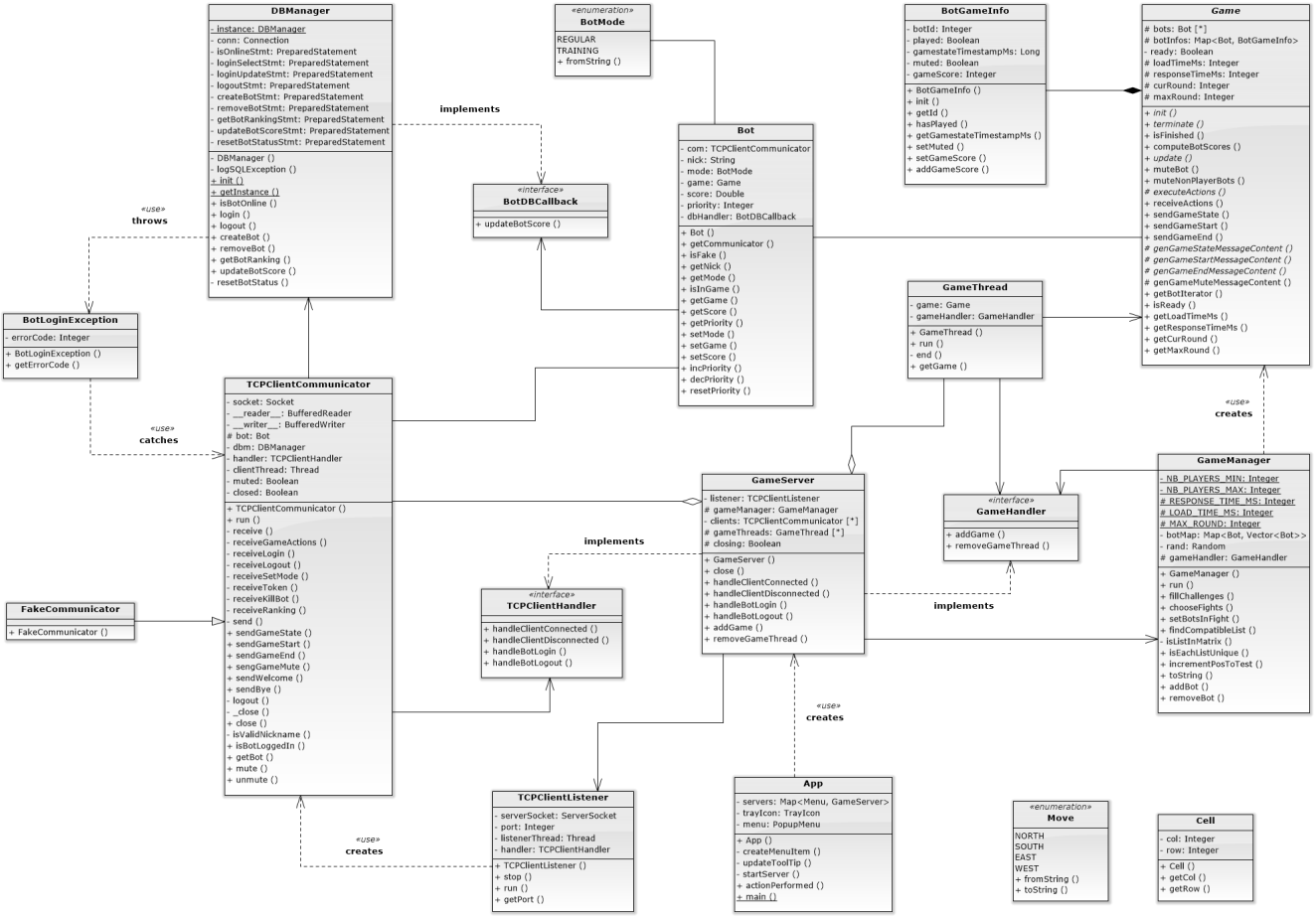
Le *GameManager* est la classe chargée de créer les parties pour les bots. La création des parties devant se faire en même temps que l'écoute des clients et la mise à jour des jeux, nous avions prévu d'implémenter le *GameManger* sous forme de Thread, mais nous nous sommes rendus compte qu'il valait mieux le programmer sous forme de tâche exécutée de manière périodique car il n'est pas nécessaire de créer les jeux en temps réel. En effet, lorsqu'un bot attend pour rejoindre une partie, il n'est pas à 5 secondes près. De plus, cela permet de libérer de la ressource sur le serveur.

GameServer est la classe centrale du serveur de jeu. C'est elle qui fait le lien entre les clients (ou bot lorsque le client est connecté), les parties, et le créateur de partie.

La classe *App* s'occupe de lancer le serveur de jeu et ajoute une petite icône dans la barre de tâche permettant de contrôler l'application une fois lancée (accès aux logs et fermeture du serveur).

Enfin, nous avons aussi ajouté deux entités souvent utilisées dans les jeux de plateau : Move qui décrit un mouvement sur le plateau (nord, sud, est, ouest) et Cell qui localise une cellule en 2 dimensions.

FIGURE 5.2 – Aperçu du diagramme de classes du Serveur de Jeu



5.4 Moteur de Jeu

Nous allons maintenant analyser l'architecture actuelle du moteur de jeu pour l'implémentation du jeu des fourmis. Nous n'avons pas fourni de spécifications par rapport à l'implémentation du jeu, mais nous avions beaucoup réfléchi aux différents composants et à la manière de les implémenter.
Le développement du moteur de jeu s'est donc fait de manière directe, en surchargeant les classes du serveur de jeu et en créant de nouvelles classes lorsque nécessaire. D'un point de vue général, nous pouvons découper notre moteur de jeu en deux parties :

— le modèle, qui contient toutes les données du jeu actuellement joué
— le contrôleur, qui est chargé de mettre à jour ces données en fonction des actions de jeu envoyées par les bots

AntMapTemplate représente un modèle de carte. Cette carte ne permet pas de jouer au jeu des fourmis, elle donne simplement la structure initiale de la carte. Elle définit les dimensions de la carte, la positions des nids, des points de nourritures et des murs. C'est en quelque sorte le stockage du fichier de map en mémoire. D'ailleurs les modèles de cartes sont chargés au démarrage du serveur par la classe *AntGameServer*, spécialement surchargée de *GameServer* pour gérer le chargement des fichiers de map.

Pour pouvoir jouer sur une carte, il faut initialiser une classe *AntGameMap* à partir des données présentes dans le modèle de carte. La classe *AntGameMap* modélise une carte dynamique contenant les positions de objets de jeu en temps réel. Autrement dit, c'est une carte sur laquelle on peut jouer.
Cette carte dynamique est implémentée sous forme de tableau à 3 dimensions. On a un tableau de lignes, chaque ligne contient un tableau de colonnes et chaque colonne contient un tableau d'objets de jeu. En effet, il est possible qu'il y ait plusieurs objets de jeu dans une même cellule. Par exemple, il peut y avoir une fourmi sur un nid ou encore, une fourmi en train de ramasser de la nourriture sur un point de nourriture. Par contre, il ne peut y avoir deux fourmis dans une même cellule mais c'est à la classe *AntGame*, qui pilote le jeu, de s'assurer de la conformité de l'état du jeu à chaque tour. La responsabilité de la carte dynamique s'arrête au stockage des objets de jeu. Cela semble maladroit de procéder ainsi mais nous n'avons pas le choix car nous ne pouvons pas prévoir à l'avance les mouvements des fourmis. Par exemple, si on prend deux fourmis appartenant à un même bot et étant positionnées sur deux cases adjacentes, il est possible que le bot déplace la première fourmi sur la deuxième mais qu'il déplace la deuxième d'une case plus loin. Or si la carte était en charge de vérifier sa conformité après chaque mouvement, alors la deuxième fourmi aurait été supprimée.

AntGameObject représente un objet de jeu positionné sur le plateau. Nous avons créé 4 sous-classes : *Ant*, *AntFoodSpawn*, *AntHill* et *AntWall* pour gérer les fourmis, les points de nourritures, les nids et les murs respectivement. Les objets de jeu peuvent être déplacés sur la carte à l'aide de l'interface *AntGameMapCallback* qui est implémentée par *AntGameMap* et qui procure une méthode permettant de rendre le déplacement effectif sur la carte. En effet, pour déplacer un objet, il ne suffit pas de changer ses indices de ligne et de colonne. Il faut aussi retirer l'objet de sa cellule actuelle et le rajouter dans sa nouvelle cellule sur la carte.

Pour gérer le brouillard de guerre dans le jeu, nous avons créé une classe *AntGameMapMask* qui modélise un masque que l'on peut appliquer sur la carte de jeu pour sélectionner certaines cellules. Un masque est en fait un ensemble de cellules définies par rapport à la cellule (0, 0). On peut créer un masque circulaire à partir d'un rayon et les cellules sont sélectionnées automatiquement.
La carte du jeu est programmée de façon à gérer l'aspect toroïdal. C'est-à-dire qu'elle est répétée à l'infini sur chaque bord. Ainsi, nous pouvons sélectionner n'importe quelle cellule de la carte sans se soucier des limites.

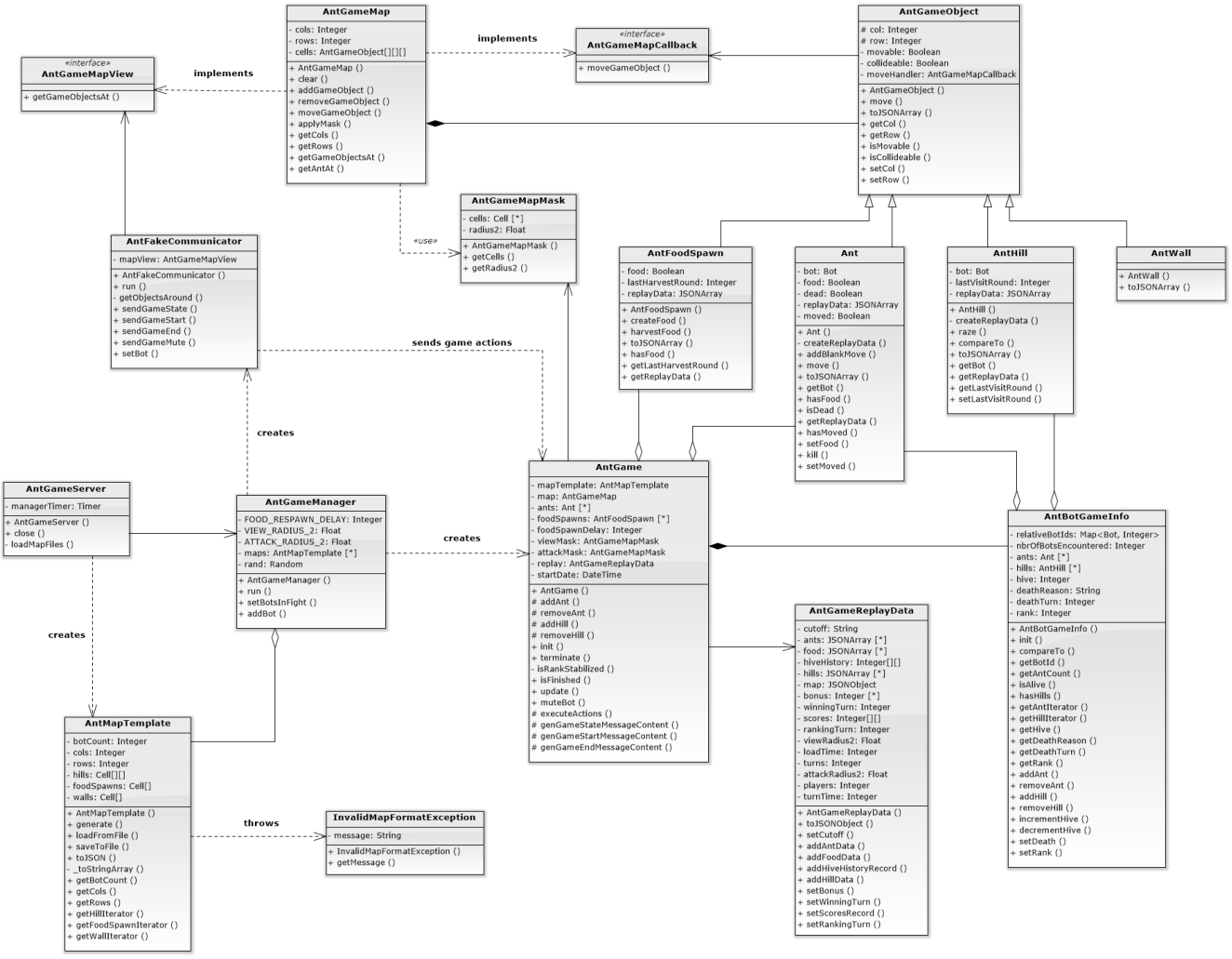
Les masques 'visionMask' et 'attackMask' ont été utilisés pour sélectionner les objets vus par une fourmi, et pour résoudre les combat entre fourmis dans la classe *AntGame*. Nous stockons les objets vus par l'ensemble des fourmis d'un bot dans un HashSet pour éviter d'enregistrer plusieurs fois la même unité.

Par ailleurs, comme nous l'avons vu dans l'architecture du serveur de jeu, *AntFakeCommunicator* est la classe chargée de jouer contre un bot en mode training. Ici, elle utilise l'interface *AntGameMapView* qui lui donne une vue complète sur la carte de jeu, ce qui lui permet de répondre des mouvements appropriés.

Enfin, l'intégration des fichiers de replay dans le moteur de jeu n'a pas été facile car il faut suivre la trace de chaque objet/score de jeu au cours de la partie et cela nous a demandé d'ajouter un bon nombre de propriétés et de méthodes dans nos classes d'objets de jeu.
Voici le fonctionnement général : *AntGameReplayData* garde la trace de toutes les données de jeu. Sachant que nous n'avons pas obligatoirement toutes les informations de replay nécessaire pour un objet de jeu lors de sa création (par exemple, le tour de mort pour une fourmi), une référence vers ces informations de replay, appelée 'replayData', est stockée dans la classe de l'objet de jeu en question. Lorsque de nouvelles informations sont disponible, les données de replay sont mises à jour dans *AntGameReplayData* en passant directement par l'objet de jeu, ce qui nous évite d'avoir à chercher l'information à mettre à jour dans l'instance de la classe *AntGameReplayData*.

Nous avons fait un tour d'horizon de l'architecture actuelle du serveur de jeu et du moteur de jeu. Pour plus de détails techniques, n'hésitez pas à lire la documentation (javadoc) du serveur, disponible dans Documentation/server/.

FIGURE 5.3 – Aperçu du diagramme de classes du Moteur de Jeu



5.5 Exemple de Bot

Il nous a fallu développer un exemple de bot à fournir avec le projet. Cet exemple nous a aussi permis de tester un déroulement de partie. Ne pouvant pas être réalisé en Java comme le reste du projet (réservé à un futur projet), nous avons décidé de le faire en C#.
Cet bot fonctionne de manière simple : il se connecte au serveur, envoie un message de login puis lance un thread écoutant constamment les messages du serveur, les analysant puis les traitant correctement. Après le message de login, il se charge donc uniquement de répondre au serveur, il ne décide jamais par lui même d'envoyer une information.
Lors d'une partie jouée sont stockés de manière permanente les murs étant apparus au moins une fois dans le champs de vision des fourmis du bot.
Les nourritures et fourmis ennemies ne sont quant à elles stockées en mémoire que pour les calculs du tour actuel, ces listes étant remises à zéro au début de chaque tour. Les fourmis personnelles du bot sont un peu spéciales : elles sont enregistrées afin de pouvoir les identifier selon leur position et prendre note de diverses information sur chacune d'elles, mais à chaque tour il est vérifié qu'elles sont toutes bien présentes dans les données envoyées par le serveur afin de les supprimer si elles sont mortes pour une raison quelconque.
Si une fourmi est identifiée comme porteuse de nourriture, le bot se charge de la faire revenir sur ses pas jusqu'à arriver à sa fourmilière. Si elle ne l'est pas, elle se déplace aléatoirement sur la carte avec une probabilité plus importante de continuer sur sa trajectoire, tout en faisant attention à ne pas rentrer dans un mur.

Mise en œuvre de composants

6.1 État actuel et fonctionnement

6.1.1 Interface client d'inscription

Le client d'inscription est une fenêtre assez simple permettant d'enter le nom d'un nouveau bot. Ce nom est vérifié : il doit commencer par une majuscule et comporter entre 3 et 16 caractères alphanumériques. Dans cette fenêtre, on doit aussi entrer l'adresse IP du Serveur de Jeu. Si le nom du bot est valide, celui-ci est envoyé en format JSON au Serveur de Jeu qui se charge de générer un token. Ce token est alors renvoyé et affiché dans la fenêtre du client. Une fonctionnalité permet aussi de copier ce token dans le presse-papier.

6.1.2 Création d'une partie

Dans l'état actuel, le matchmaking est fonctionnel. La classe qui l'implémente est *GameManager*.
Afin de rajouter un certain équilibre au niveau des matchs, nous sommes partis sur l'idée de faire jouer les joueurs ayant un niveau similaire entre eux. Il existe cependant une forte contrainte : le temps d'attente. Le principe est donc le suivant : on considère qu'un bot ne peut affronter qu'un bot dans son "champ de vision". Plus ce bot attend, plus ce "champ de vision" s'élargit, et donc plus il peut potentiellement affronter de bots.
Régulièrement, l'algorithme est lancé afin de trouver des listes de bots qui se trouvent chacun dans le champ de vision des autres. Si l'on trouve plusieurs listes possibles, on fait jouer la liste dont le temps d'attente cumulé est le plus long. Dès qu'une telle liste est trouvée, le match est automatiquement lancé et les bots sont retirés de la liste d'attente.

6.2 Tests

Afin de faire des tests sur nos différentes classes, nous avons utilisé JUnit. JUnit (cf Annexe B) est un simple framework qui permet d'écrire des tests. C'est une instance de l'architecture xUnit pour des frameworks de tests unitaires.
Nous avons donc rédigé des tests unitaires principalement sur la classe GameManager afin de vérifier les algorithmes de match making.
Nous avons également effectué des tests fonctionnels pour ce qui concerne le client d'inscription notamment. Pour réellement tester le serveur de jeu, on a créé un bot client. Cela permet de tester le protocole, soit l'ensemble des messages échangés entre client et serveur, les erreurs, etc.

6.3 Documentation

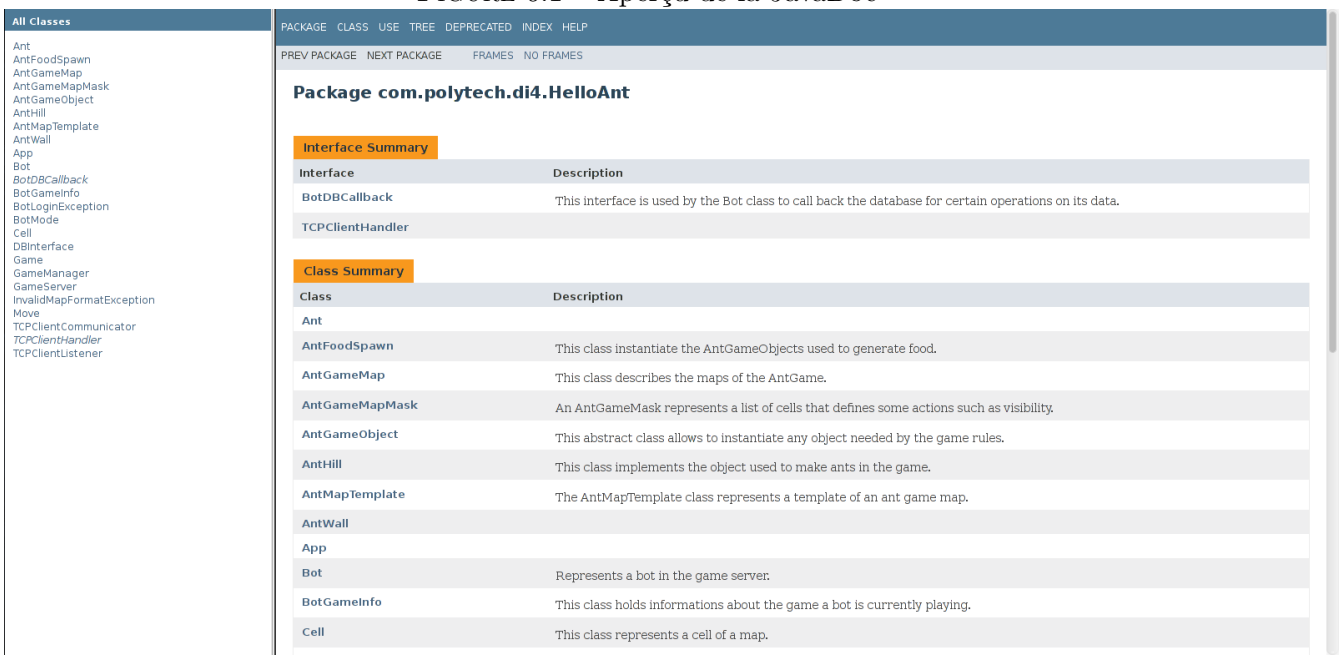
6.3.1 Javadoc

Afin de générer une documentation, nous avons choisi d'utiliser naturellement Javadoc puisque nous codons en Java.
La Javadoc (cf Annexe B) est un outil développé par Sun Microsystems (cf Annexe B) permettant de créer la documentation des API d'une application à partir des commentaires présents dans le code source. Ces commentaires se placent en en-tête des fichiers, des classes, des fonctions et des variables.

Tableau des tags de documentation Javadoc :

Tag Javadoc	Description
@author	Référence le développeur de la classe ou de la méthode
@version	Référence la version d'une classe ou d'une méthode
@see	Précise une référence utile à la compréhension
@param	Référence un paramètre de méthode. Requis pour chaque paramètre.
@return	Référence le type de valeur de retour. A ne pas utiliser pour une méthode sans retour.
@deprecated	Marque la méthode comme dépréciée (à éviter d'utiliser)
@exception / @throws	Référence aux exceptions susceptibles d'être levées par la méthode

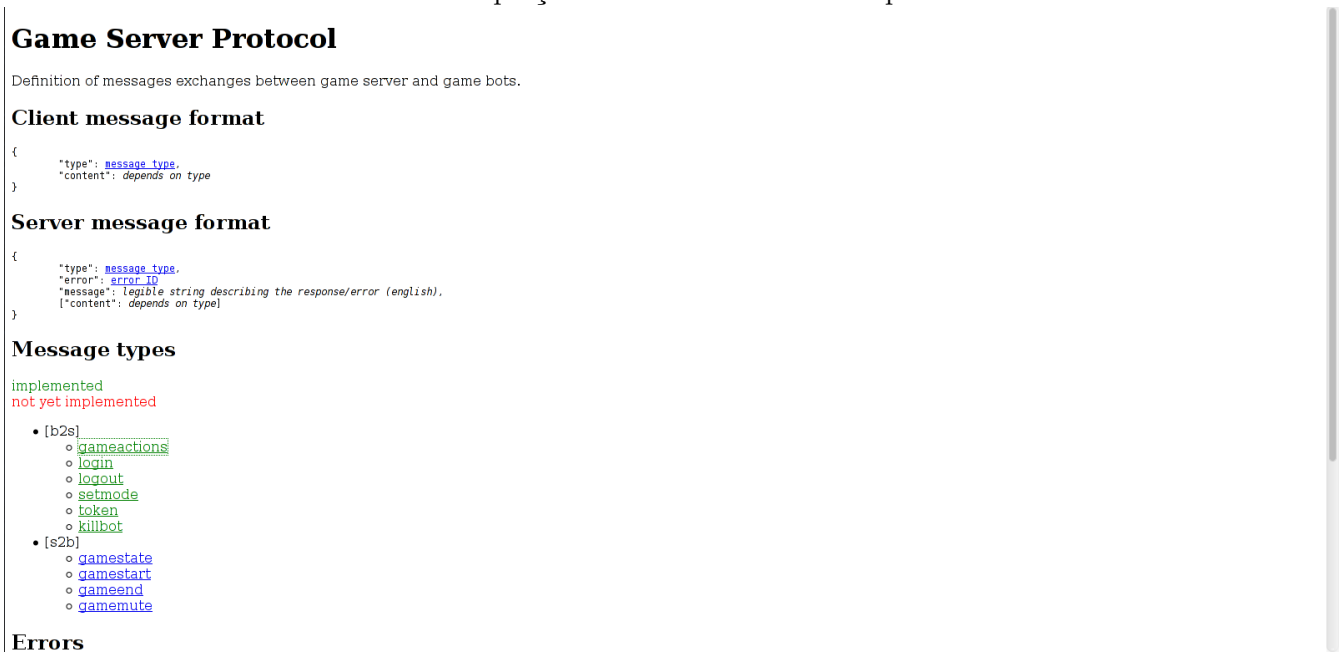
FIGURE 6.1 – Aperçu de la JavaDoc



6.3.2 Documentation du protocole

Nous avons également rédigé une documentation du protocole (figure 6.2, page 23), afin de permettre à l'utilisateur de comprendre au mieux ce qu'il peut se passer. Nous y décrivons les formats des différents échanges (requêtes autorisées et réponses possibles), les éventuels codes d'erreur... Pour accéder à ce fichier, il faut ouvrir `Documentation/protocol/index.html`

FIGURE 6.2 – Aperçu de la documentation du protocole



Game Server Protocol

Definition of messages exchanges between game server and game bots.

Client message format

```
{
        "type": message type,
        "content": depends on type
}
```

Server message format

```
{
        "type": message type,
        "error": error ID
        "message": legible string describing the response/error (english),
        ["content": depends on type]
}
```

Message types

implemented
not yet implemented

- [b2s]
  - gameactions
  - login
  - logout
  - setmode
  - token
  - killbot
- [s2b]
  - gamestate
  - gamestart
  - gameend
  - gamemute

Errors

6.3.3 Guide d'utilisateur

Ce document permet normalement à l'utilisateur de connaître précisément la démarche à suivre afin de créer un bot et commencer à jouer. Il décrit de plus les différentes règles du jeu. Par manque de temps, nous n'avons pas pu le rédiger. Néanmoins, l'utilisateur peut aller voir sur le site d'AIChallenge le guide d'utilisateur, puisque celui-ci correspond quasiment à celui que l'on aurait rédigé : ants.aichallenge.org.

Analyse critique

7.1 Avis d'Anthony

Ce projet était selon moi très intéressant, malgré un départ un peu difficile. Le sujet me plaisait beaucoup à l'origine mais la phase de spécifications a été un peu confuse et a entraîné une petite perte de motivation. Cependant, l'arrivée à une structure cohérente puis du code en lui même (permettant de tester différentes choses et de visualiser concrètement ce que nous avions et où nous allions) m'ont permis un regain d'intérêt. La période de fin fut assez chargée en débogage mais le fait d'avoir quelque chose de fonctionnel était assez gratifiant.
Ce projet était aussi pour moi l'occasion de découvrir le déroulement d'un projet à légèrement plus grande échelle que ceux en binôme que nous faisions jusqu'à présent, et nécessitant donc une plus grande coordination.

En somme ce projet était assez gros pour le temps qui nous était imparti mais je suis satisfait du modèle auquel nous sommes parvenus et de notre implémentation.

7.2 Avis de Benjamin

J'ai grandement apprécié travailler sur un projet de cette envergure. Surtout durant la phase de spécification le débat était ouvert. Quant aux remarques faites à propos de notre lenteur, je pense qu'elles étaient nécessaires afin de rompre l'inertie qui semblait s'installer. Au niveau de la répartition du travail, celle-ci a peut-être été déséquilibrée due à une différence de niveau de pratique. Cependant, chacun a participé au projet et j'espère que son aboutissement sera reconnu par nos évaluateurs. De plus je souhaiterais grandement qu'il soit poursuivi.

7.3 Avis de Jonathan

En ce qui me concerne, j'ai trouvé ce projet de groupe très instructif. On y voit rapidement certaines difficulté, notamment à travailler ensemble. Chacun a son point de vue, il n'est pas tout le temps aisé de défendre correctement son idée. La répartition équitable des tâches n'est pas non plus tout le temps facile. On se rend bien compte que garder tout le monde motivé, faire en sorte que tout se passe au mieux est un vrai métier. On remarque l'utilité et l'importance du Chef de Projet. On s'imagine un peu mieux comment ça peut être difficile de faire fonctionner les grands groupes internationaux.

A propos du développement en lui-même, ce fut un projet très intéressant. Il y a eu une bonne réflexion sur les diagrammes de classes, les cahiers de spécifications. J'ai bien aimé avoir à me questionner sur les problèmes de matchmaking, on se rend bien compte qu'il n'est pas tout le temps facile de faire quelque chose d'équitable pour tous les joueurs.

7.4 Avis de Juliette

Je suis très fière d'avoir était dans le groupe 3 de PIL. Toutes les personnes de ce groupe sont des gens qui se donnent pour un projet. Chacun a su s'impliquer et donner son avis quand cela a été nécessaire. Le sujet du projet ne m'intéressait pas tant que ça personnellement mais j'ai suivi mes camarades et je les remercie car j'ai appris beaucoup de choses. Je pense que c'est intéressant d'avoir cette vision de projet en "gros" groupes et non en binôme. Même si ça n'a pas été toujours évident. En effet, il n'est pas toujours simple de faire passer les infos à tout le monde ni d'être d'accord.
Ma conclusion : un bon projet et un bon groupe !

7.5 Avis de Minghui

J'ai été très intéressé par ce projet, c'est la première que je participais à un projet qui aura un utilisateur final. Ce projet était très grand, très complexe et nécessitait une vraie équipe pour le réaliser, et j'ai été honoré de travailler avec mes partenaires. Durant le projet, j'ai pris conscience que chacun des membres de l'équipe était talentueux, que ce soit au niveau des compétences de gestion de projet ou de la capacité à coder, de la coopération ou encore de la compréhension.
De mon point de vue, un chef de projet a émergé du groupe, et je l'ai trouvé très bon et consciencieux, et il nous a conduit à recenser et résoudre les problèmes. Il a pris beaucoup de son temps et a fourni beaucoup d'effort pour faire avancer le projet. Chaque individu a joué un rôle irremplaçable.

Encore une fois, durant ce projet, j'ai appris beaucoup de choses, et je remercie nos encadrants ainsi que toute l'équipe.

7.6 Avis de Nicolas

Pour moi, c'était une expérience très enrichissante : ce projet m'a permis de comprendre la difficulté et la puissance du travail en équipe. Je pense que c'est le lancement du projet qui est le plus compliqué, car on a tendance à partir dans tous les sens et c'est ce qui nous a fait défaut pendant les premières séances. De plus, il faut bien comprendre le besoin du client. Les conventions aussi, doivent être clairement spécifiée. Par exemple, nous avons réalisé, quelques semaines avant de tester le serveur final, que nous utilisions des couplets (col, row) pour les cellules au lieu de (row, col), comme spécifié dans les documents de AIChallenge... Il a fallu tout changer !

Il y a eu un "flop" en milieu de projet, on ne voyait plus le bout de la phase de spécifications et on avait beaucoup de mal à concevoir l'architecture du serveur de jeu. Nous nous sommes alors mis d'accord sur un modèle qui semblait stable, pour pouvoir livrer les spécifications. Mais une fois la phase de codage entamée, nous sommes vite revenus en arrière sur notre architecture, voyant que cela nous menait à un gouffre. Heureusement, tout est devenu plus clair et tandis que mes compétences en Java s'amélioraient, mon code devenait de meilleure qualité et mes idées étaient plus pertinentes.

Grâce à cette expérience, je pense pouvoir être plus efficace et plus prévoyant lors de mon prochain projet en groupe !

7.7 Avis d'Olivier

Le travail en équipe peut présenter d'énormes avantages. Chaque personne apporte son savoir et son experience à l'équipe. Cependant lors de groupes formés au hasard les compétences des membres se sont pas forcément cohérentes. De plus la gestion du groupe nécessite un travail supplémentaire d'encadrement. Si ce travail n'est pas réalisé les conséquences peuvent être problématiques pour l'avancée du projet. En effet le manque d'encadrement peut amener à une trop grande dispersion du travail ou une mauvaise répartition des tâches.
De plus, la communication étant un facteur clé du travail en groupe, il est important d'assurer la bonne transmission des avancées du projet. Cela permet à chacun de s'assurer de l'avancée de l'état global du projet et de pouvoir le comparer avec le travail qu'il a réalisé.
Même si officieusement, aucun chef réel n'était en charge du groupe, chacun a pu prendre un role décisionnel à un moment donné du projet. Néanmoins la mauvaise évaluation des tâches en début de projet a provoqué un déséquilibre dans la répartition du travail. Sur le plan de la communication, les outils utilisés durant le projet nous ont permis d'avoir une assez bonne représentation de l'état global.

7.8 Analyse générale

D'une manière générale, nous étions tous enthousiasmés à l'idée de travailler sur ce projet. Le sujet plaisait à la plupart d'entre nous. Au début nous avons été un peu hésitants quant à la direction à prendre. Nous nous sommes même rendus compte que nous avions mal compris ce qui était voulu par la partie cliente. Cependant, l'ensemble du groupe était à l'écoute et donnait son avis, et nous avons su retrouver la bonne voie.
Officiellement, aucun chef de projet n'était nommé. Néanmoins, vers le milieu du projet, nous étions un peu noyés sous les spécifications et nous ne voyions pas le bout du projet, il y a donc eu une chute de la motivation. A ce moment là, nous nous sommes même rendus compte que la direction que nous prenions n'était pas bonne. C'est là qu'un des membres du groupe a su prendre les choses en main, donner des directives afin que chacun reprenne goût et motivation dans ce travail collaboratif.

Au niveau des compétences individuelles, il nous a semblé clair qu'un groupe formé aléatoirement présente des disparités à ce niveau là. Certains ont parfois eu l'impression que le travail n'était pas réparti équitablement. Néanmoins chacun a su se rendre utile, mettre ses différentes compétences à disposition du groupe. En effet, savoir rédiger était aussi utile que savoir coder.

Ce projet est, on le pense, une bonne expérience. On a tous pris conscience des difficultés que le travail de groupe amène, malgré les avantages conséquents qu'il apporte. L'utilité d'un chef de projet se fait réellement sentir : savoir garder tout le monde motivé et organiser le groupe est un travail à temps plein. Et la vision que nous avons est probablement encore bien éloignée de la réalité des projets qui sont étalés sur des années.
Nous avons tous apprécié faire ce projet ensemble, découvrir les compétences et les manières de penser des autres. L'ambiance était très bonne, et nous avons tous un ressenti positif sur ce projet.

Chacun tient à remercier les autres membres du groupe, ainsi que les encadrants pour nous avoir permis de travailler ensemble sur ce projet.

Améliorations possibles

8.1 Client d'inscription

— Rendre l'interface de connexion plus attractive
— Ajouter une partie de tutoriel au client d'inscription expliquant comment communiquer avec le serveur, à quoi sert le token, que faire en cas de perte de celui-ci

8.2 Structure du jeu des fourmis

— Tester le jeu et son équilibre :
  — Lancer le système avec de vrais joueurs ayant élaborés leur propre bot
  — Noter les améliorations possibles que l'on peut apporter à l'équilibre du jeu (changer certaines règles, modifier certaines constantes, etc)
— Associer une fourmi à un masque afin de pouvoir avoir des masques différents selon les fourmis
— Menus plus complets, permettant de lancer plusieurs serveurs sur des ports différents, de configurer ceux-ci et de les fermer
— Améliorer l'intelligence artificielle du bot de training
— Ajouter un système de classement en fonction du score de chaque partie, à la manière de l'ELO

Conclusion

Ce projet était pour nous un véritable challenge car c'était une nouvelle expérience à la fois dans la nature du projet et la formation du groupe.

Le début du projet nous a demandé de réaliser beaucoup de choix sans vraiment pouvoir prévoir les conséquences qu'ils auraient. Nous avons donc organisé la suite selon nos propres prévisions. Cependant nous nous sommes rapidement aperçus qu'il existait de nombreux problèmes architecturaux auxquels nous n'avions pas pensé. Néanmoins une fois la réorganisation effectuée, nous avons pu nous concentrer sur l'avancement du projet et la partie programmation. Ainsi nous avons pu implémenter l'ensemble des fonctionnalités prévues au départ en ajoutant un visualiseur de partie.
Nous avons malheureusement manqué de temps pour réaliser certains tests dans le déroulement du jeu afin de procéder à certains réglages. Pour finir, nous sommes satisfaits du travail accompli et de pouvoir proposer un serveur fonctionnel à nos encadrants.

Ce projet nous a énormément appris et a été enrichissant à la fois humainement et professionnellement. Ce fut donc une belle expérience de nos années d'étude qui nous aura donné envie de renouveler l'expérience.

Glossaire

Ci-dessous se trouvent les définitions des différents termes utilisés durant le projet. Les mots sont sous la forme suivante : motEnAnglais (motEnFrançais) : definitionEnFrançais.

Action (Action) Décision prise par le bot concernant les déplacements de ses unités.
Bot (Bot) Programme sur la machine client qui joue à différentes parties.
Client (Client) Machine physique sur laquelle est lancé le bot.
Game (Jeu) Similaire à ce qu'on sous-entend dans la vie réelle (exemple : un jeu d'échec).

Game Engine (Moteur de Jeu) Implémentation des règles, permet de changer l'état de la partie.
Game Server (Serveur de Jeu) Programme réseau permettant l'interface entre les bots et le jeu.
Inscription Server (Serveur d'inscription Programme réseau qui permet au client de s'inscrire. Une application client a été créée afin de simplifier l'inscription.
Invalid movement (Mouvement invalide) Action du joueur considérée par le Moteur de Jeu comme interdite par les règles.
Map (Carte) Ensemble de cases servant d'environnement aux bots.
Match (Partie) Intervalle durant lequel différents bots s'affrontent.
Matchmaking (Création de partie) Processus permettant d'attribuer des bots et une map à un Match.
Match Score (Score de partie) Nombre de points qu'a accumulé un bot lors d'une partie d'un jeu.
Match State (Etat de partie) Les positions de toutes les unités sur la carte à un moment donné.
Match Turn (Tour de partie) Intervalle de temps pendant lequel chacun des bots a envoyé la liste de ses actions. Cela peut différer selon le type de jeu.
Player (Lecteur) Lecteur capable de lire un historique d'un match afin d'avoir un rendu visuel.
Replay (Historique) Sauvegarde de tous les mouvements et de toutes les positions des unités de chacun des bots au cours de la partie.
Token (Jeton) Identifiant unique associé à un seul bot. Chaque bot possède son propre jeton.

Liens

— AIChallenge
— Diagramme de classes initial
— Diagramme de classes du Serveur de Jeu
— Diagramme de classes du Moteur de Jeu
— Gantt (diagramme)
— GitHub
— Javadoc
— JSON
— JUnit
— Maven
— MySQL
— Trello
— SLF4J
— Sun Microsystems
— Vindinium

Développement d'un serveur de jeu pour l'initiation à la programmation

Département Informatique
4e année
2014 - 2015

Rapport de projet PIL

Résumé : Dans ce projet, nous devions créer un serveur de jeu capable de faire s'affronter des bots sur un jeu tour par tour. L'implémentation devait être telle que la reprise de notre code pour implémenter un autre jeu soit aisée. Nous y avons implémenté une version modifiée du jeu des fourmis d'AIChallenge.

Mots clefs : Serveur de Jeu, Intelligence Artificielle, Java, Bots, Communication réseau, AIChallenge, Fourmis

Abstract: In this project, we had to create a Game Server able to make bots match eachother in a turn-based game. The implementation had to be so any one who want to implement his own game can do it easily, without creating the wheel all over again. We implemented a modified version of the Ant Game on AIChallenge.

Keywords: Game Server, Artificial Intelligence, Java, Bots, Network communication, AIChallenge, Ants

---

Encadrants
Nicolas MONMARCHE
nicolas.monmarche@univ-tours.fr
Sébastien AUPETIT
aupetit@univ-tours.fr
Carl ESSWEIN
carl.esswein@univ-tours.fr

Université François-Rabelais, Tours

Etudiants
Olivier BUREAU
olivier.bureau@etu.univ-tours.fr
Anthony DEMUYLDER
anthony.demuylder@etu.univ-tours.fr
Nicolas GOUGEON
nicolas.gougeon@etu.univ-tours.fr
Jonathan LEBONZEC
jonathan.lebonzec@etu.univ-tours.fr
Benjamin MAUBAN
benjamin.mauban@etu.univ-tours.fr
Juliette RICARD
juliette.ricard@etu.univ-tours.fr
Minghui ZHANG
minghui.zhang@etu.univ-tours.fr

DI4 2014 - 2015