Analyse du projet : AcideFourmique

Team ANTivirusse

Josian Chevalier, Vladislav Fitc, Thi Ngoc Tam Nguyen

1er avril 2015

Table des matières

Partie 1

Introduction

L'équipe AcideFourmique avait deux contraintes :

— Écrire une partie du code en C

— Utiliser les agents/acteurs

Le langage principal utilisé est Scala. La bibliothèque Akka a été utilisée pour implémenter le système d'agent/acteur, et Lift-Json pour le parsing de Json.

Partie 2

Analyse du choix de technologie

L'équipe AcideFourmique a choisit de se baser sur le langage Scala pour son application, et l'approche se font donc via la programmation orientée objet. La nécessité de traiter les données représentant un monde justifient ce choix à nos yeux.

Cependant, l'utilisation de Scala permet également d'aborder certains problèmes de manière fonctionnelle, ce qui peut être intéressant pour traiter avec le langage des fourmis zombies.

De plus, les possibilités offertes par les compositions mixin peuvent offrir un réel avantage pour les définitions de stratégies.

L'utilisation de Akka semble un choix naturel en raison de leurs contraintes et de l'utilisation de Scala.

L'ajout de logs donne tout son intérêt à l'utilisation de C, mais les scripts des zombis auraient pu permettre de l'utiliser sans ajouter des fonctionnalités non nécessaires.

Partie 3

Analyse de la facilité de compréhension

3.1 Lisibilité du code

L'utilisation de Scala permet de rendre la lecture du code simple et fluide.

Le code est bien indenté, et les commentaires explicitent correctement tout ce qui n'est pas déjà évident.

Cependant s'il est facile à comprendre ce que fais le code de chaque fichier, la compréhension globale du projet n'est pas facile pour autant. Les relations entre les classes ne sont pas toujours évidentes, et l'utilité des classes n'est parfois pas très clair à l'aide du code. Par exemple, il est difficile de comprendre l'intérêt de la classe IA en lisant le code, et elle semble n'exister que pour satisfaire la contrainte.

Nous avons un module similaire, cependant il hérite du rôle de décision et effectue toute l'analyse de la partie en cours pour décider de ses coups, ce qui indique clairement son rôle au travers de son code.

3.2 Compréhension par le rapport

Cependant une lecture de leur rapport permet de comprendre le rôle qu'ils souhaitaient lui donner, et les idées de stratégies sont inventives et pourraient être efficace. Mais ce rôle n'a pas été implémenté, et un squelette de cette classe ou des appels de fonctions aurait permit de le comprendre directement dans le code, même sans l'implémenter.

L'absence de liens entre les modules dans le diagramme de leur rapport ne permet pas de comprendre la logique globale de fonctionnement, là ou le code ne simplifie pas la tâche.

3.3 Remarques générales

Les noms des classes et des objets ne permettent pas toujours de bien saisir le rôle des différents éléments. Par exemple l'objet Antroid interagit avec le serveur pour tout ce qui concerne la création de parties et le système d'authentification mais ce nom ne semble pas adapté et nous permet pas de deviner ce que l'on trouvera à l'intérieur de cette classe.

Nous avons un module similaire, mais nous lui avons donné le nom Player, pour indiquer qu'il s'agit du joueur qui interagit avec le serveur.

Les fonctions sont généralement bien nommées, ce qui aide à la compréhension locale.

Partie 4

Analyse des choix de conception

4.1 Liens logiques entre les modules

On regrettera principalement le manque de cohérence dans les interactions entre les différents modules, donc la logique de fonctionnement reste obscure.

De plus, les commandes ne passent pas par le modèle (le module game), qui ne sert qu'à stocker les données du serveur sans offrir d'outils pour les manipuler plus simplement. Il s'agit donc d'un objet très primitif.

La classe Game ne semble être jamais utilisée dans ce projet, ce qui jour laisse perplexe quand à son rôle de représentation de donnée.

Nous pensons qu'un découpage de la problématique en couche aurait été plus adapté. Il existe bien ici une couche réseau, mais sa logique reste plutôt transparente vu de l'extérieur.

Nous avons préféré opté pour une approche par couche dans notre projet, ou nous avons donné un rôle central au modèle, qui permet de manipuler toutes les données sous injonction de la couche de décision, tout en abstrayant totalement l'utilisation du serveur dans la couche réseau.

4.2 Couplage et Interfaçage

Par exemple, la création d'une partie dans Main indique que la classe Antroid ne cache pas suffisamment sa logique de fonctionnement, et il en résulte un couplage trop fort, puisqu'on est obligé d'effectuer tous les traitements successifs dans la classe là ou une seule fonction aurait suffit en relayant tout cela à l'intérieur de la classe Antroid.

Nous avons une approche de la création de partie similaire, cependant nous avons fait le choix de cacher tous les traitements à l'intérieur de la classe qui s'en occupe.

Le couplage avec la couche de réseau est trop fort à nos yeux, un changement de spécification de l'API entraînerait de nombreux changements dans le code en raison de l'absence d'abstraction des commandes de fourmis au sein de la couche réseau.

Notre approche consiste plutôt à faire appeler par le modèle les actions de fourmis qu'il veut effectuer dans la couche réseau, ce qui permet d'y centraliser les envoie de messages liés aux actions.

4.3 Correction et robustesse

Le makefile avec sbt est bien organisé, et le projet compile et marche sans soucis. L'exemple d'implémentation fonctionne bien.

Il est difficile d'évaluer sa robustesse par des tests, mais la simplicité de ce projet suggère une bonne robustesse.