Architecture logicielle
Rapport de projet

Clément Badiola Samuel Da Silva

Chargé de TD : David Auber

21 novembre 2012

Introduction

Ce document est le rapport d'un projet d'architecture logicielle effectué par un binôme d'étudiants dans le cadre de leur deuxième année de Master.

Le projet concerne la modélisation des données d'un jeu de type *RTS* (Real-Time Strategy). Il s'agissait de développer des fonctionnalités en utilisant nos connaissances des patterns et des bonnes pratiques en terme de génie logiciel.

Mots-clés : architecture logiciel, pattern, conception, refactoring

Table des matières

Chapitre 1

Cahier des charges

Sommaire

Comme nous venons de le voir, notre projet consiste à nous intéresser à la modélisation des données associées à un jeu de type *RTS* . Il s'agit donc de définir pour ce programme les objectifs aussi bien du point de vue non-fonctionnel que du point de vue des tests. Voyons tout d'abord les besoins non-fonctionnels.

1.1 Besoins non-fonctionnels

1.1.1 Refactoring

Le refactoring consiste à retravailler un code source dans le but d'améliorer sa lisibilité et son efficacité, et de simplifier sa maintenance. L'introduction de nouveaux patterns induit le besoin de refactoriser souvent le code afin de simplifier le codage et la compréhension. En plus d'un simple nettoyage, cela nous amène à vérifier que notre architecture répond toujours aux objectifs fixés.

L'objectif est bien sûr d'obtenir un gain de clarté, de lisibilité, de maintenabilité, et probablement de performances. Nous pouvons ainsi continuer l'ajout de fonctions sur une base saine.

1.1.2 Transparence pour le client

Nous pensons qu'un des objectifs du génie logiciel est de tenter d'obtenir une architecture telle que l'apport de nouvelles fonctionnalités se fasse de manière transparente pour le client. Nous pourrions donc ajouter des contraintes ou des fonctions sans que le *main* ou les tests unitaires soient dégradés.

1.1.3 Flexibilité

La flexibilité d'une architecture est une pierre angulaire pour la maintenabilité d'une application. Dans cette optique, nous devons garder notre code le plus simple possible, mais également penser aux contraintes induites par l'utilisation des patterns.

1.2 Tests prévisionnels

Afin de garantir la fiabilité de notre livrable, nous réaliserons une série de tests unitaires et de tests fonctionnels. Les tests unitaires nous permettrons de nous assurer du bon fonctionnement de certaines parties déterminées du logiciel.

1.2.1 Tests de la génération de combattants

Nous réalisons un ensemble de génération de soldats tout en les équipant de pièces d'armement puis nous vérifions la cohérence de leur force de frappe et de leur réaction aux attaques adversaires.

1.2.2 Tests de la génération de groupes armés

Nous générons un ensembles de groupes composés de soldats puis nous vérifions que les groupes sont bien cohérents par rapport aux combattants assignés.

1.2.3 Tests des combats

Nous faisons combattre des combattants et des groupes pour vérifier la cohérence de leurs réactions aux actions d'attaque et de défense.

1.2.4 Tests des contraintes d'équipement

Les combattants et groupes armés sont équipés d'une arme ou d'un bouclier puis ré-équipés d'un même type d'armement pour vérifier qu'ils ne possèdent pas ensuite deux fois le même type d'équipement.

1.2.5 Tests des observateurs

Les observateurs doivent fournir un résultat prédéterminé pour des groupes armés ou des combattants sur un scénario fixé. Nous vérifions que les résultats correspondent au scénario.

1.2.6 Tests des fabriques

Les fabriques doivent permettre de générer des combattants d'une époque particulière et d'équiper ces combattants avec l'armement de l'époque. Nous testons que les combattants soient bien ceux attendus pour les époques du moyen-age et du futur lointain. Nous testons également que leur équipement soit bien conforme à l'époque. Par exemple, un Space marine ne sera pas équipé d'une épée en bois, mais d'un laser.

Chapitre 2

Architecture

Sommaire

L'architecture constitue un des éléments les plus importants pour le développement d'une bonne application. Voici les concepts sur lesquels nous nous appuyons pour réaliser une architecture cohérente :

- Garder une architecture la plus simple possible. Chaque classe représente quelque chose de précis.
- Ranger les méthodes dans les bonnes classes.
- Limiter les attributs des classes afin de limiter les bugs.
- Limiter les dépendances externes et au langage. Les langages évoluent vite et il peut être parfois intéressant de pouvoir passer une application sur un autre langage.

2.1 Diagramme des cas d'utilisation

Nous avons réalisé un schéma des cas d'utilisations, visible en figure 2.1 page suivante afin de synthétiser les besoins de l'utilisateur, pour offrir une vision simplifiée du système. Nous rappellerons brièvement les fonctionnalités par la suite.

Générer des combattants

Le client peut générer un combattant d'une classe spécifique. Il génère un combattant en utilisant un des types disponibles. Il doit spécifier l'armement du combattant après l'avoir généré.

Simuler des échanges de coups

Si des combattants ont été auparavant générés, ils peuvent s'échanger des coups, qu'ils soient équipés ou non. Un combattant pourra frapper une cible ou parer un coups.

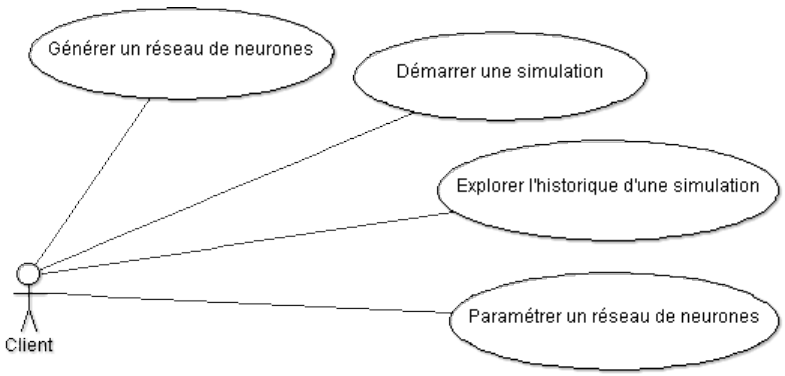

FIGURE 2.1 – Diagramme de cas d'utilisation

Créer des armées

Les armées sont composées d'éléments qui peuvent être des groupes de combattants ou des combattants. Les armées peuvent frapper une cible ou parer une attaque.

Afficher les informations d'un groupe armé

Le client peut afficher les combattants d'un groupe armé ou compter les effectifs d'un groupe armé.

Afficher les combattants morts et les groupes décimés

Le client peut être averti du décès d'un combattant ou de l'éradication d'un groupe armé. Il peut prévenir les amis d'un combattant du décès de celui-ci. Il peut aussi être averti du nombre de morts au fur et à mesure de l'évolution des combats.

Créer des familles de combattants par époques historiques

Le client peut créer des combattants spécifiques à une époque et les équiper avec le type d'armement conforme à cette époque. Par exemple, un guerrier Cromagnon ne se verra pas équipé d'une armure Terminator.

2.2 Les classes

Le diagramme global des classes de l'architecture a été réalisé à l'aide du logiciel *Architexa* intégré à *Eclipse*. Nous ne parlerons que de notre architecture finale, obtenue au fur et à mesure du refactoring.

Étant donné l'étendue de l'application, nous ne pouvons pas présenter de diagramme UML à la fois complet et lisible dans ce rapport. Voici le diagramme des classes sans les méthodes, en figure 2.2 page suivante.

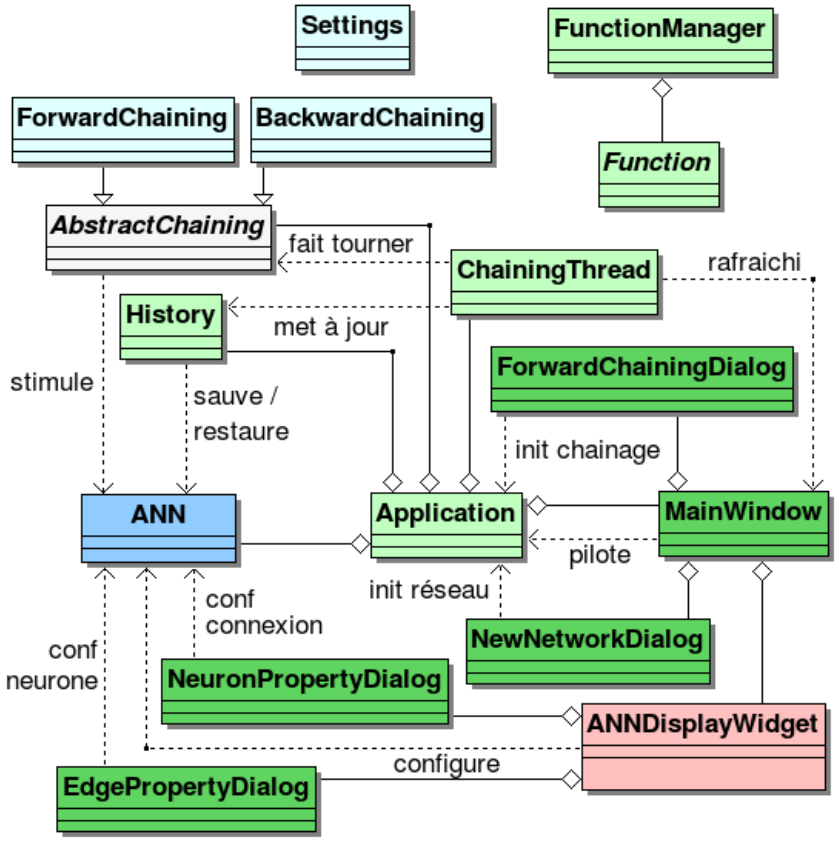

FIGURE 2.2 – Diagramme de classes de l'application

Chapitre 3

Travail

Chapitre 4

Bilan

Sommaire

4.1 Bilan

En conclusion, nous pouvons dire que le bilan général est plutôt bon. Nous avons globalement satisfait les besoins client tout en réalisant nos objectifs initiaux en terme de refactoring et de qualité.

Au niveau de l'architecture, les méthodes sont restées simples, ce qui limite selon nous les sources de bugs. Nous limitons les dépendances avec les bibliothèques Java. L'application est donc plus maintenable et évolutive.

4.1.1 Échecs

Malgré ce bilan positif, nous déplorons le manque de temps pour mélanger l'utilisation intelligente des patterns et des principes de généricité et d'introspection, qui forment une combinaison pouvant répondre à certains problèmes liés à l'introduction de patterns.

4.2 Si c'était à refaire...

Si c'était à refaire, nous aborderions le problème différemment. En incluant nos connaissances en gestion de projets, nous pourrions aborder les problèmes liés à la complexification des phases d'intégration. En connaissant les patterns aux sein d'une équipe, nous aurions pu mesurer les effets de leur utilisation dans le cadre d'un projet utilisant les méthodes agiles, comme l'*XP* par exemple.

Table des figures

Rapport de projet d'architecture logicielle.
Wargame data modelizing.
Master 2, 2012

Les sources de ce rapport sont disponibles librement sur https://github.com/mr-nours/rapport-al.