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* . Définissons donc pour ce programme les objectifs du point de vue non-fonctionnel.

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.

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 4 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.

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 5. En rouge la partie développeurs, en bleu le cœur de l'application (côté client), et en vert les classes plus utilitaires.

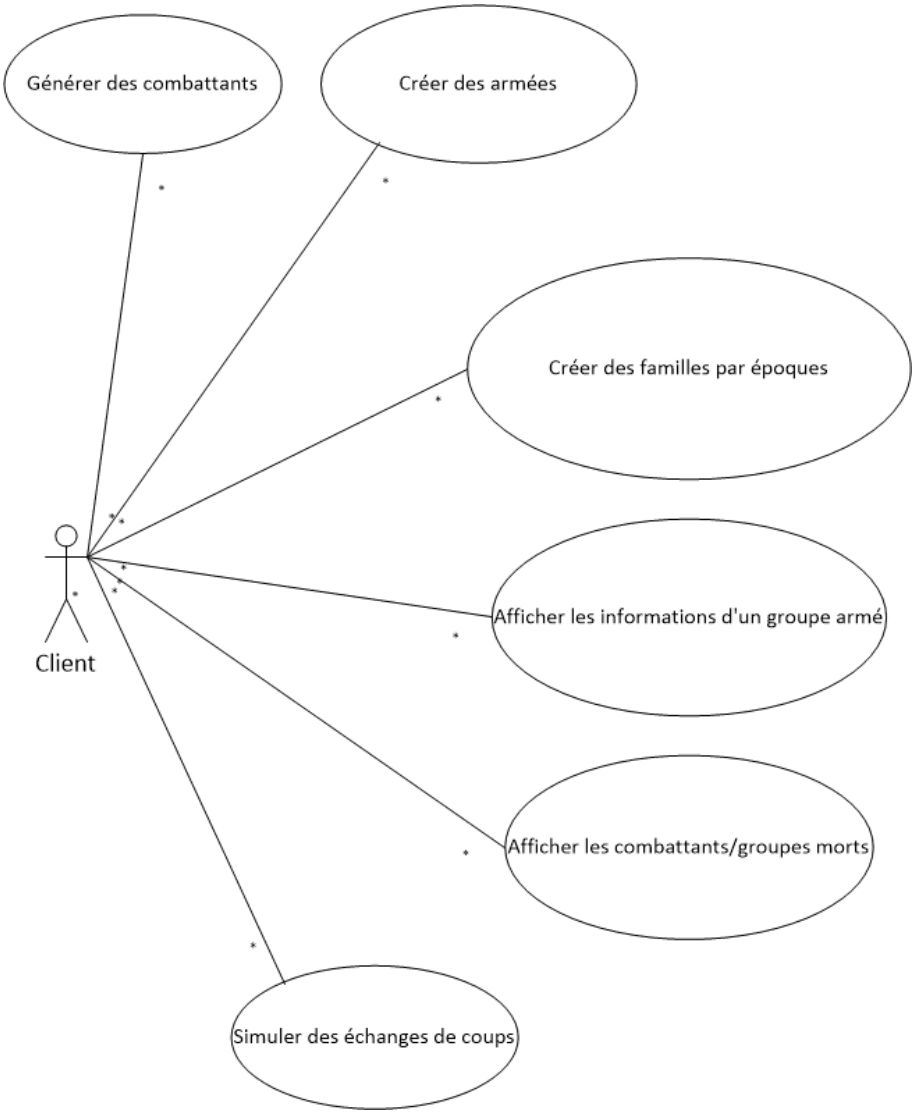

FIGURE 2.1 – Diagramme de cas d'utilisation

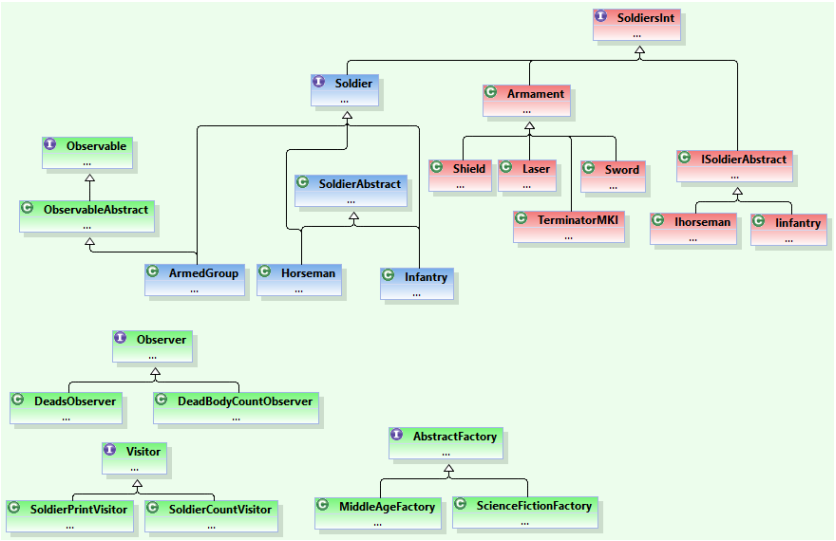

FIGURE 2.2 – Diagramme de classes de l'application

Chapitre 3

Travail

Sommaire

3.1 Travail

3.1.1 Décorateur

La première étape de conception du projet consistait à permettre à des soldats de disposer d'armements (un bouclier et une épée). Deux catégories de soldats devaient exister, les fantassins et les cavaliers, les soldats disposant d'un certain nombre de point de vie.

Pour réaliser cette première étape, nous avons commencé par créer les deux classes de soldats ainsi que deux classes d'armes. Afin d'anticiper le fait de pouvoir rajouter plus de catégories de soldat et plus d'armes, nous avons créé deux classes abstraites (Armament et SoldierAbstract).

L'objectif était alors de pouvoir rajouter une ou plusieurs armes sur un soldat existant, et cela de façon dynamique, c'est à dire sans avoir à recréer le soldat.

Pour répondre à cette problématique, nous avons utilisé le pattern Décorateur. Nous avons donc créé une classe SoldierInt représentant l'interface d'un

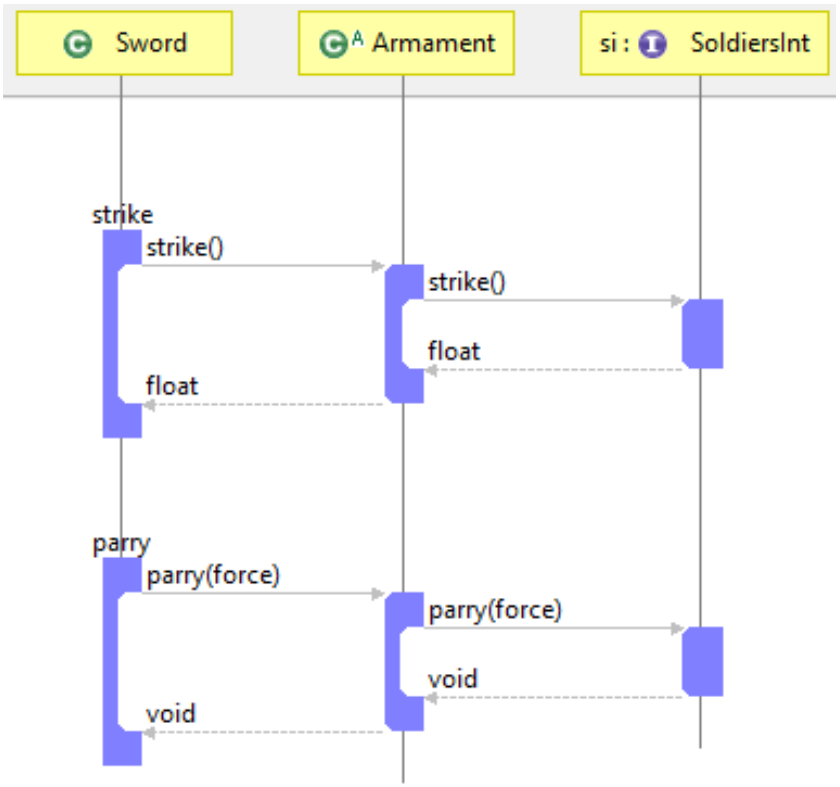

FIGURE 3.1 – Diagramme de séquence du pattern décorateur

soldat. Nous avons ensuite fait hériter notre classe SoldierAbstract de SoldierInt, et nous avons fait hérité notre classe Armament de SoldierInt, classe devenant notre décorateur puisque nous avons rajouté dans cette classe une référence à SoldierInt nous permettant ainsi de décorer notre soldat.

Ainsi donc, nous pouvons maintenant décorer un soldat avec des armes.

Une autre architecture possible aurait été de séparer les armes des soldats , ceci en créant un décorateur SoldierArmed avec comme sous classe SoldierWithShield et SoldierWithSword, ceci permettrait de séparer le comportement des armes du comportement des soldats, permettant ainsi de définir des méthodes spécifiques aux armes, sans en affecter les soldats, et vis vers ça. Cette solution a pour inconvénient de devoir rajouter plusieurs classes.

Une fois notre première architecture créé, nous avons rajouté la possibilité pour un soldat de porter des coups à ses adversaires (et à contrario de les parer) avec une vivacité dépendant de l'armement et de la catégorie du solat.

3.1.2 Procurateur

Afin de contrôler les actions demandées par le client sur le soldat (comme par exemple l'ajout d'une nouvelle arme), nous avons appliqué le pattern Procuration nous permettant ainsi de fournir au client une représentation de l'objet

soldat qui aura pour rôle ici de contrôler lors de l'ajout d'une arme sur un soldat si cette arme est déjà présente ou non sur le soldat.

Nous avons donc créé une interface Soldier contenant deux méthodes permettant d'ajouter des armes (addShield et addSword),interface héritant de notre classe SoldierInt. Les classes Infantry et Horsman ont quand à elle été dupliqués et implémentent l'interface Soldier (modifié par la suite car une classe abstraite implémente maintenant l'interface soldat afin d'éviter la duplication de code).

Enfin, nous avons rajouté une durée de vie à nos armes avec la variable RESISTANCE que nous avons placé dans nos classes d'armes (Shield et Sword). La résistance de l'arme diminue de 1point à chaque utilisation (c'est à dire à chaque appel de strike() ou de parry()).

3.1.3 Composite

Le seconde étape dans le projet consistait à rajouter la possibilité pour le client (et cela sans qu'il est besoin de s'en préoccuper) de créer des groupes armé, un groupe armé de soldats pouvant être composé de plusieurs groupes armés, eux-mêmes décomposés en sous-groupes armés, et ainsi de suite. Cette notion de groupe armé fait ici apparaître une structure d'arbre avec des éléments composé d'autres éléments, structure faisant apparaître une certaine hiérarchie. C'est donc pour cela que nous avons choisi de mettre en place le pattern Composite en créant une classe ArmedGroup contenant une référence de Soldier (qui est notre proxy, ce que manipule le client).

3.1.4 Visiteur

Comme nous avions maintenant à la fois des groupes armé de soldats ainsi que des soldats de différente catégories, afin de faciliter l'ajout de nouvelles fonctionnalités à la fois sur les groupes armés et les soldats eux même, nous avons mis en place le pattern Visiteur. Sans ce pattern, à chaque ajout de fonctionnalité, nous étions obligé de rajouter les méthodes dans nos classes Horsman, Infantry et ArmedGroup, surtout si nous voulions faire des traitement différent en fonction de la classe. Grâce au pattern Visiteur, l'ajout de fonctionnalité supplémentaire (comme par exemple afficher tous les soldats formant un groupe armé ou encore compter des effectifs de soldats par rapport à leur type au sein d'un groupe armé) se fait maintenant dans une seule classe, sans avoir à toucher au reste du code. Nous avons donc par exemple créé une classe SoldierPrintVisitor pour ajouter la fonction d'affichage des soldats ainsi que la classe SoldierCountVisitor pour ajouter la fonction de comptabilisation, et tout ceci de manière totalement transparente vis à vis du reste du code, c'est là tout l'intérêt du pattern Visiteur, pas de modification à faire sur les autres classes de l'application et possibilité de faire des traitements spécifique pour chacune des classes de l'application. En résumé, nous pouvons maintenant ajouter des fonctionnalités sur les objets de notre choix sans toucher au reste du code.

3.1.5 Observateur

Une des autres problématiques de ce projet était de pouvoir suivre le déroulement des conflits au fur et à mesure de l'action. Une possibilité aurait été

de faire de nombreux print dans nos fonctions, ce qui aurait produit des affichages au fur et à mesure du déroulement de la bataille, c'est à dire à chaque appel de parry() ou de strike(). Cependant, cette méthode n'est vraiment pas très pratique et pas du tout maintenable ! C'est donc ici que le pattern Observateur va être intéressant, en effet, il va nous permettre d'observer un objet et dès qu'il change, on va pouvoir être notifié de ce changement afin d'appliquer les traitements adéquat suite à ce changement d'état. Ainsi donc, pour afficher par exemple au fur et à mesure les noms des soldats morts au sein d'un groupe armé ainsi que les armés détruites, nous avons mit en place une interface Observer avec pour sous classe DeadsObserver, classe qui se chargera de réaliser l'affichage. Nous avons également rendu observable notre classe ArmedGroup en créant l'interface Observable. ObservableAbstract implémente cette interface et contient une liste d'Observer, une liste qui permettra donc à l'objet lorsqu'il change d'en informer tout ses observateurs. Cette classe est également intéressante ici car elle nous permet de définir une interdépendance de type un à plusieurs, de tel façon que si on souhaite envoyer des télégrammes d'excuse à une liste d'ami une fois qu'on est mort (même si cela n'est pas envisageable dans la vrai vie), grâce au pattern, on pourra notifier tout les objets qui dépende du mort (donc de l'objet qui est observé par sa liste d'amis). L'inconvénient et la limite ici du pattern est que le graphe peut vite devenir complexe. En effet, si l'on veut mettre en place le système des envoi de télégrammes d'excuse à ses amis, c'est à dire les soldats d'un même groupe par exemple, il va falloir que chaque objet soit observé par tout les autres objets « amis », ce qui va provoquer la création de plein d'objet de type Observable provoquant alors un ralentissement notable du programme ! Au vue de ces dernières remarques, nous n'avons pas retenu la fonction des télégramme via ce pattern là.

3.1.6 Singleton

Enfin, afin qu'il ne puisse y avoir qu'une seule instance de chaque observateur, nous avons mis en place le pattern Singleton, cela en donnant accès à l'instance de l'observateur via la méthode getInstance(), méthode qui renvoit la variable Instance (déclaré en statique). Le constructeur est quand à lui rendu inaccessible.

3.1.7 Fabrique abstraite

3.2 Tests

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

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

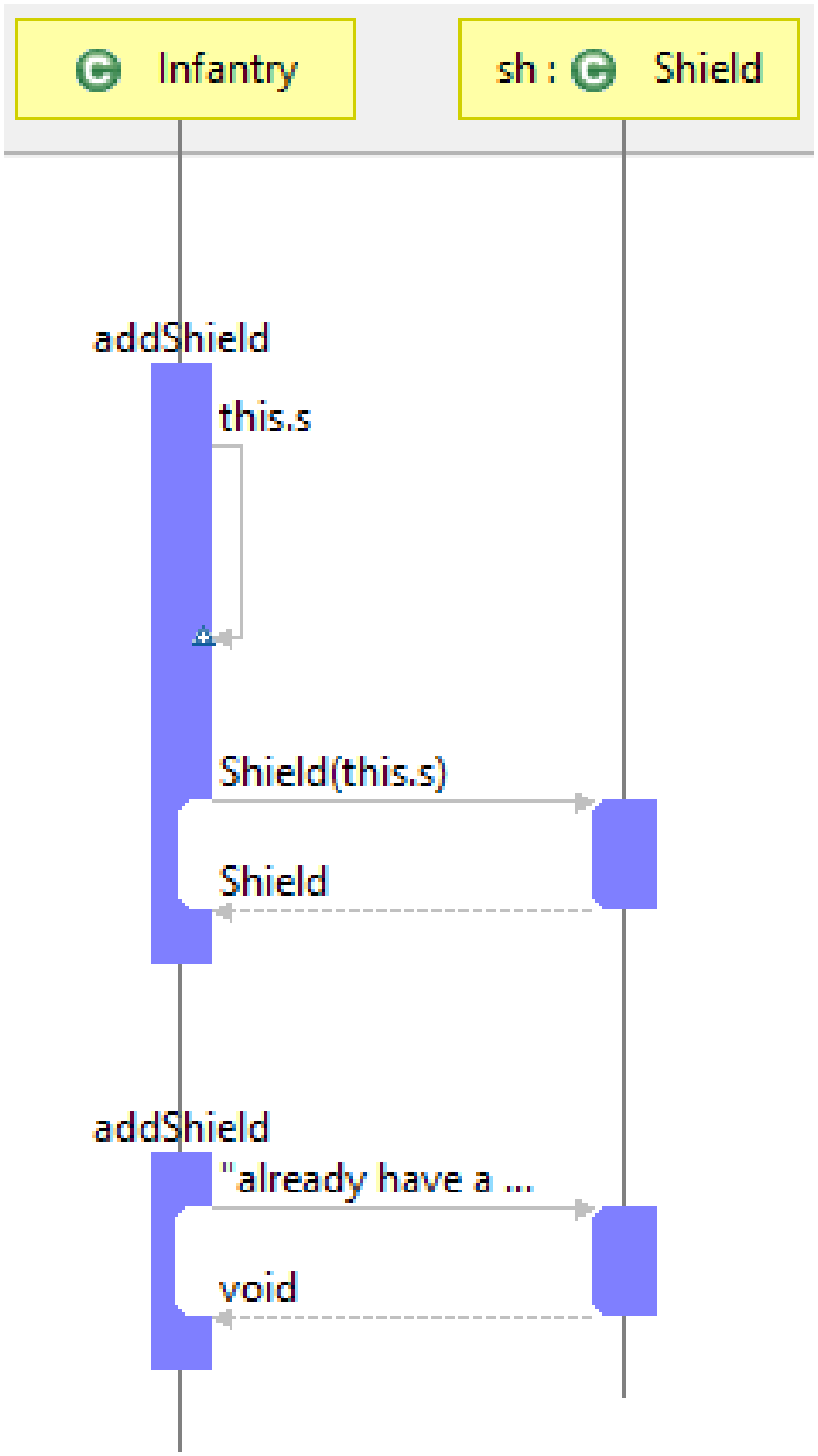

FIGURE 3.2 – Diagramme de séquence du pattern procurateur

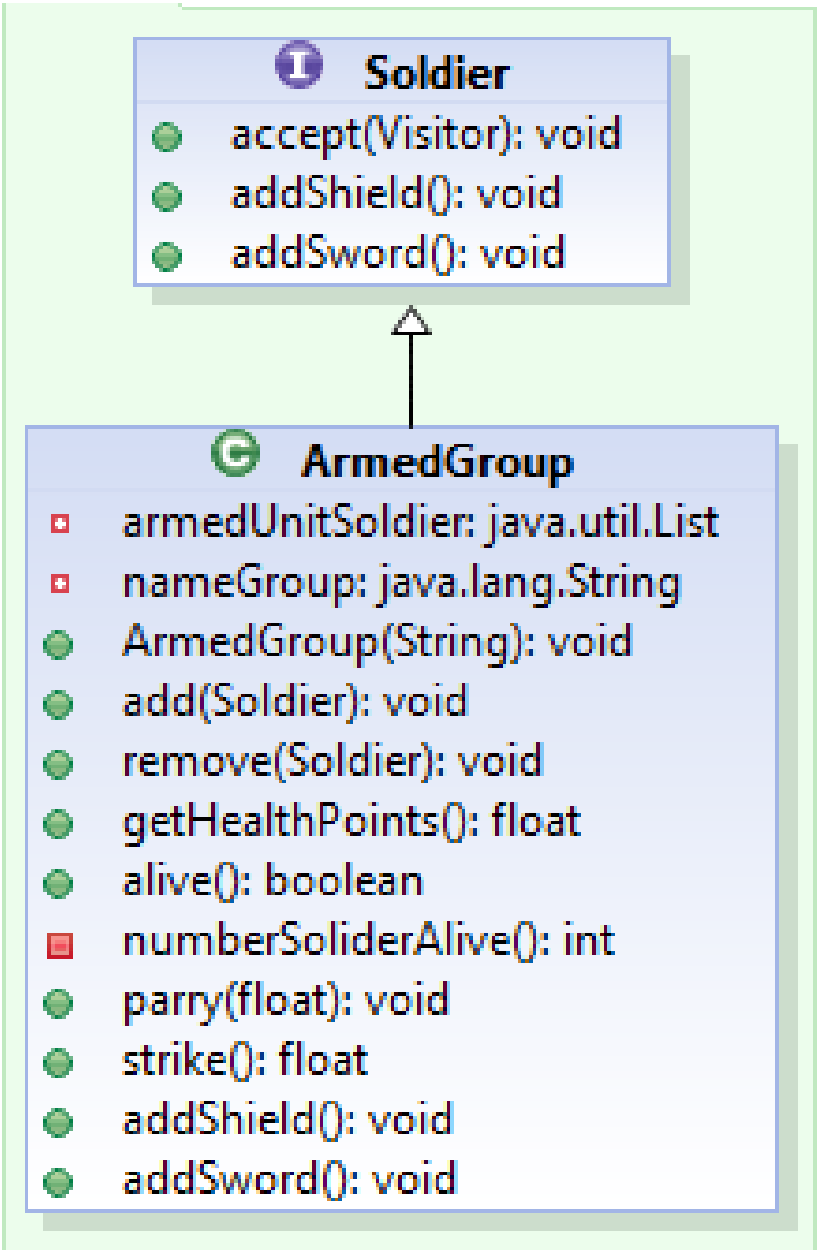

FIGURE 3.3 – Diagramme de classes du pattern composite

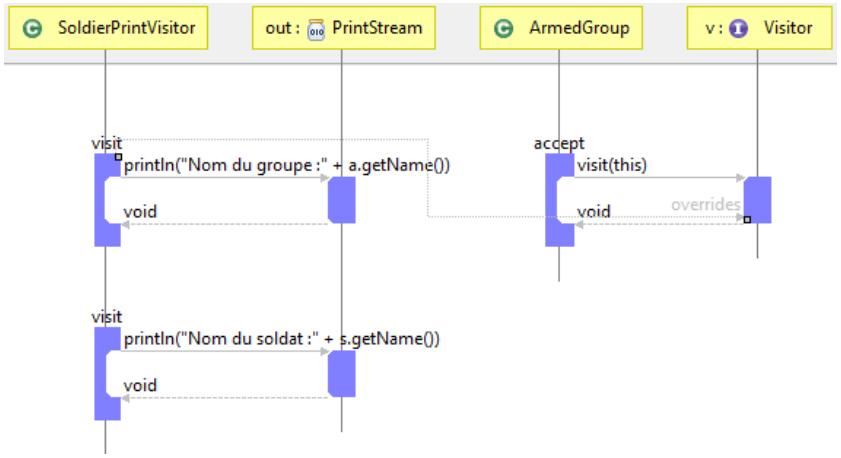

FIGURE 3.4 – Diagramme de séquence du pattern visiteur

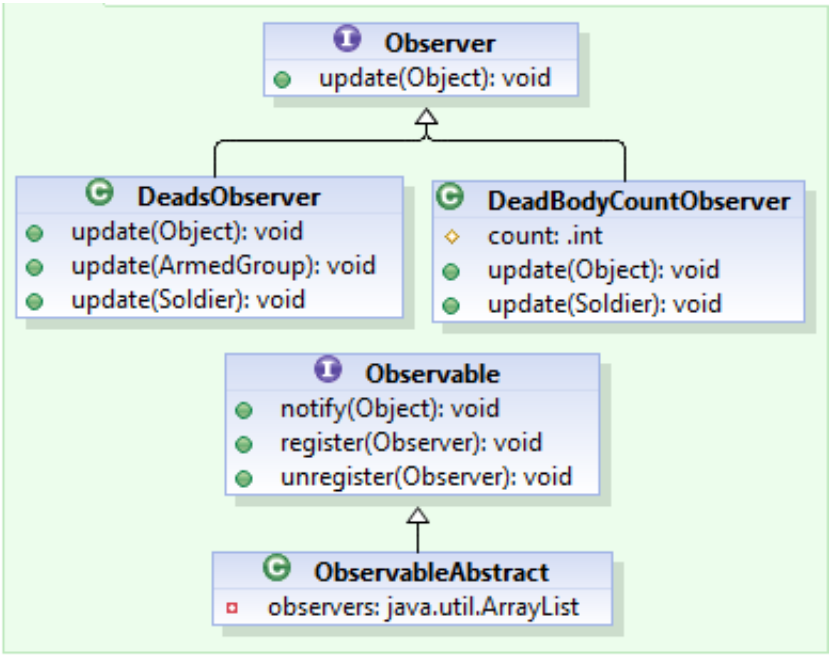

FIGURE 3.5 – Diagramme de classes du pattern observeur

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.