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

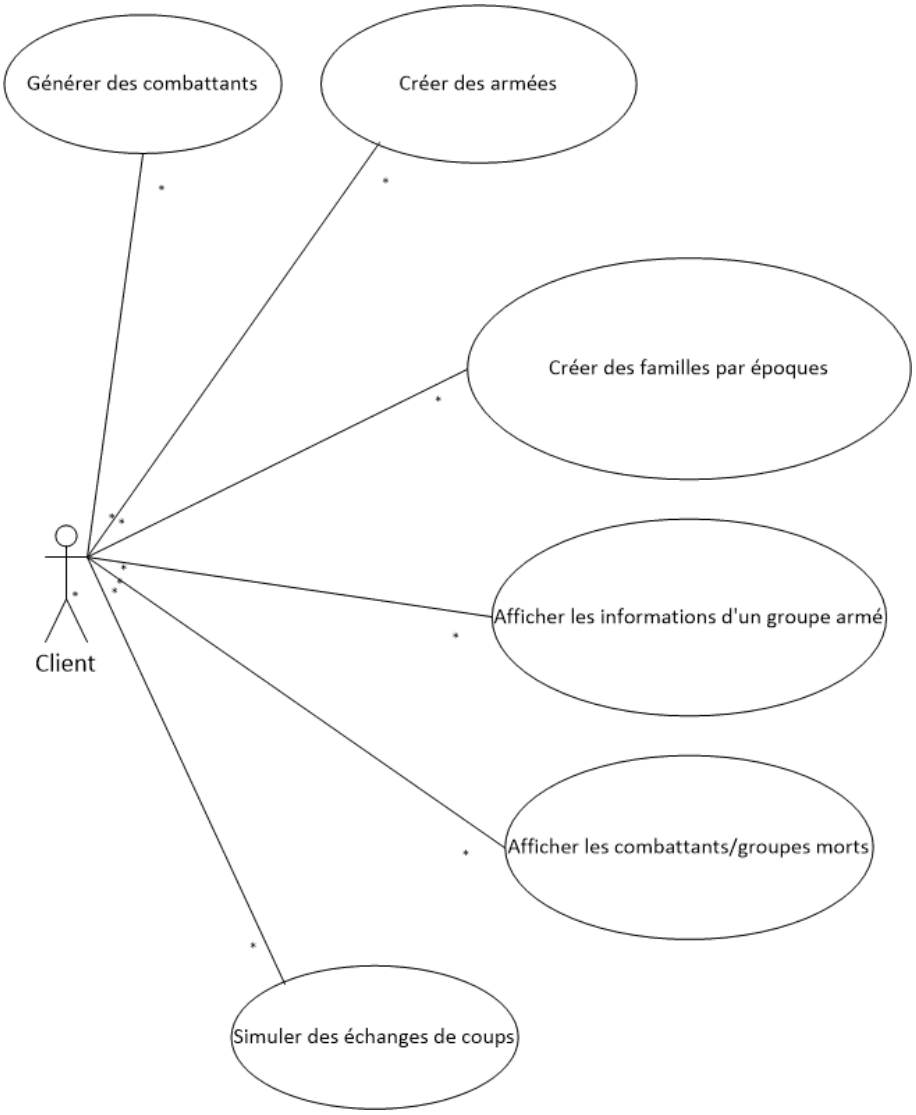

FIGURE 2.1 – Diagramme de cas d'utilisation

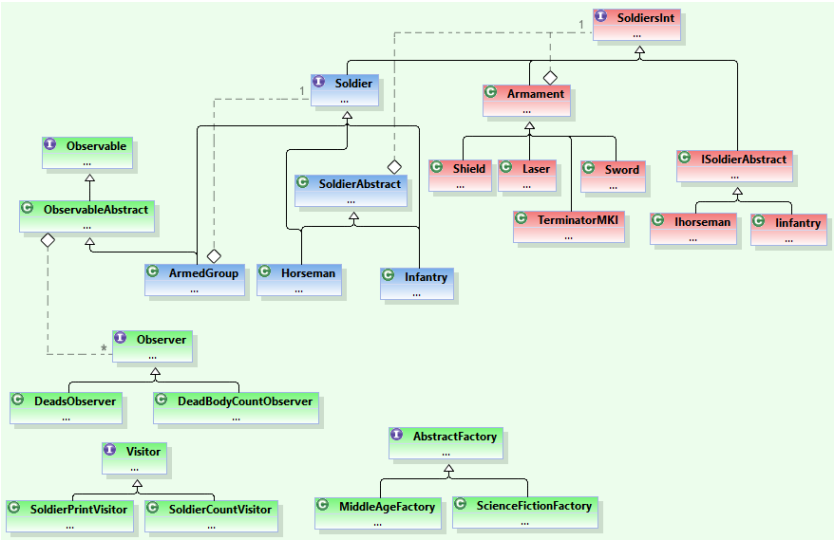

FIGURE 2.2 – Diagramme de classes de l'application

Chapitre 3

Travail

Sommaire

3.1 Travail

3.1.1 Décorateur

La première étape de conception du projet consistait en l'implémentation de soldats disposant d'armements (un bouclier et une épée par exemple). Nous disposions de deux catégories de soldats : les fantassins et les cavaliers. Les soldats possédaient un certain nombre de points 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 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 interface *SoldierInt* associée aux soldats. Nous

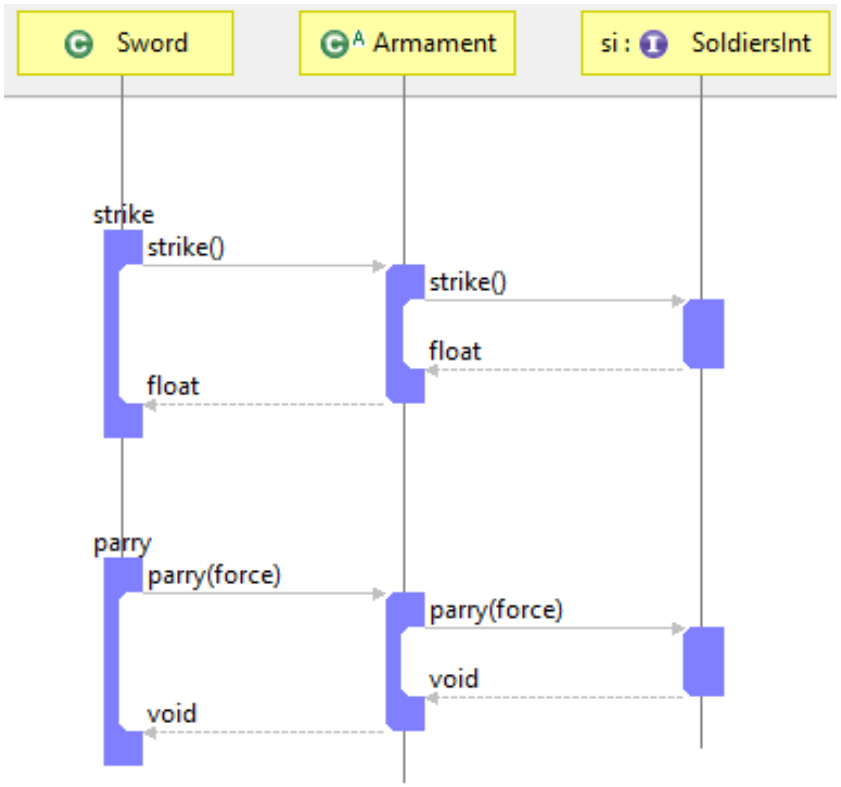

FIGURE 3.1 – Diagramme de séquence du pattern décorateur

avons ensuite fait implémenter notre interface *SoldierInt* par nos classes *SoldierAbstract* et *Armament*. *Armament* devenant notre décorateur puisque nous avons rajouté dans cette classe une délégation sur un *SoldierInt*, ce qui permet de décorer notre soldat, voir la figure 3.1.

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, pour ainsi définir des méthodes spécifiques aux armes sans affecter les soldats, et inversement. Cette solution a pour inconvénient de nécessiter l'ajout d'un nombre de classes croissant avec le nombre d'armements implémentés.

Une fois notre première architecture créée, 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 soldat.

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 *Pro-*

curation, voir la figure 3.2 page 13, qui fournit au client une représentation "augmentée" de l'objet soldat, et qui aura pour rôle de contrôler, lors de l'ajout d'une arme sur un soldat, si le soldat possède déjà cette arme ou non.

Nous avons donc créé une interface *Soldier* contenant deux méthodes permettant d'ajouter des armes : *addShield* et *addSword*. Cette interface étend l'interface plus générale *SoldierInt*. Les classes *Infantry* et *Horseman* ont quand à elle été dupliquées et implémentent l'interface *Soldier*(nous avons par la suite intercalé une classe abstraite 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. La résistance de l'arme diminue de 1 point à chaque utilisation (c'est-à-dire à chaque appel aux méthodes *strike* ou de *parry*).

3.1.3 Composite

La seconde étape dans le projet consistait à rajouter la possibilité pour le client de créer des groupes armés. Un groupe armé de soldats peut être composé de plusieurs groupes armés, eux-mêmes décomposables en sous-groupes armés, et ainsi de suite. Cette notion de groupes armés fait ici apparaître une structure d'arbre avec des éléments composés d'autres éléments, structure qui fait penser à une hiérarchie. C'est pour cela que nous avons appliqué le pattern Composite, visible en figure 3.3 page 14, en créant une classe *ArmedGroup* contenant une référence vers un objet *Soldier* (qui représente notre *procurateur*, ce que manipule le client).

3.1.4 Visiteur

Nous avions à la fois des groupes armés ainsi que des soldats de différentes catégories. Afin de faciliter l'ajout de nouvelles fonctionnalités à la fois sur les groupes armés et sur les soldats, nous avons mis en place le pattern *Visiteur*. Sans ce pattern, à chaque ajout de fonctionnalités, nous étions obligés de rajouter les méthodes dans nos classes *Horseman*, *Infantry* et *ArmedGroup*, d'autant plus si nous voulions faire des traitement différents en fonction de chaque classe. Grâce au pattern *Visiteur*, l'ajout de fonctionnalités supplémentaires (comme l'affichage de tous les soldats formant un groupe armé ou le comptage 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 le reste du code. Nous avons créé une classe *SoldierPrintVisitor* pour ajouter la fonction d'affichage des soldats ainsi que la classe *SoldierCountVisitor* pour ajouter la fonction de comptabilisation. Vous pouvez voir un exemple du déroulement des appels de méthodes en figure 3.4 page 15. Tout ceci est totalement transparent vis-à-vis du reste du code : c'est là tout l'intérêt du pattern *Visiteur*. On n'applique donc qu'un nombre minimal de modifications sur les autres classes de l'application et nous avons la possibilité de faire des traitements spécifiques 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 problématiques de ce projet était de pouvoir suivre le déroulement des conflits au fur et à mesure du déroulement de l'action. Nous aurions pu produire de nombreux print dans nos fonctions, ce qui aurait généré des affichages au fur et à mesure du déroulement des batailles. Cependant, cette méthode est peu flexible et peu maintenable! C'est donc ici que le pattern *Observateur* va être intéressant, car il nous permet d'observer un objet et d'être notifié de ses changements afin d'appliquer les traitements adéquats suite à ces changements. Ainsi, pour afficher au fur et à mesure les noms des soldats morts au sein d'un groupe armé ainsi que les armés détruites, nous avons mis en place le système visible en figure 3.5 page 15. Il est composé en premier lieu d'une interface *Observer* avec pour sous-classe *DeadsObserver*. Cette dernière 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'*Observers*, liste qui permet à l'objet, lorsqu'il change, d'en informer tous ses observateurs. Cette classe est également intéressante car elle 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'un soldat meurt, on pourra notifier tous les objets qui dépendent du mort. L'inconvénient et la limite du pattern est que le graphe des relations d'observation peut vite devenir complexe. En effet, si l'on veut mettre en place le système des envois de télégrammes d'excuse à ses amis, c'est à dire à des soldats de l'armé, il va falloir que chaque objet soldat soit observé par tous les autres objets soldat « amis », ce qui implique la création de nombreux objet de type Observer (autant d'observer que de soldat) 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*. Il donne accès à l'instance de l'observateur via la méthode getInstance(), méthode qui renvoi la variable *Instance* (déclarée *private*). Le constructeur est quand à lui rendu non visible depuis l'extérieur de la classe.

3.1.7 Fabrique abstraite

La dernière partie du projet consistait à créer plusieurs familles de soldats historiquement cohérentes, c'est-à-dire que pour une famille donnée, le type d'armement devait être conforme à son époque.

Pour le client, la notion d'époque vis-à-vis des armes doit rester transparente. En effet, on veut associer les bonnes armes aux soldats construits par le client sans qu'il ait à se soucier de la cohérence de son habillement. Il n'aura donc pas à spécifier lui même les classes concrètes correspondant aux armes d'une époque ou d'une autre pour habiller son soldat. Ainsi, le client n'aura pas accès au processus de construction de son équipement (telle arme de défense pour telle époque, telle arme d'attaque pour telle époque), tout se fera automatiquement.

Pour réaliser cette demande, nous avons utilisé le pattern *Fabrique abstraite*.

Nous avons donc créé une classe *AbstractFactory* comportant les méthodes *setDefensiveWeapon(Soldier s)* et *setOffensiveWeapon(Soldier s)*, qui permettent d'affecter à un soldat donné une arme défensive/offensive conforme à l'époque. Ces méthodes se chargeront alors de la construction de l'équipement souhaité, en toute conformité avec l'époque. L'implémentation de ces deux méthodes se trouve dans les classes *MiddleAgeFactory* et *ScienceFictionFactory*, qui représentent deux époques différentes. L'armement pourra ainsi être construit différemment en fonction de l'époque (par exemple les soldats de l'époque moderne seront équipés de lasers). Nous avons également rajouté des méthodes dans la fabrique permettant de construire des soldats de type différent (soldat à pied, soldat monté).

Ainsi, grâce au pattern *Fabrique abstraite*, nous pouvons garantir le maintient de la cohérence de nos équipements et de nos soldats en fonction de l'époque.

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.

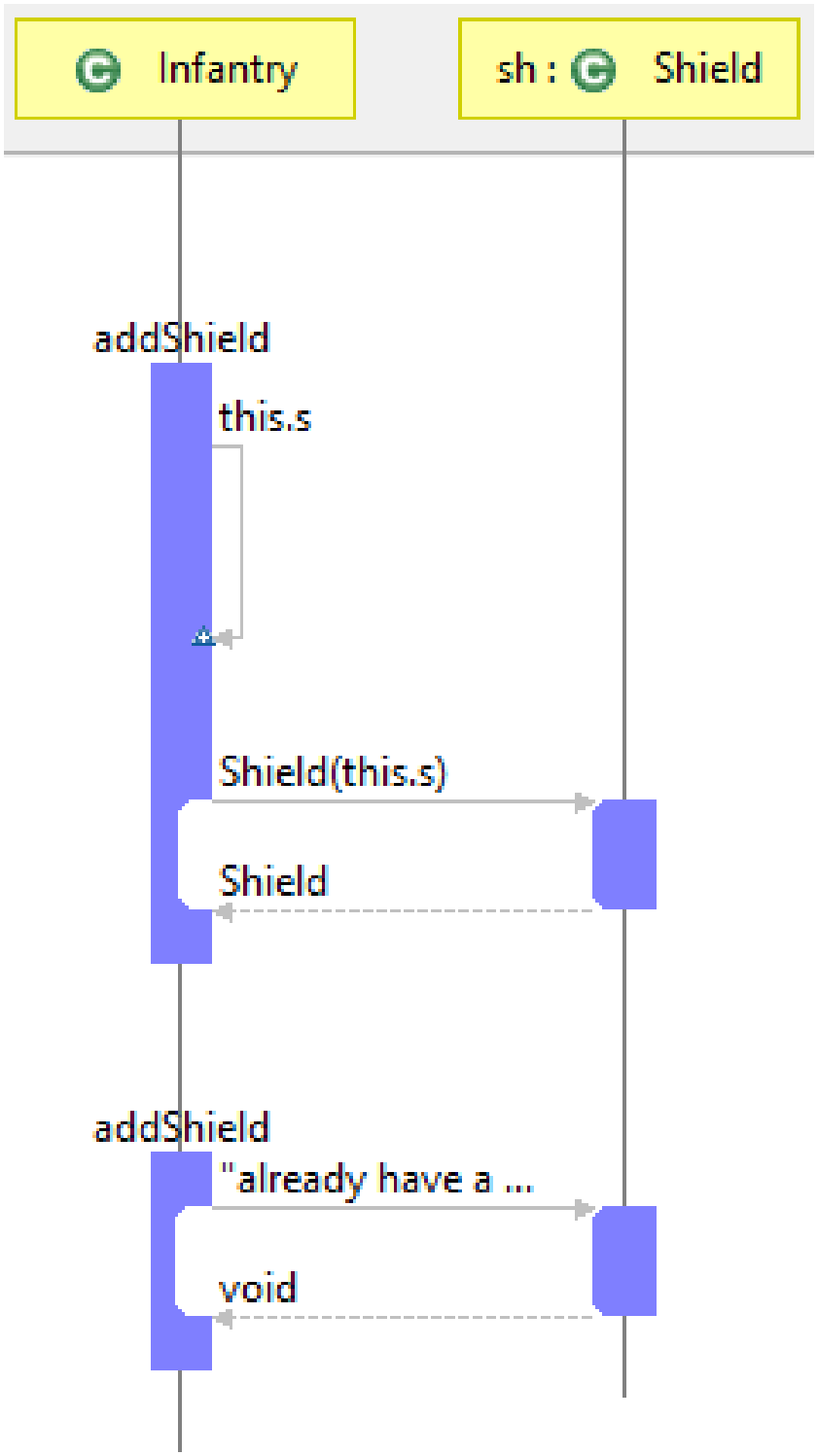

FIGURE 3.2 – Diagramme de séquence du pattern procurateur

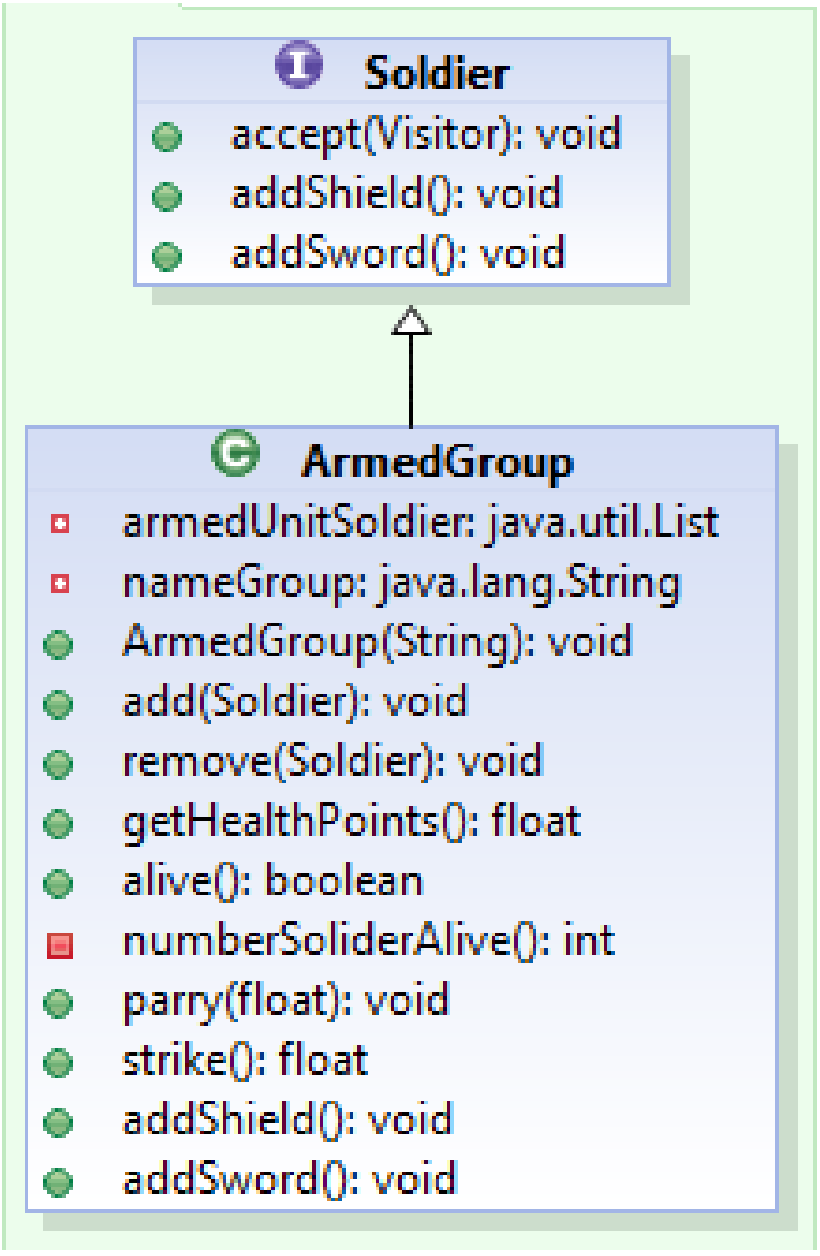

FIGURE 3.3 – Diagramme de classes du pattern composite

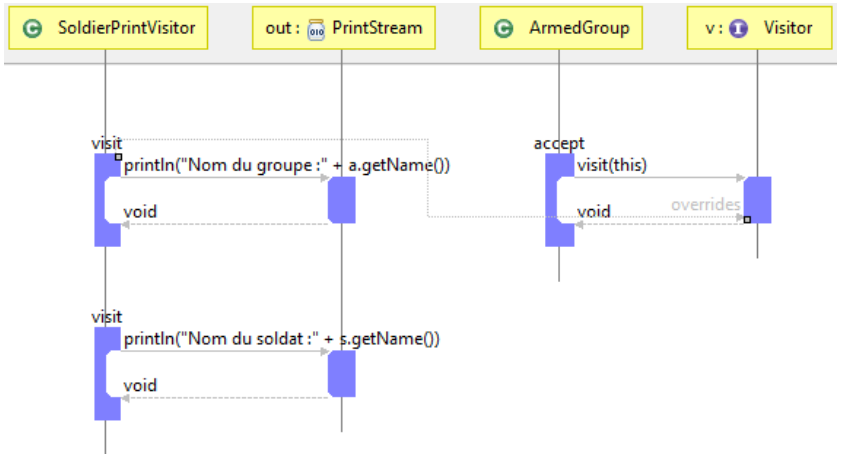

FIGURE 3.4 – Diagramme de séquence du pattern visiteur

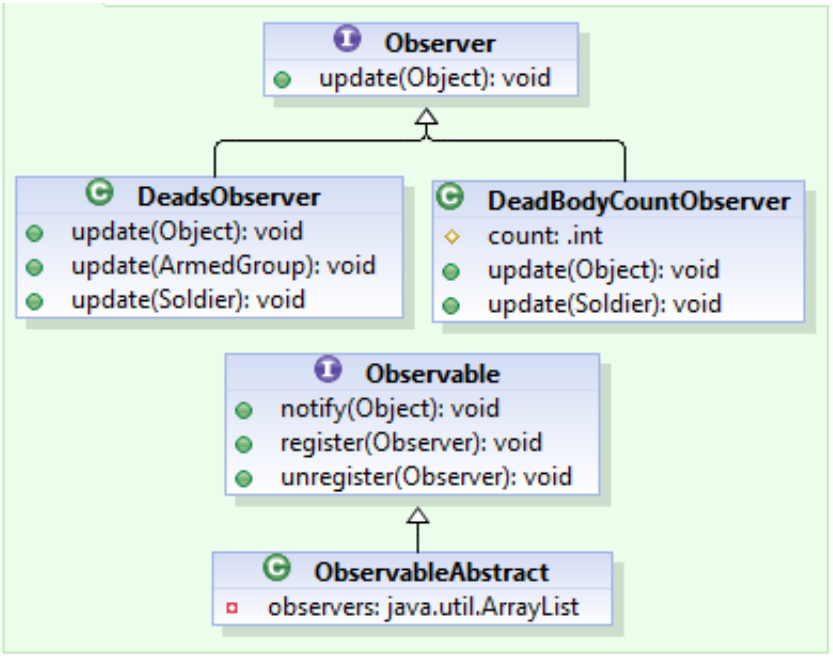

FIGURE 3.5 – Diagramme de classes du pattern observeur

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.