Guerin Yoann
Noleau Baptiste

# Architecture Logicielle 2012-13

# Soldats décorés & Armées visitées, observées et fabriquées

## Sommaire

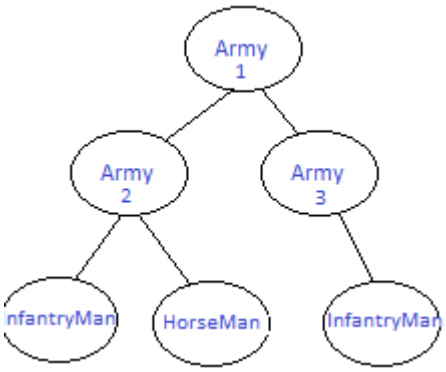
## Décorateur

## Proxy

## Composite

### But

Il faut être capable de créer des groupes de soldats afin de constituer une armée. Une armée peut contenir plusieurs sous groupes. On va donc mettre en place une structure arborescente. La classe **Army** sera un nœud, tandis que les **Infantryman** et **Horseman** seront les feuilles.




Cette configuration sera reprise pour les tests unitaires

### Implémentation

**Army** est le composite, il hérite de **Soldier** tout comme **HorseMan** et **InfantryMan** à la différence qu'il détient une liste de **Soldier** : créant ainsi de la composition.

➔Héritage + composition = récursivité

**Army** va déléguer les taches sur son père **Soldier.** Il n'a conscience que de ses fils contenu dans la liste de **Soldier** et leur applique ses méthodes.

Les méthodes ont été faites selon les consignes. La méthode parry était un peu ambigüe, nous l'avons interprété comme ceci :

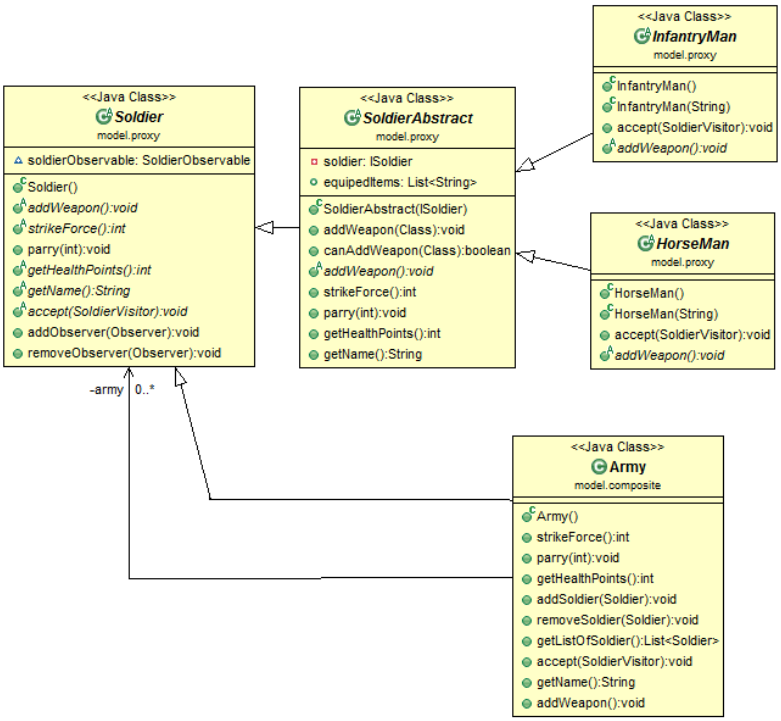
```java
@Override
public void parry(int damage) {
    int damagePerSoldier  = damage / this.army.size();
    for (Soldier s : army)
        s.parry(damagePerSoldier);

    super.parry(damagePerSoldier);
}
```

Sera détaillé plus tard pour le pattern Observer

- La répartition des dégâts se fait à partir du nœud où la méthode est appelée. Si on demande à parer 300 dégâts depuis *Army1* ,150 dégâts seront envoyés à Army2 et Army3. Etant donné que ce sont des nœuds, la méthode parry de **Army** est rappelée et les dégats sont de nouveaux partagés entre les fils. InfantryMan et HorseMan de l'Army2 devront parer chacun 75 dégâts, tandis que l'infantryMan de l'Army3 devra parer tout seul 150 dégâts.

### *Diagramme de classes*




## Visiteur

## Observer

### But

Il faut afficher les noms des Soldats/groupes qui sont morts, ainsi que le nombre de victimes en temps réel.

### Implémentation

Notre interface **Soldier** est l'observable. Plutôt que de lui faire étendre la classe **java.util.Observable**, nous avons décidé d'utiliser la délégation.

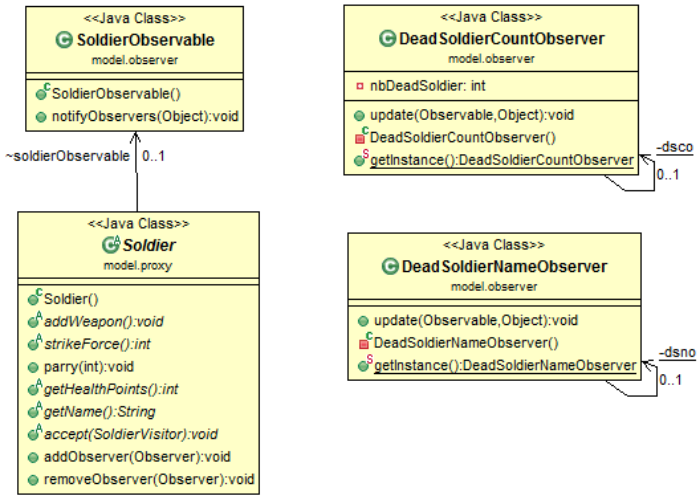
Il a été nécessaire de transformer l'interface Soldier en classe abstraite. C'est à l'intérieur que l'on spécifie que l'on prévient les observateurs quand un soldat ou une armée pare (notifyObservers). Comme nous utilisons les méthodes de la classe **java.util.Observable** (non redéfinies), il est nécessaire d'appeler setChanged().

C'est dans les méthodes update() de nos observateurs que nous contrôlons la vie des soldiers et agissons en conséquence. Cela peut devenir coûteux lorsque l'on a beaucoup d'observateurs = il serait peut être préférable de faire le test et de ne prévenir les observateurs que lorsque c'est nécessaire (vie = 0).

Nous avons donc affiché les noms de soldats mort ainsi que des groupes (pour les groupes nous faisons un affichage de tel sorte que l'on sache quels soldats étaient dedans).

Pour le nombre de soldats morts au fur et à mesure, nous ne comptabilisons que les soldats et nous les groupes.

#### *Diagramme de classes*




## Singleton

### But

Avoir une seule instance de nos observateurs.

### Implémentation

Nous avons directement intégré le code des singleton dans les Observateurs plutôt que de créer un Singleton abstrait ➔Cela n'aurait pas eu beaucoup d'intérêt car ça n'aurait pas enlevé beaucoup de code à écrire.

## Abstract Factory

## But

## Implémentation

Ajout de 2 armes supplémentaires : LightSaber et Winchester dans le package du décorateur (extends AbstractWeapon).

Nous avons remplacé les méthodes addSword et addShield (contenues dans le proxy) par une méthode générique addWeapon(Class weaponType). Un attribut a également été ajouté à la classe Soldier : Une liste contenant les objets qui lui ont été équipé.

Au niveau du port d'arme, nous avons interprété le sujet comme ceci :

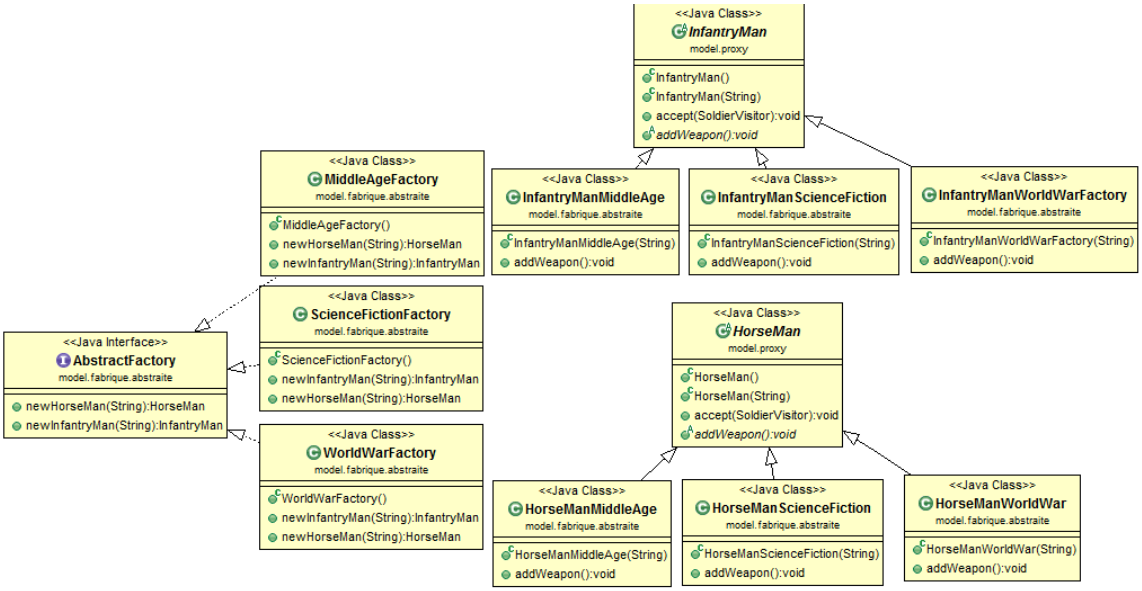
- MiddleAge :
  - Sword : possibilité d'en équiper une
  - Shield : possibilité d'en équiper un
    ➔Une arme différente dans chaque main
- ScienceFiction :
  - LightSaber : Possibilité d'en avoir 2 (un par main)
    ➔ Une arme de même type main gauche et main droite
- WorldWar :
  - Winchester : possibilité d'en équiper un
    ➔L'arme occupe les deux mains, on ne peut avoir qu'une

Pour mettre en place cette stratégie (savoir si on peut équiper telle ou telle arme) il a été nécessaire d'utiliser l'introspection. Chaque Arme possède des attributs accessibles par des getter afin de savoir si on peut l'équiper aux 2 mains (gauche ou droite) et si c'est une arme qui occupe les deux mains (isOneHand & canBeEquipedInBothHands).
A ce niveau la, la stratégie est donc décrite dans la classe Abstraite SoldierAbstract du proxy. Nous aurions pu la mettre en place dans les produits concrets mais cela aurait entrainé une grande duplication de code.
Nous avons considéré que quelque soit la période (moyen age, science fiction, il n'est pas possible d'avoir 2 boucliers en même temps, 2 snipers, …). Par ailleurs, si une arme vient se rajouter au panel, il suffit juste de la créer et de renseigner ses attributs pour avoir le comportement que l'on désire. La méthode addWeapon() fonctionnera avec cette nouvelle arme sans qu'il soit nécessaire de la modifier.

Les produits concrets héritent directement de InfantryMan/HorseMan du pattern Proxy : ainsi il est possible de réutiliser les méthodes précédentes sans avoir à les redéfinir où à faire de délégation ➔ économie de code.
Les armes à utiliser selon les époques sont spécifiées dans les produits concrets, la méthode addWeapon() est redéfinie.
La méthode addWeapon() n'accepte plus de paramètre dans la classe Army, chaque soldat est armé avec une arme de son époque.

Nous sommes conscients que notre solution n'est pas très « objet » en utilisant l'introspection mais elle permet néanmoins d'économiser beaucoup de lignes de code et de faciliter l'extension du programme si des développeurs sont amenés à ajouter de nouvelles armes.

### *Diagramme de classes*




## Tests unitaires

Deux fonctions permettent de tester notre fabrique abstraite.

Dans la première (`addWeaponFactory`) nous testons les soldats individuellement afin de vérifier que les armes ont bien été ajoutées aux soldats en fonction de leur famille :

- Respect des caractéristiques de l'arme (combinaison possible ou pas avec d'autres armes)
- Leur force d'attaque a bien augmenté.

Dans la deuxième (`addWeaponArmyFactory`) nous équipons des sous-groupes et vérifions que les soldats qui sont dedans ont bien été modifiés.