Cours Patterns

TD N°3 : Design Patterns Gof structurels

Enseignant : Evelyne VITTORI

Les exercices sont indépendants les uns des autres.

Exercice 1 - Jeu d'aventures - Adapter

On considère un simulateur de jeu d'aventure permettant de gérer différents personnages de plusieurs types (humains, trolls, orcs, etc.). Chaque personnage peut se battre mais il doit aussi savoir marcher, courir et sauter.

On dispose d'une interface Deplacable (cf. figure 1) pour regrouper l'ensemble des savoir-faire liés aux déplacements.

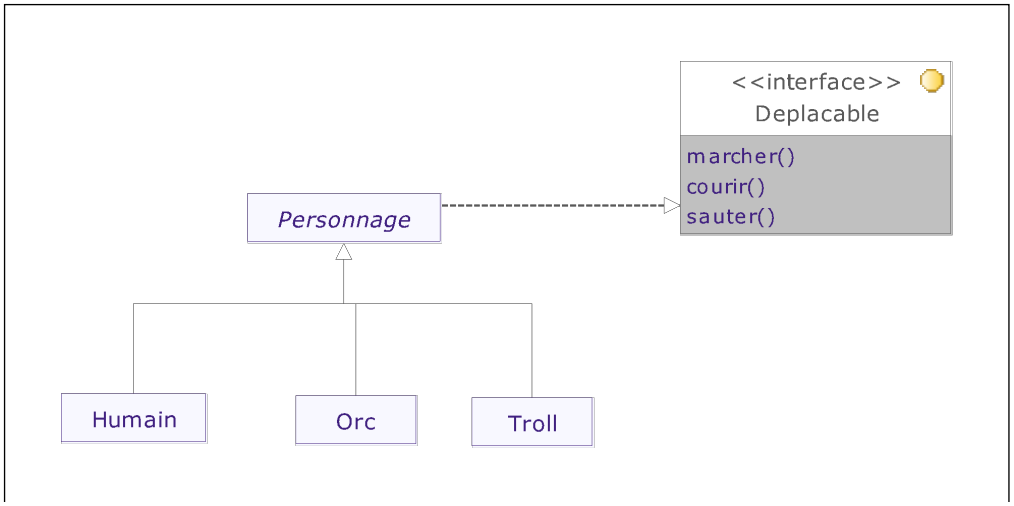

Figure 1 : Interface Deplacable

Comme le montre la figure 1, chaque type de personnage est modélisé par une classe concrète qui doit implémenter l'interface Deplacable .
On suppose à présent que l'on doit ajouter un nouveau type de Personnage, les taurens. Une autre équipe de concepteurs, nous propose de nous fournir une classe déjà développée et possédant des fonctionnalités similaires. Il s'agit de la classe TaurenEtranger (cf. figure 2).
Les méthodes avancer() et trotter() correspondent respectivement à des implémentations des méthodes marcher() et courir() de l'interface Deplaçable.

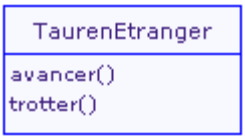

Figure 2 : Classe Tauren Etranger

Pour éviter de recoder inutilement des méthodes qui existent déjà, on souhaite réutiliser les méthodes cette classe TaurenEtranger. Malheureusement, nous ne pouvons pas l'intégrer directement dans notre hiérachie et nous ne pouvons pas non plus la modifier. Le pattern Adapter nous propose deux solutions.

Question 1.1 : Utiliser le pattern Adapter dans sa version « Class Adapter » pour résoudre ce problème. Définissez un diagramme de classe illustrant votre solution.

Question 1.2 : Utiliser le pattern Adapter dans sa version « Object Adapter » pour résoudre ce problème. Définissez un diagramme de classe illustrant votre solution.

Question 1.3 : Coder en Java votre solution de la question 1.2 en utilisant la classe simplifiée TaurenEtranger suivante :

```
public class TaurenEtranger {
	public void avancer(){
		System.out.println("Le tauren se met à avancer!");
	}
	public void trotter(){
		System.out.println("Le tauren avance au trot!");
	}
}
```

Vous définirez un programme de test permettant de créer un personnage de type Tauren, de le faire marcher puis courir.
L'exécution du programme doit donner lieu à un affichage console similaire à l'affichage suivant :

```
Le tauren se met à avancer!
Le tauren avance au trot!
```

Exercice 2 - Simulateur de jeu d'aventures - Decorator

On considère un simulateur de jeu d'aventures disposant de deux catégories de personnages : les humains et les taurens et l'on s'intéresse à la modélisation de leurs déplacements.
Une première modélisation a conduit à la définition du diagramme de classe de la figure 3.

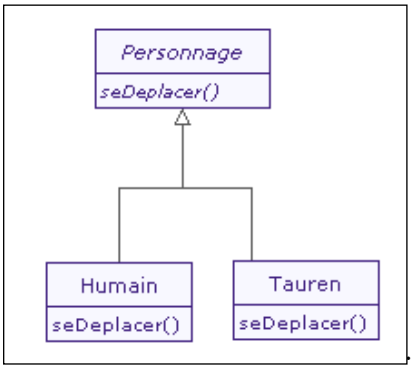

Figure 3 : Diagramme de classe Personnage

La méthode seDeplacer() définit le comportement des personnages lors de leur déplacement. Chaque type de personnage (humain ou tauren) possède un comportement de déplacement spécifique.
On souhaite à présent permettre à ce comportement d'être complété par de nouvelles actions en fonction de l'acquisition d'un ou plusieurs pouvoirs par le personnage.
Ainsi selon les pouvoirs qu'il aura acquis, lorsqu'on lui demandera de se déplacer, un personnage pourra non seulement se déplacer mais en plus devenir invisible et/ou déclencher un orage.
N'importe quel personnage pourra acquérir n'importe quel pouvoir. On peut ainsi imaginer toutes les combinaisons possibles de pouvoirs.
Pour l'instant, on se limite aux deux pouvoirs évoqués ci-dessus mais la conception doit permettre d'enrichir le jeu par la définition de pouvoirs supplémentaires.

Question 2.1 Proposer un diagramme de classe permettant de modéliser ce problème en utilisant le design **pattern Decorator**.

Question 2.2 Coder en Java les classes de votre diagramme.
Définissez un programme de test implémentant le scénario suivant :

- définition d'un personnage de type humain ayant pour nom Titi et possédant les pouvoirs de devenir invisible et de déclencher des orages
- déplacement du personnage (ce déplacement doit se traduire par le fait de se déplacer et devenir invisible et déclencher un orage).

L'exécution de ce programme doit avoir pour résultat un affichage console tel que :

```
L'humain de nom Titi commence à
avancer.
Titi devient invisible!!
Titi déclenche un orage!!
```

Question 2.3 Illustrer le fonctionnement de votre programme de test en définissant un diagramme de séquence modélisant son exécution.

Exercice 3 - Forêt enchantée - Composite

Dans le cadre de la modélisation d'une forêt enchantée, on souhaite modéliser des éléments magiques tels que des arbres. Chaque arbre peut contenir des nids magiques. Certains nids renferment des phénix qui, une fois réveillés, peuvent rendre un personnage immortel. D'autres nids contiennent des gemmes magiques qui augmentent

la force du personnage de 10 points. Les nids peuvent également être vides ou renfermer d'autres nids avec différents éléments magiques à l'intérieur. Les nids contenant des phénix ou des gemmes ne peuvent pas contenir d'autres nids.

L'interaction avec un nid doit déclencher l'interaction en cascade de chaque nid qu'il contient. Si un personnage interagit avec un nid de phénix, il devient immortel et ne peut donc plus être tué par les dangers de la forêt.

Lorsqu'un personnage rencontre un arbre magique, il doit pouvoir interagir avec tous les nids de l'arbre mais aussi avec un nid spécifique en précisant son numéro d'ordre.

Question 3.1 : Utilisez le pattern Composite pour résoudre ce problème et modéliser les nids. Proposez un diagramme de classe illustrant votre solution.

Question 3.2 : Codez en Java une version simplifiée de votre solution. Mettez en place un scénario de test comprenant les étapes suivantes :

- Créez trois nids: un nid contenant une gemme, un nid contenant un phénix et un nid renfermant deux autres nids (un nid avec une gemme et un nid vide).
- Créez un arbre appelée "Arbre Lumineux" qui contient les trois nids précédents.
- Créez deux personnages: "Luna" et "Orion".
- Faites interagir Luna avec tous les nids de l'Arbre Lumineux
- Faites interagir Orion uniquement avec le nid complexe (nid numéro 3).
- Affichez l'état final des personnages.

L'exécution du programme devrait ressembler à :

```
CREATION DE L'ARBRE MAGIQUE : Arbre Lumineux qui contient 3 nids
Nid N° 1 : Nid gemme
Nid N° 2 : Nid phénix
Nid N° 3 : Nid Complexe contenant 2 nids

INTERACTION AVEC Luna
Luna se trouve face à Arbre Lumineux qui contient 3 nids
Luna décide d'interagir avec tous les nids de l'arbre
-Interaction avec le nid N° 1 : Nid gemme
Luna a trouvé une gemme magique : sa force augmente de 10 points!
-Interaction avec le nid N° 2 : Nid phénix
Luna a réveillé un phénix : il devient immortel!
-Interaction avec le nid N° 3 : Nid Complexe contenant 2 nids
Luna a trouvé une gemme magique : sa force augmente de 10 points!
Luna n'a rien trouvé dans ce nid!

INTERACTION AVEC Orion
Orion se trouve face à Arbre Lumineux qui contient 3 nids
Orion décide d'interagir avec le nid numéro 3 : Nid Complexe
contenant 2 nids
Orion a trouvé une gemme magique : sa force augmente de 10 points!
Orion n'a rien trouvé dans ce nid!

ETAT FINAL DES PERSONNAGES
Luna est immortel avec une force de 20 points
Orion a une force de 10 points
```

Exercice 4 - Design Pattern Proxy

On considère la modélisation d'une application de vente en ligne de véhicules.
L'application comporte notamment une classe FicheVehicule représentant le composant graphique chargé d'afficher la fiche d'un véhicule et les informations associées. Cette fiche offre également la possibilité pour chaque véhicule de visualiser un film qui présente le véhicule.

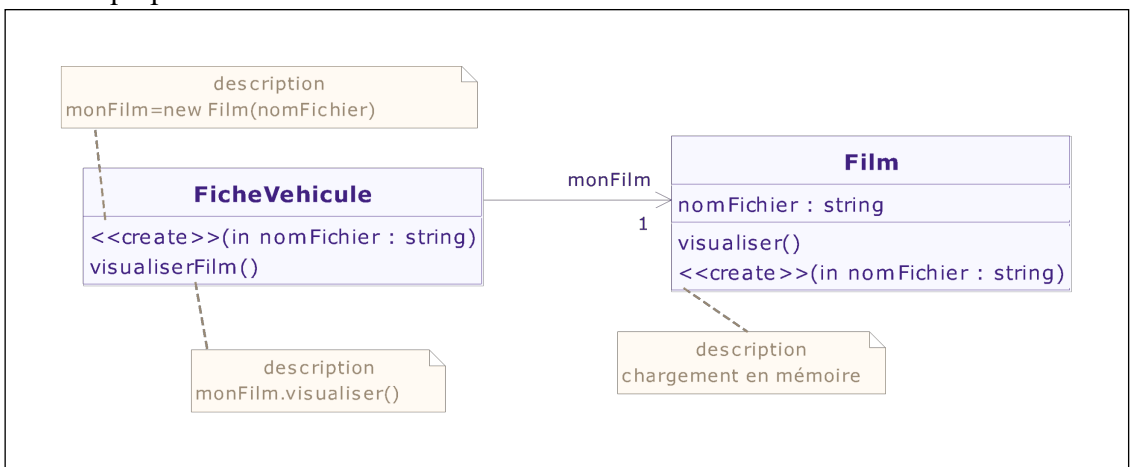

Figure4 : Diag Classe Vehicule-Film

Le diagramme de la figure 4 propose une solution mais on vous demande de l'améliorer sans modifier le corps des méthodes de la classe Film.
En effet, dans la mesure où un film est un objet très lourd à gérer en mémoire (place occupée + temps de chargement), l'objet Film ne devra être créé (et donc chargé en mémoire) que lorsque l'utilisateur demande effectivement sa visualisation en invoquant la méthode visualiserFilm.
Une solution consiste à afficher une photo dans la fiche du véhicule à la place du film lors de la création de la fiche et de déclencher la création et la visualisation du film uniquement lorsque la méthode visualiserFilm est invoquée pour la première fois.
Le pattern Proxy nous donne une solution pour définir cette conception sans modifier la classe Film.

Question 4.1 Définissez un diagramme de classe représentant la solution.

Question 4.2 Coder en java votre solution et définissez un programme de test réalisant les actions suivantes :

1. Création d'une fiche véhicule avec le film contenu dans le fichier « film.mpg » et la photo associée dans le fichier « photo.jpg ».
2. Visualisation du film.

L'exécution du programme doit donner lieu à un affichage console similaire à l'affichage suivant :

```
AFFICHAGE DE LA FICHE VEHICULE
Affichage de la photo : photo.jpg
DEMANDE DE VISUALISATION DU FILM
Visualisation réelle du film (fichier film.mpg)
```