TD 6 : Abstraction et *design patterns*

Objectifs pédagogiques :

- **abstraction et interfaces**
- ***Design Pattern Decorator***
- ***Design Pattern Composite***
- ***Design Pattern Adapter***

6.1 Abstraction (interfaces)

Nous considérons un catalogue d'articles à vendre. Il existe différentes classes d'articles : livres, films, etc. Chaque article a un nom (`String`) retourné par `getName()` et un prix (`int`) retourné par `getPrice()`, ainsi que des attributs qui varient d'une classe à l'autre. Par exemple, un livre aura un nombre de pages, tandis qu'un film aura un réalisateur. A priori, nous avons donc des classes `Book` et `Movie` comme ci-dessous :

```
public class Book {
   public String getName() ...
   public int getPrice() ...
   public int getPages() ...
   ...
}
```

```
public class Movie {
   public String getName() ...
   public int getPrice() ...
   public String getDirector() ...
   ...
}
```

Un catalogue est une collection hétérogène d'objets (livres, films, etc.) ayant pour seule caractéristique commune d'avoir un nom et un prix. Si nous appelons `Item` le type des articles d'un catalogue, alors nous aimerions que le code suivant affiche à l'écran la liste des articles et leur prix:

```
public static void printCatalogue(Collection<Item> items)
{
   for (Item i : items)
      System.out.println(i.getName() + " : " + i.getPrice() + " EUR");
}
```

Question 1. Proposez une définition de `Item`. Indiquez comment modifier les classes `Book` et `Movie` pour permettre à `printCatalogue()` d'afficher un catalogue contenant des livres et des films.

6.2 Ajout de fonctionnalité (Decorator)

Nous souhaitons pouvoir moduler le prix d'un article pour proposer des promotions. Il s'agit d'ajouter à chaque article une méthode `setRate(float rate)` qui fixe un taux, entre 0 et 1. Le prix retourné par `getPrice` est alors le prix de base, multiplié par ce taux. Avant le premier appel à `setRate`, le taux vaut 1 (`getPrice` retourne le prix de base).

Question 2. Montrez comment le *Design Patter Decorator* permet cet ajout grâce à l'introduction d'une classe `RateDecorator`, sans avoir à modifier les interfaces et classes existantes. Donnez le diagramme UML. Donnez le code de `RateDecorator`. Donnez enfin un exemple de création d'un livre (classe `Book`) avec une promotion de 50 %.

Question 3. Supposons que nous définissons dans la classe `Book` une méthode `toString` de la manière suivante :

```
@Override public String toString() { return getName() + " " + getPrice(); }
```

Quel est le résultat de `System.out.println` sur un livre décoré avec une promotion de 50% ? Vous pouvez dessiner un diagramme des objets en présence pour vous aider. Comment corriger le problème ?

6.3 Lots d'articles (Composite)

Nous souhaitons ajouter à nos catalogues des lots. Un lot est un article composé d'autres articles (par exemple, un coffret de livres ou de films). Nous appelons `Box` la classe des lots. Un lot possède :

- un nom qui lui est propre, accessible avec `getName()` ;
- un prix, `getPrice()`, calculé comme la somme des prix des articles contenus dans le lot ;
- des méthodes `addItem(Item)` et `removeItem(Item)` pour ajouter ou supprimer un article ;
- une méthode `getItems()` pour retourner la collection des articles du lot.

Le *Design Pattern Composite* nous permet d'ajouter la gestion des lots sans modifier aucune des classes et interfaces existantes et sans modifier le code client `printCatalogue` ci-dessus. Grâce à ce *Design Pattern*, nous pourrons proposer des lots hétérogènes, contenant à la fois des livres et des films, et même des lots contenant d'autres lots. Par ailleurs, si nous ajoutons dans le futur de nouveaux types d'articles, ceux-ci pourront être inclus dans des lots sans avoir à modifier la classe `Box`.

Question 4. Donnez un diagramme UML des classes et interfaces en présence, notamment celles de la question 1 ainsi que `Box`. Est-il nécessaire d'introduire une nouvelle interface ?

Question 5. Donnez l'implantation de la classe `Box`.

Question 6. Soit `box` une variable de type `Box` et `book` une variable de type `Book`. Quel sera l'effet de `box.getItems().add(book)` ? Discutez les avantages et les inconvénients de ce comportement.

Question 7. Nous souhaitons programmer au niveau du client (donc hors des classes `Book`, `Box`, etc.) une méthode pour afficher la liste de tous les articles (sans leur prix), y compris ceux contenus dans des lots : `public static void printAllItems(Item i)`.

Donnez le code de cette méthode.

Question 8. Discutez les avantages et les inconvénients d'incorporer les méthodes `addItem`, `removeItem`, `getItems` dans le type `Item`.

6.4 Intégration d'un composant tiers (Adapter)

Nous supposons qu'une bibliothèque développée indépendamment de notre catalogue propose une hiérarchie de classes :

```
public class Sofa {
   public Sofa(String model, int price) ...
   public String model() ...
   public int price() ...
   public int color() ...
   ...
}
```

```
public class Convertible extends Sofa {
   public Convertible(String model, int price) ...
   public void open() ...
   public void close() ...
   ...
}
```

Question 9. Nous souhaitons ajouter ces objets comme articles dans notre catalogue. Il n'est cependant pas permis de modifier les classes `Sofa`, `Convertible`, etc. de la bibliothèque tiers. A priori, nous pouvons penser à utiliser l'héritage ou la délégation. Donnez une solution utilisant la délégation. Discutez ses avantages et ses inconvénients par rapport à l'utilisation de l'héritage.