TD 6 : Abstraction et *design patterns*

Objectifs pédagogiques :

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

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.

La bonne solution est de définir une interface `Item` déclarant les fonctionnalités communes et uniquement celles-ci :

```
public interface Item {
   String getName();
   int getPrice();
}
```

Il faut ensuite indiquer que `Book` et `Movie` implantent cette interface en ajoutant `implements Item` à `public class Book` et `public class Movie`.

Il y a d'autres solutions possibles, mais elles ont des inconvénients :

– une classe abstraite `Item` : ceci mobilise le lien d'héritage et laisse à penser qu'`Item` contient une implantation (même partielle), alors qu'en réalité nous spécifions uniquement les signatures ;

– une classe `Item` : tous les inconvénients de la classe abstraite, plus la nécessité de donner des implantations par défaut de `getName` et `getPrice` qui n'ont pas de sens (et donc lancent une exception) ; rien n'empêche alors de créer un objet de classe `Item`, d'implantation partielle. La vérification se fait à l'exécution (exception) et non à la compilation (typage).

Rappelons que les interfaces n'ont pas d'attribut, pas de constructeur, et généralement pas d'implantation de méthodes (bien que Java 8 l'autorise, avec le modificateur `default`). Toutes les méthodes déclarées sont par défaut `public` et `abstract`, donc inutile de préciser ces modificateurs.

6.2 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 :

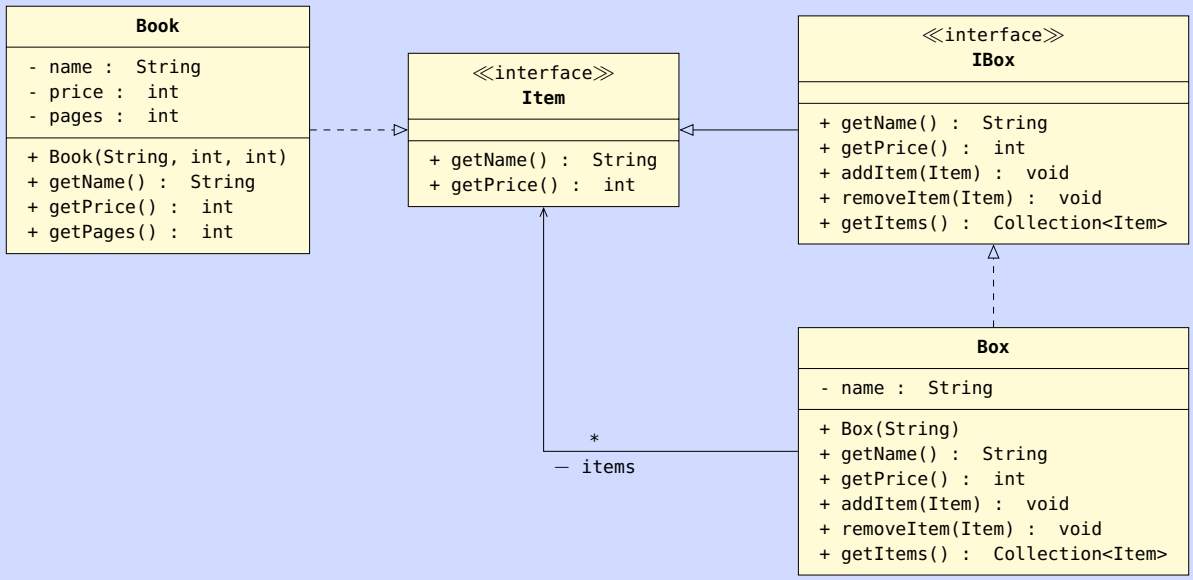
– un nom qui lui est propre, accessible avec `getName()` ;
– un prix, `getPrice()`, calculé en faisant la somme des prix des articles contenus dans le lot ;
– des méthodes `addItem(Item)` et `removeItem(Item)` pour ajouter ou supprimer un article ;
– une méthode `getItems()` pour retourner la collection 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, des lots contenant d'autres lots, et même des lots contenant des types d'articles qui ne seront définis qu'ultérieurement.

Question 2. Donnez un diagramme UML des classes et interfaces en présence, notamment celles de la question précédente ainsi que `Box`. Est-il nécessaire d'introduire une nouvelle interface ?

Il n'est pas *nécessaire*, pour le moment du moins, d'ajouter de nouvelle interface. Il est néanmoins toujours *utile* de découpler l'implantation du composite de celle du client en introduisant une interface `IBox`, donc nous le faisons ici (voir aussi la question 6 sur l'énumération des articles d'un catalogue). Cela permet d'envisager, dans le futur, d'autres composites qui ont la même interface mais une implantation différente, ou bien d'adapter des composites existants à notre client, etc.

Notons aussi que `addItem`, `removeItem`, `getItems` sont uniquement dans `IBox` et pas dans `Item`, puisque les `Item` ne sont pas tous des composites. Voir également la question 6 pour un choix alternatif.



Question 3. Donnez l'implantation de la classe `Box`.

Ce code propose une version naïve de `getItems`. Voir la question suivante pour une discussion des autres choix possibles.

```
public interface IBox extends Item {
   void addItem(Item item);
   void removeItem(Item item);
   Collection<Item> getItems();
}
```

```
public class Box implements IBox {

   private final String name;
   private List<Item> items = new LinkedList<>();

   public Box(String name) { this.name = name; }

   @Override public String getName() { return name; }

   @Override public int getPrice() {
      int p = 0;
      for (Item i : items) p += i.getPrice();
      return p;
   }

   @Override public void addItem(Item item) { items.add(item); }
   @Override public void removeItem(Item item) { items.remove(item); }
   @Override public Collection<Item> getItems() { return items; }
}
```

Question 4. 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.

La réponse dépend de la manière dont est implantée la collection d'articles dans `Box`.
Si nous faisons naïvement :

```
@Override public Collection<Item> getItems() { return items; }
```

alors `box.getItems().add(book)` va ajouter `book` au lot. Il a donc le même effet que `box.addItem(book)`.
Une autre solution est que `getItems()` retourne une copie de la liste. Ceci se fait assez simplement, grâce aux constructeurs de collections :

```
@Override public Collection<Item> getItems() { return new LinkedList<>(items); }
```

La version avec copie est plus coûteuse, mais offre certains avantages :

– Elle garantit l'encapsulation en interdisant au client d'accéder directement à la représentation interne du composite.

– Elle permet de garder un comportement cohérent si le type de `items` est changé en un type qui n'implante pas `Collection` (par exemple un tableau nu Java). Dans ce cas, `items` ne peut pas être retourné directement par `getItems()` et il devra être copié dans une classe qui implante `Collection`. Forcer la copie dans tous les cas évite, pour le client, un comportement différent en fonction des choix d'implantation liés à `items`.
– Elle permet d'écrire une itération qui modifie le contenu d'un lot :

```
for (Item i : box.getItems()) { if (...) box.removeItem(i); }
```

En effet, il n'est pas possible de modifier une liste dans le corps d'une boucle qui la parcourt (une exception `ConcurrentModificationException` est signalée). Ici, la boucle parcourt une copie de l'attribut `items`, ce qui permet à `removeItem` de modifier l'attribut original dans le corps de la boucle.
– Elle permet de modifier `addItem` et `removeItem` pour avoir un comportement plus complexe (voir la question 7, ou aussi l'utilisation d'un observateur qui n'est pas considérée dans ce TD). Un appel à `box.getItems().add(book)` ne permet pas à `Box` d'être informé d'un ajout, alors que `box.addItem(book)` le permet. Il est donc nécessaire d'obliger le client à utiliser `addItem`.

Notez que c'est une copie de surface (voir la question suivante).

Question 5. Montrez comment implanter une méthode `clone()` qui fait une copie d'un article. La copie doit se faire en profondeur : lors de la copie d'un lot, il sera fait (récursivement) une copie de chaque article du lot. Donnez en particulier la méthode `clone()` de la classe `Box`.

Il faut :
– ajouter dans l'interface `Item` la signature `Item clone()`,
– ajouter des méthodes `clone` dans chaque classe.

Dans `Book`, nous ferions :

```
@Override public Book clone() {
   return new Book(name, price, pages);
}
```

Dans `Box`, cela donne :

```
@Override public Box clone() {
   Box b = new Box(getName());
   for (Item i : items) b.addItem(i.clone());
   return b;
}
```

Nous rappelons qu'une redéfinition d'une méthode dans une classe (ou interface) dérivée peut utiliser une signature avec un type de retour plus précis. Par exemple, `clone` dans `Box` retourne un `Box` et non un `Item`. La méthode peut également utiliser une visibilité plus permissive. En fait, `public Item clone()` raffine déjà `protected Object clone()` qui est définie dans `Object`.

Question 6. 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. Discutez les avantages et les inconvénients d'incorporer les méthodes `addItem`, `removeItem`, `getItems` dans le type `Item`.

Contrairement à la méthode `clone`, nous implantons `printAllItems` en dehors des classes `Book`, `Box`, etc. Nous ne pouvons donc pas profiter de la liaison dynamique pour exécuter un code différent en fonction de

la classe de l'article considéré. Il est nécessaire d'utiliser des tests dynamiques de type `instanceof` et des conversions :

```
public static void printAllItems(Item i) {
   System.out.println(i.getName());
   if (i instanceof IBox) {
      IBox b = (IBox) i;
      for (Item j : b.getItems()) printAllItems(j);
   }
}
```

Notons que le test `instanceof` est vis-à-vis d'une interface (`IBox`). Cela permet de rendre le code client compatible avec toutes les implantations de composite, et pas seulement vis-à-vis de `Box`.

Le code ci-dessus peut être simplifié en considérant que tous les articles implantent `getItems()`. Il faut :

- ajouter `getItems()` dans l'interface `Item` ;
- implanter `getItems()` dans les classes feuilles ; ces objets ne contenant pas d'autre objet, la méthode retourne une liste vide ; par exemple, nous ajouterions dans `Book` et `Movie` :

```
public Collection<Item> getItems() { return Collections.emptyList(); }
```

- le test `instanceof` et la conversion peuvent simplement être supprimés de `printAllItems`, puisque `Item` possède maintenant la méthode `getItems`.

Remarquons l'utilisation de la méthode `emptyList`. Elle retourne une liste vide. Mais, contrairement à `new LinkedList<>()`, la liste est immuable et n'est allouée qu'une seule fois (ce qui est plus efficace).

Nous pouvons aller plus loin et unifier complètement `Item` et `IBox` en ajoutant aussi `addItem` et `removeItem` dans `Item`. L'avantage est encore de pouvoir implanter des fonctionnalités sans avoir à se soucier de savoir si l'objet implante `Item` ou `IBox`. Cependant, il est plus difficile de définir un comportement cohérent à donner à `addItem` et `removeItem` pour les articles, comme `Book`, qui ne contiennent jamais de sous-objet. Nous pouvons décider de ne rien faire, ou bien de retourner une exception.

Question 7. Étant donné un article, nous souhaitons savoir s'il est contenu dans un lot et, si c'est le cas, retrouver le lot le contenant. Pour cela, nous proposons d'ajouter une méthode `getParent()` qui retourne le lot le contenant, ou `null`. Montrez comment implanter cette méthode. Est-il possible qu'un article appartienne à plusieurs lots ?

- `getParent` doit exister dans tous les articles, la méthode est donc ajoutée dans l'interface `Item` ;
- la méthode `getParent` et un attribut `IBox parent` la supportant doivent exister dans toutes les implantations de `Item` ; il vaut donc mieux factoriser cette implantation, par exemple dans une classe abstraite dont toutes nos implantations dérivent ;
- les méthodes `addItem(item)` et `removeItem(item)` d'un lot doivent pouvoir modifier l'attribut `parent` de `item` ; il faut donc ajouter un `setParent` dans `Item` ;
- il existe un problème de visibilité : `setParent` devient accessible aussi au client, ce qui n'est pas souhaitable ; malheureusement, il n'y a pas de solution à ce problème tant que nous utilisons des interfaces car elles imposent une visibilité `public` à toutes les méthodes ; nous pourrions résoudre le problème avec de l'héritage de classe et la visibilité par défaut, package, en supposant que le client est dans un package différent.

Dans notre solution, nous gardons une interface avec `setParent`. Comme `addItem` et `removeItem` doivent être redéfinis, il est utile d'interdire la modification de la liste d'articles via la collection retournée par `getItems`

(voir la question 4, en retournant une copie dans `getItems`).

```java
public interface Item {
   String getName();
   int getPrice();
   IBox getParent();
   void setParent(IBox parent);
}
```

```java
public abstract class AbstractItem implements Item {
   private IBox parent = null;
   @Override public IBox getParent() { return parent; }
   @Override public void setParent(IBox parent) { this.parent = parent; }
}
```

```java
public class Box extends AbstractItem implements IBox {
   ...
   @Override public void addItem(Item item) {
      item.setParent(this);
      items.add(item);
   }

   @Override public void removeItem(Item item) {
      item.setParent(null);
      items.remove(item);
   }
}
```

Il faut ajouter à `Book` un `extends AbstractItem`.
Pas de changement à `IBox`.

Nous imposons qu'un article appartienne à au plus un lot, car `getParent` retourne un seul lot ou `null`. Supporter le partage d'un article entre plusieurs lots est possible, mais nécessiterait de maintenir une liste de parents (non fait ici).

6.3 Intégration d'un composant tiers (Adapter)

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

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

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

Question 8. Nous souhaitons ajouter ces objets comme articles dans notre catalogue. Il n'est 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 ces deux solutions et discutez leurs avantages et leurs inconvénients.

Solution par héritage :

```java
public class SofaItem extends Sofa implements Item {
   public SofaItem(String model, int price) { super(model, price); }
   public String getName() { return model(); }
   public int getPrice() { return price(); }
}

public class ConvertibleItem extends Convertible implements Item {
   public ConvertibleItem(String model, int price) { super(model,price); }
   public String getName() { return model(); }
   public int getPrice() { return price(); }
}
```

Inconvénients :

- il faut créer une classe pour chaque classe de la bibliothèque tiers (`SofaItem` n'est pas réutilisable pour `Convertible`, pas d'héritage multiple) ;
- `SofaItem` expose des méthodes de la bibliothèque tiers que nous pourrions vouloir cacher (`model`, `open`, etc.)

Solution par délégation, nous utilisons le Design Pattern Adapter:

```java
public class SofaAdapter implements Item {
   private Sofa sofa;
   public SofaAdapter(Sofa sofa) { this.sofa = sofa; }
   public String getName() { return sofa.model(); }
   public int getPrice() { return sofa.price(); }
   public Sofa getSofa() { return sofa; }
}
```

Avantages :

- la classe est réutilisable pour `Convertible`, et plus généralement toute classe dérivant de `Sofa` ;
- une interface peut aussi être décorée ;
- contrôle fin sur les méthodes exportées de la bibliothèque tiers ;
- le lien d'héritage n'est pas mobilisé ; il est donc possible de faire hériter `SofaAdapter` d'une implantation, par exemple `AbstractItem` vu en question 7, pour réutiliser du code.

Inconvénient : si nous passons un `Convertible` au constructeur de `SofaAdapter`, alors `getSofa` retournera un `Sofa`, i.e., nous perdons l'information de type. Il faut tester le résultat avec `instanceof` et faire une conversion pour retrouver l'objet à son type d'origine.

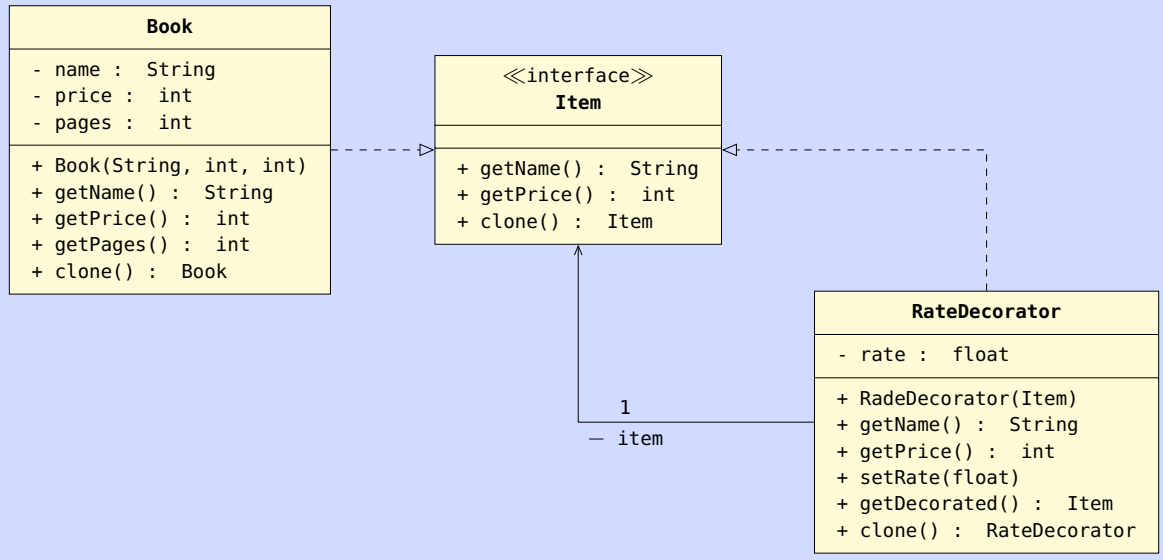
6.4 Ajout de fonctionnalité (Decorator)

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 9. 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 %.

Remarques :

- ci-dessous, nous commençons par donner un décorateur simplifié, sans passage par un décorateur abstrait ;
- attention à la conversion flottant vers entier, dans `getPrice`, qui doit être explicite ;
- nous pouvons également demander l'implantation de `clone` (donnée ci-dessous) ;
- j'inclus une méthode `getDecorated`, toujours utile pour retrouver le décoré ;
- le même décorateur peut ajouter notre fonctionnalité `setRate` à toute classe implantant `Item` ; avec l'héritage, il faudrait dupliquer chaque classe existante ;
- il y a toutefois un inconvénient : l'objet réduit le décoré à l'interface `Item` ; nous perdons toutes les méthodes supplémentaires ; celles-ci redeviennent accessibles via un `getDecorated` suivi d'une conversion d'objet (il y a donc les mêmes avantages et inconvénients que l'adapteur de l'exercice précédent).




```java
public class RateDecorator implements Item {
   private Item item;
   private float rate = 1;

   public RateDecorator(Item item) { this.item = item; }
```

```
   @Override public String getName() { return item.getName(); }

   @Override public int getPrice() { return (int) (item.getPrice() * rate); }

   public void setRate(float rate) { this.rate = rate; }

   @Override public RateDecorator clone() {
      RateDecorator r = new RateDecorator(item.clone());
      r.setRate(rate);
      return r;
   }

   public Item getDecorated() { return item; }
}
```

Exemple :

```
   Book b = new Book("La disparition", 10, 999);
   RateDecorator r = new RateDecorator(b);
   r.setRate(0.5);
```

Nous pouvons aussi penser à une construction un peu plus lourde avec :

- un décorateur abstrait `AbstractItemDecorator` qui se contente de déléguer au décoré sans rien changer ;
- un décorateur `RateDecorator` qui hérite de `AbstractItemDecorator` et ne redéfinit que `getPrice`.

L'avantage de cette technique apparaît pour une classe avec de nombreuses méthodes, et des décorateurs qui ne changent qu'une ou deux méthodes : nous factorisons ainsi le code de délégation directe (sans modification) au décoré. Rappelons aussi que les IDE comme Eclipse savent générer automatiquement ces méthodes de délégation.

Note : `AbstractItemDecorator` n'est pas fait pour être instancié directement, c'est donc une classe `abstract`, elle a un constructeur de visibilité package, et pas d'implantation de `clone`.

```
public abstract class AbstractItemDecorator implements Item {
   private Item item;

   AbstractItemDecorator(Item item) { this.item = item; }

   @Override public String getName() { return item.getName(); }

   @Override public int getPrice() { return item.getPrice(); }

   public Item getDecorated() { return item; }

   abstract public AbstractItemDecorator clone();
}
```

```java
public class RateDecorator extends AbstractItemDecorator {
   private float rate = 1;

   public RateDecorator(Item item) { super(item); }

   @Override public int getPrice() { return (int) (super.getPrice() * rate); }

   public void setRate(float rate) { this.rate = rate; }

   @Override public RateDecorator clone() {
      RateDecorator r = new RateDecorator(getDecorated());
      r.setRate(rate);
      return r;
   }
}
```

Question 10. Supposons que nous définissons dans la classe `Book` une méthode `toString` de la manière suivante :

```java
@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 ?

Reprenons le livre avec une promotion de 50% de la question précédente :

```java
Book b = new Book("La disparition", 10, 999);
RateDecorator r = new RateDecorator(b);
r.setRate(0.5);
```

`System.out.println(r)` va appeler la méthode `toString` de `RateDecorator`. Pour l'instant, cette méthode n'est pas définie, donc c'est celle qui est héritée d'`Object` qui sera utilisée. Ce n'est pas ce qui est attendu (elle affiche le type et l'adresse mémoire de l'objet). Nous souhaiterions profiter de la méthode `toString` redéfinie par `Book`. Il est pour cela nécessaire de définir `toString` dans `RateDecorator` (ou dans `AbstractItemDecorator`).

Une idée naturelle est de déléguer, dans `RateDecorator`, à l'objet décoré :

```java
public String toString() {
  return getDecorated().toString();
}
```

Cependant, nous aurons l'affichage suivant :

```
La disparition 10
```

qui indique le prix original, sans tenir compte de la promotion. En effet, `getDecorated().toString()` appelle `b.toString()`, qui appelle donc `b.getPrice()`, et pas `r.getPrice()`.

Une solution serait de recopier le code de `toString` depuis `Book` dans `RateDecorator` (ou `AbstractItemDecorator`) :

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

ainsi, `r.toString()` appellera `r.getPrice()`. Pour éviter la recopie de code, `toString` peut être définie une fois pour toutes dans une classe abstraite dont tout le monde hérite (comme `AbstractItem`).

Notons que, dans ce cas, `toString` de `RateDecorator` ne délègue plus au `toString` de l'objet décoré. Donc, par exemple, si `Book` redéfinit `toString`, alors `r.toString()` continuera à appeler le code de `RateDecorator` et jamais celui de `Book`. Nous perdons le bénéfice de la liaison dynamique. Nous voyons ici les limites du décorateur.

La dernière solution pour obtenir un comportement de `r.toString` qui dépend du type de l'objet décoré serait de faire un test explicite, avec `instanceof`, dans le `toString` de `RateDecorator`. Gérer tous les cas possibles dans `RateDecorator` est cependant très lourd et difficilement maintenable : il faudrait modifier `RateDecorator` à chaque ajout d'une nouvelle classe implantant `Item`...