TD-TP : Le Design pattern « Décorateur »

COMPRENDRE LE DESIGN PATTERN « Décorateur »

Le patron de conception Décorateur permet d'ajouter de nouvelles fonctionnalités à une classe de façon dynamique sans impacter les classes qui l'utilisent où en héritent. Un décorateur permet donc d'attacher dynamiquement de nouvelles responsabilités à un objet. Les décorateurs offrent une alternative assez souple à l'héritage pour composer de nouvelles fonctionnalités.

```
// Programme qui illustre la mise en œuvre du Patron de Conception Décorateur
public class TesteDécoration {

  // METHODE PRINCIPALE : main ()
  public static void main(String[] args) {

        // Création d'un composantDeBase
        IComposant composantDeBase = new ComposantDeBase();
        composantDeBase.operation();   // Fait l'opération de base
        System.out.println();

        // Création d'un autreComposant de base...
        IComposant autreComposant = new ComposantDeBase();
        // ... que l'on dote du comportement de décoration du décorateur A ...
        autreComposant = new DecorateurA_DeComposant (autreComposant);
        autreComposant.operation(); // Fait l'opération de base décorée par A
        System.out.println();

        // ... et on le transforme en le dotant EN PLUS des décorations du décorateur B
        autreComposant = new DecorateurB_DeComposant (autreComposant);
        autreComposant.operation();// Fait l'opération de base décorée par A et B
  }
}
```

Affichera :

```
Opération de base
```

puis une fois décoré par A :

```
# Prédécoration A #
Opération de base
# Postdécoration A #
```

puis une fois décoré en + par B :

```
% Prédécoration B %
# Prédécoration A #
Opération de base
# Postdécoration A #
% Postdécoration B %
```

```java
// Définit l'Interface d'objets qui seront décorés
public interface IComposant {
   // CONSTRUCTEUR        -non-
   // METHODES A IMPLEMENTER : opération
   public void opération(); // abstraite par définition
}
```

```java
// Implémente l'Interface pour les objets décorables
public class ComposantDeBase implements IComposant {

   // METHODE A IMPLEMENTER : opération
   public void opération () {
      System.out.println("Opération de base");
   }
}
```

```java
// Définit une Abstraction pour décorer les méthodes
public abstract class DécorateurDeComposant
                                 implements IComposant {
   //ATTRIBUT composantDécoré, le composant à décorer
   public IComposant composantDécoré;

   // CONSTRUCTEUR
   public DécorateurDeComposant (IComposant composant) {
      setComposantDécoré (composant);
   }
   // METHODES D'ENCAPSULATION : composantDécoré
   public void setComposantDécoré (IComposant composant) {
      this.composantDécoré = composant;
   }
   public IComposant getComposantDécoré () {
      return this.composantDécoré;
   }
}
```

```java
// Implémente une décoration pour méthodes définies dans l'Interface
public class DécorateurA_DeComposant extends DécorateurDeComposant {
      // CONSTRUCTEUR
      public DécorateurA_DeComposant (IComposant composant) {
            super (composant);
      }

      // METHODE A DECORER AVANT et/ou APRES : opération
      public void opération () {
            System.out.println("# Prédécoration A #"); // Déco avant
            composantDécoré.opération();
            System.out.println("# Postdécoration A #");// Déco après
      }
}
```

```java
// Implémente une autre décoration pour les méthodes de l'Interface
public class DécorateurB_DeComposant extends DécorateurDeComposant {
      // CONSTRUCTEUR
      public DécorateurB_DeComposant (IComposant composant) {
            super (composant);
      }

      // METHODE A DECORER AVANT et/ou APRES : opération
      public void opération () {
            System.out.println("% Prédécoration B %"); // Déco avant
            composantDécoré.opération();
            System.out.println("% Postdécoration B %");// Déco après
      }
}
```

Travail à faire :

1. Etudier le code ci-dessus qui met en œuvre le patron de conception Décorateur, en déduire (a) diagramme de classe UML, (b) objets et résultat.
2. Créer un projet `Eclipse` intitulé `DécorateurDeComposants` dans lequel créer toutes les classes identifiées ci-dessus, et obtenir l'exécution attendue.

APPLIQUER LE DESIGN PATTERN « Décorateur »

Sur la base de ce schéma de solution, vous allez produire un exemple concret. Dans cet exemple, les composants sont des `Pizza`.

L'opération à décorer est la méthode `toString` qui dans son implémentation de base retourne la chaîne de caractères
« `Pizza de base : Tomate – Fromage` ».

Les deux décorateurs seront `DecorerAvecOlives` pour l'un et `DecorerAvecChampignons` pour l'autre.

Ils compléteront/décoreront la chaîne de caractère avec « `+ des olives` » pour l'un, et avec « `+ des champignons` » pour l'autre.

Travail à faire :

3. Sur la base du diagramme de classe établi en **1.** produire le diagramme de classe correspondant au cas pratique des `Pizza`.

4. Créer un projet Java sous `Eclipse`, intitulé `DécorateurPizza` dans lequel coder les classes du diagramme établi en **3.**
La classe principale `TestePizzaDécorée` créera `unePizza` dont il demandera sa composition `toString`, puis enrichira cette pizza avec des olives et des champignons grâce aux décorateurs, ce que vous vérifierez en sollicitant l'opération `toString` qui aura été décorée (spécialisée) par chacun des deux décorateurs.