Architecture Logicielle & Design Patterns

TD/TP Design Pattern - Decorator

Exercice 1 — Boissons personnalisables

Une application de café en ligne permet de commander des boissons comme du **café**, du **thé**, ou du **chocolat chaud**.
Chaque boisson peut être **personnalisée** avec des suppléments : lait, sucre, chantilly, caramel, etc.

Actuellement, chaque combinaison (ex. *Café au lait sucré avec chantilly*) est implémentée comme une **nouvelle sous-classe**, ce qui entraîne une explosion du nombre de classes.

🎯 Objectif :

Utiliser le **Design Pattern Decorator** pour permettre d'ajouter dynamiquement des options à une boisson sans créer de nouvelles sous-classes.

⚙️ Tâches :

1. Créez une interface `Beverage` avec les méthodes :
    - `String getDescription()`
    - `double getCost()`
2. Implémentez une classe de base `Coffee` (ou `Tea`, `Chocolate`).
3. Créez une classe abstraite `BeverageDecorator` qui implémente `Beverage` et contient une référence vers une autre boisson.
4. Implémentez des décorateurs concrets :
    - `MilkDecorator`
    - `SugarDecorator`
    - `WhippedCreamDecorator`
5. Dans le code client, composez plusieurs décorateurs pour créer une boisson sur mesure.

🧠 Résultat attendu :

La description et le prix final d'une boisson sont construits dynamiquement selon les décorateurs utilisés.
Exemple :

```
Boisson : Café + Lait + Sucre → 3.20 €
```


Exercice 2 — Système de logging flexible

Un système d'application utilise actuellement une classe `Logger` qui écrit tous les messages dans un fichier texte.
On souhaite maintenant **étendre les fonctionnalités** du logger sans modifier la classe existante :

- Ajouter une **sortie console**,
- Ajouter un envoi des logs vers un **serveur distant**,
- Et éventuellement les **chiffrer** avant envoi.

Modifier la classe `Logger` à chaque nouvelle fonctionnalité la rendrait complexe et fragile.

🎯 Objectif :

Appliquer le **Design Pattern Decorator** pour permettre d'empiler plusieurs traitements sur les logs (console, fichier, serveur, chiffrement, etc.) de manière flexible.

⚙️ Tâches :

1. Créez une interface `Logger` avec une méthode `log(String message)`.
2. Implémentez une classe `FileLogger` (écrit dans un fichier).
3. Créez une classe abstraite `LoggerDecorator` contenant un `Logger` interne.
4. Implémentez plusieurs décorateurs :
    - `ConsoleLoggerDecorator`
    - `RemoteLoggerDecorator`
    - `EncryptedLoggerDecorator`
5. Composez plusieurs décorateurs dans le code client pour empiler les comportements.

🧠 Résultat attendu :

Un seul appel `logger.log("Erreur critique")` peut :

- afficher le message dans la console,
- l'écrire dans un fichier,
- et l'envoyer au serveur.

🧩 Exercice 3 — Éditeur d'images avec filtres dynamiques (niveau avancé)

Vous développez un **éditeur d'images** qui permet d'appliquer des **filtres visuels** (contraste, flou, luminosité, noir et blanc, etc.).
Initialement, chaque filtre était implémenté dans une sous-classe de `ImageEditor`, rendant le système lourd et rigide.

On souhaite maintenant pouvoir **enchaîner librement plusieurs filtres**, dans **n'importe quel ordre**, sans dupliquer du code ni modifier la classe de base.

🎯 Objectif :

Utiliser le **Design Pattern Decorator** pour appliquer plusieurs effets à une image de manière flexible et dynamique.

⚙️ Tâches :

1. Créez une interface `Image` avec une méthode `apply()`.
2. Implémentez une classe `BaseImage` représentant l'image d'origine.
3. Créez une classe abstraite `ImageDecorator` qui implémente `Image` et contient une référence vers une autre image.
4. Implémentez plusieurs décorateurs :
    - `BrightnessDecorator`
    - `BlurDecorator`
    - `ContrastDecorator`
    - `BlackAndWhiteDecorator`
5. Dans le code client, composez plusieurs décorateurs pour simuler un pipeline de filtres.

🧠 Résultat attendu :

L'image passe à travers chaque décorateur successivement :

```
Image originale → + contraste → + luminosité → + flou → résultat final
```

💬 Question bonus :

- Que faudrait-il modifier pour ajouter un nouveau filtre "bordure" ?
- En quoi cette architecture respecte le principe **Open/Closed** ?