Decorator and Adapter Patterns

1 Partie UML

Au cours de cet exercice, nous allons construire un diagramme de classe incrémentalement. Chacune des questions enrichira donc celui-ci. Pour chaque classe, indiquer ses attributs (privés), ses opérations (incluant la redéfinition de la méthode `toString()`), et son ou ses constructeurs.

On souhaite gérer un banc de test pour étudier les performances de divers composants de voitures. Une voiture est constituée d'un châssis, d'un ou plusieurs moteurs, d'un ou plusieurs types de freins,...

Question 0 • On imagine que les voitures peuvent être montées avec toute combinaison de composants comme des moteurs à essence, des moteurs diesel, des moteurs électriques, des freins à disque, et toutes sortes de composants de voiture à l'infini...
A quoi pourrait ressembler un premier diagramme de classes modélisant toutes ces voitures ?

Question 1 • Définissez l'interface **Voiture**, comprenant des accesseurs pour une accélération (`float`), un freinage (`float`), une masse (`float`) et pour un prix (`float`).

Question 2 • Ajoutez au diagramme de classes une implémentation `<<vide>>` de cette interface nommée **Chassis** (sans oublier les attributs).

Comme l'objectif du banc est de tester les diverses combinaisons d'options, vous allez utiliser le pattern *Décorateur* et monter la voiture comme un mécano.

1.1 Le décorateur

Question 3 • Complétez le diagramme en définissant une classe abstraite **VoitureMontee**, qui applique le pattern en enveloppant une voiture.

Question 4 • Définissez ensuite les classes **VoitureMoteurEssence** et **VoitureMoteurDiesel** qui modifient les méthodes `getAcceleration()`, `getFreinage()`, `getMasse()` et `getPrix()`.

Question 5 • Donnez le code Java de la méthode `getMasse()` de **VoitureMoteurEssence** en supposant que le moteur pèse 300 Kg.

Question 6 • De manière similaire, ajoutez les classes **VoitureFreinsDisque** et **VoitureFreinsFoucault** qui modifieront la force de freinage. Détaillez le code des différentes méthodes.
Vous pouvez constater maintenant que, afin de respecter le principe DRY, votre code peut être "refactorer" en en extrayant une partie vers la classe (abstraite) de base.

Question 7 • Donnez le code du constructeur de la classe **VoitureMontee**, puis celui permettant de construire une voiture hybride avec un moteur essence, un moteur diesel et des freins à disque.

1.2 L'adaptateur

Le **BancDeTest** remplit une fiche de renseignements pour chaque voiture, et sa méthode `lancerTests()` retourne la liste des fiches remplies.

Question 8 • Ajoutez au diagramme une classe **Fiche** contenant des données telles que la voiture concernée, sa vitesse maximale ou sa distance de freinage. On ne notera pas les accesseurs/mutateurs par souci d'économie.

On aimerait maintenant trier notre liste de fiches selon divers critères, choisis à l'exécution. En java, la méthode statique `Collections.sort` permet le tri d'une liste, à condition que les éléments soient `Comparable`.

Question 9 • Ajoutez au diagramme UML une classe **TriFicheVitesse** qui servira d'*adaptateur* entre une **Fiche** et l'interface `Comparable <TriFicheVitesse>`. Elle devra en particulier définir la méthode `compareTo(TriFicheVitesse autre)` qui retourne un entier -1, 0 ou +1 selon l'ordre des fiches comparées.

On considère maintenant des chassis fabriqués par un consortium européen que l'on doit pouvoir monter dans toute voiture et tester sur notre banc d'essai. La classe **ChassisEuropeen** est fournie, elle possède exactement les mêmes méthodes que l'interface Voiture. Cependant, cette classe étant fournie sous sa version compilée, il n'est absolument pas possible de la modifier.

Question 10 • Comment peut-on utiliser ce chassis dans une voiture ? Donnez le diagramme de classes, puis détaillez le code.

On considère ensuite des **ChassisAnglais** fabriqués au Royaume Uni, et pour lesquels une classe (ici encore non modifiable) est fournie, mais dont les identificateurs de méthodes sont en anglais `getMass`, `getBraking`, `getPrice`...

Question 11 • Proposez une solution permettant d'utiliser ce chassis dans une voiture, puis détaillez le code.

2 Partie Java

L'objectif de cette partie est d'implanter et de tester les différents diagrammes que vous avez réalisés dans la première partie du TD. N'attendez pas d'avoir tout écrit pour tester !

Dans votre dépôt local, vous trouverez l'interface `Voiture`, qui est un peu plus riche que celle vue dans la partie UML, et les classes `Chassis`, `FreinDisque`, `FreinFoucault`, `MoteurDiesel`, `MoteurEssence`, `Fiche` et `DynamiqueVoiture`. Cette dernière s'occupe de tous les calculs du banc de test.

Question 12 • Ajoutez la classe abstraite `VoitureMontee` qui implémente l'interface `Voiture`.

Question 13 • Connectez les sous-classes `FreinDisque`, `FreinFoucault`, `VoitureMoteurDiesel`, `VoitureMoteurEssence` à la super-classe `VoitureMontee` (héritage). Sachant que la masse, la puissance et le coefficient de freinage sont cumulables, n'oubliez pas de redéfinir les méthodes au niveau des sous-classes.

Question 14 • Dans la fonction `main` de la classe `BancDeTest`, construisez et ajoutez une voiture dotée d'un châssis, d'un moteur, et d'un système de freins. Ajoutez la voiture au banc de test et lancez les tests

Question 15 • Décorer maintenant avec différent composants vos voitures et admirez la force du pattern.

Question 16 • Ajoutez votre adaptateur `TriFicheVitesse`.

Vous pouvez ainsi trier votre liste en utilisant :

```
Collections.sort(resultats, (ficheResultat, autre) ->
    ficheResultat.compareAcceleration(autre));
```

Ou encore, avec une référence de méthode (Java 8+) :

```
Collections.sort(resultats, FicheResultat::compareAcceleration);
```