TP4 : Transports

2024-2025

But : Modéliser l'activité d'une compagnie de transport de véhicules et leurs passagers par ferry. Le mode de calcul du tarif de transport d'un véhicule dépend de ses caractéristiques propres, le tarif de transport d'une personne est fixé à 15 €.

Diagramme de classes

Vous ferez un diagramme de classes des différentes entités du TP ; ce diagramme devra être complété et modifié au fur et à mesure de l'évolution du TP.

1 Version initiale

Créer toutes les classes dans un premier paquetage **v1**.

1.1 Modélisation des véhicules

Les fonctionnalités d'un véhicule sont définies par la spécification suivante :

```java
public interface IVehicule {
    // determiner la longueur d'un vehicule
    public int getLongueur();

    // determiner le nombre de passagers
    public int getPassagers();

    // connaitre l'immatriculation
    public String getImmatriculation();

    // representation affichable
    public String toString();

    // calculer le tarif du vehicule
    public float calculerTarif();
}
```

Caractéristiques de quelques véhicules

Important : Pensez à programmer les différentes classes de véhicules en donnant la possibilité d'étendre la hiérarchie.

Auto
- longueur égale à 2 unités ; nombre de personnes quelconque ;
- tarif = 350 € s'il s'agit d'un véhicule tout-terrain, 100 € sinon, à quoi s'ajoute le tarif de transport des passagers.

Programmez la classe Auto.

Bus
- longueur quelconque ; nombre de personnes quelconque ;
- tarif = 200 € + 50 € par unité de longueur, à quoi s'ajoute le tarif de transport des passagers.

Programmez la classe Bus.

1.2 Modélisation du ferry

Voici les caractéristiques d'un ferry :
- un ferry a une capacité limitée, aussi bien en unités de longueur qu'en nombre de passagers transportés ;
- on doit pouvoir placer dans un ferry un véhicule de n'importe quelle catégorie.

Les véhicules transportés dans un ferry seront placés dans l'une des structures de données de la bibliothèque java.

On veut doter la classe Ferry des méthodes suivantes :
- méthode **ajouter** qui ajoute un véhicule dans un ferry si c'est possible ; cette méthode renvoie un booléen pour indiquer si l'ajout a pu se faire ;
- méthode **calculerTarif** qui calcule le tarif total de transport des véhicules présents dans le ferry ;

- méthode **toString** qui permet l'affichage du contenu du ferry : détail de chaque véhicule, longueur disponible, nombre de places disponibles, tarif total des véhicules transportés.

NB : pensez à utiliser **StringBuilder**.

Programmez la classe Ferry en veillant à permettre aisément le remplacement d'une structure de données par une autre.

1.3 La compagnie de transport

Programmez dans une autre classe la fonction main ; celle-ci doit :

- créer un Ferry ;
- répéter les opérations ci-dessous jusqu'à ce que le ferry soit plein, de préférence sans intervention humaine ; à cet égard, la classe Random pourra vous être utile.
  - créer différentes sortes de véhicules ;
  - ajouter chacun d'eux dans le ferry ;
  - afficher le contenu détaillé du ferry.

Changez la structure de données utilisée pour les véhicules, recompilez et testez.

1.4 Clonage polymorphe

Problématique :

- De façon très générale, quand on place des instances dans une structure de données, on effectue en réalité une copie de référence. Les instances utilisées par la structure sont donc les mêmes que celles de l'entité qui les a créées puis placées dans la structure : les instances sont donc partagées par plusieurs entités distinctes.
  C'est le cas ici :
  - la compagnie de transport crée les instances ;
  - le ferry stocke dans une structure de données des références sur les instances de la compagnie de transport.

  Les instances de véhicules sont donc partagées entre la compagnie de transport et le ferry.
- Dans certaines situations, il peut être souhaitable que les instances ne soient pas partagées entre plusieurs entités, mais que chaque entité gère des instances indépendantes : c'est le cas par exemple, si l'on souhaite que les modifications d'instances faites par l'une des entités soient sans effet pour l'autre entité.
- Il faut donc fournir un mécanisme qui permet de créer des copies d'instances : la deuxième entité va donc créer puis gérer des copies d'instances.

Réalisation : pour créer une copie d'une instance, il faut programmer dans la hiérarchie de véhicules une méthode de clonage (méthode **clone()**) et imposer que toutes les classes concrètes de véhicules implémentent l'interface Cloneable.

Faites les modifications nécessaires dans la hiérarchie de véhicules et la classe Ferry pour que cette classe gère des copies des véhicules qui y sont ajoutés.

1.5 Trier les véhicules

On veut pouvoir trier les éléments du ferry en ordre croissant ou décroissant. Dans la classe **java.util.Collections**, on trouve un grand nombre de fonctions statiques qui permettent de réaliser différents traitement sur des collections. En particulier, on trouve la fonction suivante :

```
1 static <T extends Comparable<T>< void sort(Collection<T> list);
```

Cette fonction permet de trier en ordre croissant une liste "selon l'ordre naturel des éléments".

1.5.1 Premier tri

Ajoutez dans la classe Ferry une méthode trier() qui trie le contenu du ferry à l'aide de la fonction sort ci-dessus. Identifiez le problème qui se pose à la compilation, puis sa solution. Programmez les fonctionnalités nécessaires, sachant qu'on peut définir "l'ordre naturel des éléments" à partir de leur longueur.
Complétez la fonction main pour tester cette fonctionnalité.

1.5.2 Autres tris

On veut donner à la compagnie de transport la possibilité d'utiliser d'autres critères de tri et aussi de choisir le sens de tri (croissant/décroissant), par exemple :

- tri par longueur décroissante ;
- tri par tarif croissant ou décroissant ;
- tri selon plusieurs critères : par longueur croissante et en cas d'égalité par nombre de passagers croissant ;
- etc...

On remarque qu'il existe dans la classe **java.util.Collections** une autre fonction de tri :

```
static <T> void sort(Collection<T> col, Comparator<T> c);
```

On rappelle l'interface **Comparator<T>** vue en TD :

```
public interface Comparator<T> {
    public int compare(T o1, To2);
}
```

1) tri par tarif croissant Programmez la classe **ComparateurTarif** qui implémente l'interface **Comparator<T>** et compare deux véhicules en fonction de leurs tarifs respectifs. Ajoutez une méthode trier() dans la classe Ferry qui prend en paramètre un comparateur et effectue le tri à l'aide de ce comparateur. Complétez la fonction main pour tester cette fonctionnalité.

2) choix du sens de tri On veut donner à la compagnie le choix du sens de tri : ordre croissant ou décroissant. Complétez la classe **ComparateurTarif** pour offrir cette nouvelle possibilité.
Remarque : ce changement ne doit avoir aucune incidence sur la classe Ferry.
Complétez la fonction main pour tester cette fonctionnalité.

3) tri par longueur décroissante Programmez ce qui est nécessaire pour offrir cette nouvelle fonctionnalité ; complétez la fonction main pour la tester ; même remarque.

4) tri multi-critères Programmez la classe **ComparateurMulti** ; cette classe doit offrir la possibilité de combiner deux comparateurs quelconques.
Remarque : ce changement ne doit avoir aucune incidence sur la classe Ferry.
Complétez la fonction main pour tester cette fonctionnalité.

5) Comment est-il possible, sans programmer de nouveau comparateur ni modifier la classe Ferry, de trier les véhicules selon plus de deux critères ? Par exemple tri par longueur croissante, puis par tarif décroissant puis par numéro d'immatriculation croissant ? Faites le nécessaire...

Bonus : Essayez d'utiliser une lambda-expression à la place d'un comparateur.

1.6 Autres véhicules

1) Ambulance : Une ambulance est une auto, dont le tarif de transport est toujours nul, qu'il s'agisse d'un véhicule tout terrain ou non. Programmez la classe Ambulance et complétez la fonction main pour incorporer ce nouveau véhicule.

2) Cycle : Un cycle ne peut transporter qu'une personne ; sa longueur est d'une unité et son tarif de 20 € à quoi s'ajoute le tarif passager. Programmez la classe Cycle ; même remarque.

1.6.1 Diagramme de classes

Mettez à jour le diagramme des différentes entités utilisées jusqu'ici.

2 Mutualisation

Préambule : copiez tous les fichiers dans un nouveau paquetage **v2**.

2.1 Mutualiser les fonctionnalités des véhicules

Modifiez l'organisation des classes de véhicules (sans oublier le diagramme de classes) de façon à mutualiser le maximum d'attributs et de fonctionnalités des différents véhicules programmés puis testez.

Remarques :
- seules les classes de véhicules doivent être modifiées par cette nouvelle organisation ;
- les attributs qui ne sont pas des constantes doivent rester privés.

2.2 Comparateurs : un patron de conception

En regardant les différents comparateurs, on s'aperçoit qu'on a plusieurs classes très similaires :
- attribut pour mémoriser le sens du tri ;
- méthode compare dont l'algorithme (très simple) est le même pour toutes les classes ; seul change le critère de comparaison.

Une première amélioration consiste à programmer une classe mère commune qui mémorise cet attribut. Une deuxième amélioration consiste à programmer la logique de la méthode **compare()** dans la classe mère ; celle-ci fera appel à une méthode (par exemple **docompare**), qui sera déclarée **abstraite** et **protected** dans la classe mère, pour comparer les caractéristiques des véhicules ; les classes filles n'ont plus qu'à implémenter cette méthode **docompare** (toujours **protected**) qui compare deux véhicules sans se soucier de l'ordre croissant/-décroissant.
Cette façon de concevoir la méthode **compare()** suit un patron de conception nommé **patron de méthode** : celui-ci consiste à définir la structure d'un algorithme dans une classe de base (éventuellement abstraite), à l'aide d'opérations dont certaines peuvent être abstraites et qui devront être implémentées par les classe filles.
Modifiez l'organisation de vos classes de comparateurs (sans oublier le diagramme de classes) pour programmer la méthode **compare()** selon un patron de méthode.

Remarque : seules les classes des comparateurs doivent être modifiées par cette nouvelle organisation.

3 Calcul du tarif de transport

Préambule : copiez tous les fichiers dans un nouveau paquetage **v3**.
Il n'est pas cohérent que les véhicules soient chargés de calculer eux-mêmes leur tarif de transport. Cette responsabilité incombe plutôt à la compagnie de transport.
En outre, on veut disposer d'un moyen simple de modifier les tarifs. Voici la nouvelle organisation à mettre en place.

3.1 Calculer le tarif

1) supprimez la méthode **calculerTarif** dans la hiérarchie de véhicules ;

2) créez une classe abstraite Tarif qui va mémoriser les éléments qui entrent dans le calcul du tarif d'un véhicule :
- tarif passager
- tarif fixe par véhicule
- tarif variable par véhicule

Remarque : les attributs qui ne sont pas des constantes doivent rester privés. Dans cette classe,
- programmez un constructeur **protected** pour initialiser les attributs ;
- programmez les **accesseurs** et **mutateurs** ;
- déclarez une méthode abstraite de calcul du tarif qui prend un véhicule en paramètre.

3) créez une classe fille pour chaque classe concrète de véhicule ; programmez dans cette classe la méthode de calcul du tarif.

Remarque importante : pour garantir qu'une modification de tarif d'une classe de véhicules soit répercutée sur toutes les instances de cette classe, il ne devra être possible que de créer une unique instance de chaque classe de tarif. C'est exactement le rôle du patron de conception "Singleton".

Le patron de conception Singleton

Dans chaque classe concrète de tarif :
- déclarez un attribut statique destiné à mémoriser l'unique instance de cette classe ; cet attribut sera initialisé à **null**, soit dans la déclaration, soit dans un bloc statique ;
- programmez un constructeur **protected** ;
- programmez la méthode statique **createSingleton** qui prend les mêmes paramètres que le constructeur ; cette méthode se charge de créer et mémoriser une instance unique de la classe puis renvoie l'instance ainsi créée ;
- programmez la méthode statique sans paramètre getInstance qui renvoie l'unique instance de la classe.

La seule façon de créer une instance de tarif est donc de faire appel à la méthode **createSingleton** de la classe de tarif souhaitée ; après la création, la seule façon d'obtenir une instance de cette classe est de faire appel à la méthode **getInstance**.

3.2 Associer un véhicule et son tarif

La compagnie de transport procède ainsi :
- création d'une (unique) instance de tarif pour chaque classe de véhicule ;
- création des instances de véhicules comme auparavant ;
- association entre chaque instance de véhicule et l'instance de tarif correspondante ; cette association sera réalisée à l'aide d'une table de correspondance (**HashMap**) dont la clé sera le numéro d'immatriculation du véhicule (unique !) et la valeur associée sera l'instance de Tarif correspondant à ce véhicule.

Lorsqu'on voudra calculer le tarif d'un véhicule, il faudra :
- chercher l'instance de tarif associée au véhicule dans la table ;
- appeler la méthode de calcul de tarif de cette instance.

Remarque : l'accès à la table des tarifs ne pourra se faire que dans la classe qui gère la compagnie de transport.

Question : Quelle propriété doit vérifier la clé d'une table de type **HashMap** ? Faites toutes les modifications nécessaires.

Modification de tarif

Effectuez ensuite deux changements de tarifs :
- augmentation du tarif passager de 10%
- doublement du tarif tout terrain

Après chaque modification, faites afficher le contenu du ferry.