TP - Agence de location

Ce sujet est composé de deux parties, la première permet de manipuler les collections et les tables de hachage et la seconde consiste en une première mise en œuvre simple de l'héritage.

Ce travail sera à mettre dans un dossier nommé `tp7/` de votre dépôt Git.

Récupérez l'archive sur le portail pour en utiliser les codes source et les tests fournis.

On s'inspire du sujet du TD sur les agences de location de voiture en y apportant les modifications et extensions suivantes :

- on remplace la classe `Car` par la classe `Vehicle` dont voici le diagramme UML

rental::Vehicle
- brand : String - model : String - productionYear : int - dailyRentalPrice : float
+Vehicle(brand : String, model : String, productionYear : int, dailyRentalPrice : float) + getBrand() : String + getModel(): String + getProductionYear() : int + getDailyRentalPrice() : float + equals(o : Object) : boolean + toString() : String

- l'interface `CarFilter` devient `VehicleFilter`. Elle est adaptée pour manipuler des objets `Vehicle` et non plus des objets `Car` ;
- la classe de l'exception `UnknownCarException` devient `UnknownVehicleException`.

Dans chacun des cas, il s'agit d'un simple renommage, le reste est inchangé.

Collections et Tables

`RentalAgency`

Voici des évolutions apportées au cahier des charges du TD sur les agences de location.

- les méthodes `addVehicle` et `removeVehicle` remplacent `addCar` et `removeCar`, et ont été adpaptées pour prendre en compte des véhicules ;
- la méthode `select` de la classe `RentalAgency` devient :
  ```
  public List<Vehicle> select(VehicleFilter filter)
  ```
  dont le résultat est la liste des véhicules qui sont acceptés par le filtre `filter` passé en paramètre.

On ajoute à la classe `RentalAgency` la gestion des locations des véhicules par des clients. Un client ne peut louer qu'un véhicule à la fois.
Pour représenter les clients, on utilisera la classe `Client` fournie. Dans cette classe, les clients sont modélisés par un attribut représentant l'âge et un autre correspondant à leur nom qui sera une chaîne de caractères. On supposera que les noms sont uniques, et donc qu'il n'y a pas d'homonyme :

Client
- age : int - name : String
+ Client(name : String, age : int) + getName() : String + getAge() : int + equals(o : Object) : boolean + hashCode() : int

Les méthodes `equals` et `hashCode` fournies permettent l'utilisation des objets `Client` comme clefs d'une table de hachage.

On décide de **gérer les locations des véhicules par une table** (de type `java.util.Map` et une instance `java.util.HashMap`) qui associe les clients (clés) avec le véhicule (valeur) loué par le client. Un client n'est présent dans cette table que s'il est en train de louer un véhicule. Il en est donc « supprimé » dès qu'il rend un véhicule.

Vous trouverez en annexe un rappel des principales méthodes de l'interface `Map<K,V>`. Etudiez les car vous devez les utiliser pour coder les méthodes demandées ci-dessous.

On complète la classe `RentalAgency` avec les méthodes suivantes :

- `public boolean hasRentedAVehicle(Client client)` renvoie `true` si et seulement si `client` est un client qui loue actuellement un véhicule et donc `false` sinon.
- `public boolean isRented(Vehicle v)` renvoie `true` si et seulement si le véhicule est actuellement loué, `false` sinon.
- `public float rentVehicle(Client client, Vehicle v)`
  `throws UnknownVehicleException, IllegalStateException`

  permet au client `client` de louer le véhicule `v`. Le résultat est le prix de location.

  Cette méthode **ne doit pas** modifier le contenu de l'attribut `theVehicles` de l'agence : les véhicules gérés par l'agence sont toujours les mêmes.

  L'exception `UnknownVehicleException` est levée si le véhicule `v` n'existe pas dans l'agence.

  L'exception `IllegalStateException` est lancée dans deux situations :

  – si le véhicule `v` est déjà loué par un client,
  – si le client `c` est déjà loueur d'un autre véhicule.
- `public void returnVehicle(Client client)` : le client `client` rend le véhicule qu'il a loué. Il ne se passe rien si ce client n'avait pas loué de véhicule.
- `public Collection<Vehicle> allRentedVehicles()` renvoie la collection des véhicules de l'agence qui sont actuellement loués.
- une méthode `displaySelection` qui prend en paramètre un filtre pour les véhicules et affiche les véhicules acceptés par ce filtre.

  Vous devez bien sûr réutiliser la méthode `select`.

Le code de la classe `UnknownVehicleException` est fournie. Consultez en le code.

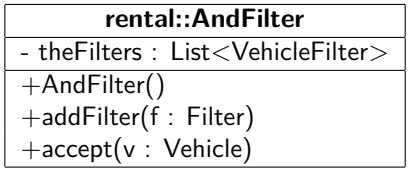
Q 1 . Complétez le code de la classe `RentalAgency` fournie en tenant compte du cahier des charges mentionnés ci-dessus.

N'oubliez pas les tests. Lorsque vous complèterez la classe `RentalAgencyTest` fournie, vous utiliserez l'anotation `@Before` dont vous trouverez un exemple d'utilisation dans les classes `VehicleTest` et `ClientTest` fournies.

`@Before` est présentée dans le document sur les tests présent dans la zone Documents du portail[1].

`AndFilter`

Dans ce TP, le filtre « intersection », `AndFilter`, est un peu différent de celui de la question 10 du TD.
On souhaite en effet que ce filtre permette de regrouper un **nombre quelconque** de filtres (et plus seulement deux comme dans le TD). Ces filtres sont mémorisés dans une liste.
A la construction d'un `AndFilter`, il ne contient aucun filtre et il est donc vérifié. Les filtres qui le composent sont ajoutés via la méthode `addFilter`. Le diagramme UML de la classe `AndFilter` est le suivant :



Q 2 . Complétez le code la classe `AndFilter` fournie pour permettre de réaliser l'intersection d'un nombre quelconque d'objets `VehicleFilter`.

Vous devez bien sûr fournir des tests pour cette classe.

[1] Voir https://www.fil.univ-lille1.fr/ routier/enseignement/licence/poo/tdtp/tests-complement.pdf.

Q 3 . Comme dans le TD, créer une classe `MainAgency` qui définira une méthode `main` grâce à laquelle vous effectuerez quelques expérimentations en créant quelques objets véhicules que vous ajouterez à une agence et en affichant les résultats de sélections par des filtres (dont au moins un `AndFilter`).

Créez aussi des clients et faites leur louer et rendre des véhicules et écrivez du code qui utilise les différentes méthodes de `RentalAgency` que vous avez définies.

Héritage

Cette seconde partie va être l'occasion de mettre en œuvre l'héritage dans des situations simples. On va créer des sous-classes de `Vehicle` et une sous-classes de `RentalAgency`.

Q 4 . Créez une classe `Car` qui hérite de `Vehicle`. Une voiture a comme propriété additionnelle le nombre de passagers qu'elle peut accueillir. La classe `Car` dispose de l'accesseur associé.

La méthode `toString` de cette classe reprend les mêmes informations que `Vehicle` complétées du nombre de passagers (il faut gérer cela au mieux).

Q 5 . Créez une classe `Motorbike` qui hérite de `Vehicle`. Une moto a comme propriété additionnelle la cylindrée (exprimée en cm^3). La classe `Motorbike` dispose de l'accesseur associé et sa méthode `toString` reprend les mêmes informations de `Vehicle` complétées par cette cylindrée.

Q 6 . Complétez la méthode `main` précédente avec l'ajout d'objets `Car` et `Motorbike` dans l'agence et faites une sélection des véhicules sur un prix maximum. Vous afficherez le résultat de cette sélection.

Q 7 . Certaines agences appliquent une sur–tarification pour des clients qu'elles considèrent comme des « jeunes conducteurs ».

Créez une classe `SuspiciousRentalAgency` qui hérite de `RentalAgency` et qui applique un surcoût de 10% sur le prix de location pour les clients dont l'âge est inférieur à 25.

Comment gérer au mieux ce surcoût au niveau du code ? En particulier, on souhaite que si le calcul du coût de location de `RentalAgency` est modifié alors le surcoût de 10% reste valable.

Q 8 . Complétez la méthode `main` précédente en définissant un objet `SuspiciousRentalAgency` et en vérifiant son bon comportement pour différents clients, en particulier l'application du surcoût.

Annexe : `Map<K,V>`

Voir les documents de cours https://www.fil.univ-lille1.fr/ routier/enseignement/licence/poo/cours/collections-impression.pdf, diapo 18 à 22

On rappelle ci-dessous les principales méthodes de l'interface `Map<K,V>`.
Vous utiliserez des objets de la classe `HashMap<K,V>`.

`V get(K key)` récupère la valeur associée dans la table à la clé `key`

`void put(K key, V value)` ajoute une entrée (un couple) (clé, valeur) à la table (si il existe déjà une entrée pour cette clé, elle est remplacée)

`V remove(Object key)` supprime de la table le couple associé à la clé `key`

`boolean containsKey(Object key)` teste (avec `equals` et `hashCode`) l'existence dans la table d'une entrée associée à la clé `key`

`boolean containsValue(Object value)` teste (avec `equals`) l'existence dans la table d'une entrée avec la valeur `value`

`Collection<V> values()` renvoie la **collection** des valeurs présentes dans la table.

`Set<K> keySet()` renvoie l'**ensemble** des clés présentes dans la table.