TP - Agence de location

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 (toujours disponible sur le portail) en y apportant les modifications et extensions suivantes :

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

Vehicle
- brand : String - model : String - productionYear : int - dailyrentalPrice : int
+Vehicle(brand : String, model : String, productionYear : int, dailyrentalPrice : int) + getBrand() : String + getModel() + getProductionYear() + getDailyRentalprice() + equals(o : Object) : boolean + toString()

- l'interface `Criterion` est adaptée pour gérer des objets `Vehicle` et non plus des objets `Car`.
- la méthode `select` de la classe `RentalAgency` devient :

  `public List<Vehicle> select(Criterion crit)`

  dont le résultat est la liste des véhicules qui satisfont le critère `crit` passé en paramètre.
- ajoutez une méthode `displaySelection` qui prend en paramètre un critère et affiche les véhicules que ce critère sélectionne. Vous réutiliserez bien sûr la méthode `select`.
- Le critère intersection de la question du TD peut maintenant regrouper un **nombre quelconque** de critères (et plus seulement deux) qui sont ajoutés via la méthode `addCriterion`. Son diagramme UML est le suivant :

InterCriterion
- theCriteria : List<Criterion>
+InterCriterion() +addCriterion(c: Criterion) +isSatisfiedBy(v : Vehicle)

- Comme dans le TD, créer un `main` grâce auquel vous effectuerez quelques expérimentations en créant quelques objets véhicules et en affichant les résultats de sélections.
- On ajoute à la classe `RentalAgency` la gestion des locations des véhicules. Un client ne peut louer qu'un véhicule à la fois.

  On pourra utiliser et si nécessaire compléter la classe `Client` fournie où les clients sont simplement modélisés par un attribut correspondant à leur nom qui sera une chaîne de caractères. On supposera que les noms sont uniques, il n'y a pas d'homonyme[1].

  On décide de gérer ces locations par une table (`java.util.Map`) qui associe les clients (clés) avec le véhicule (valeur) qu'ils ont loué. Un client n'est présent dans cette table que si il est en train de louer un véhicule. Il en « ressort » donc dès qu'il rend un véhicule.

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

  - `public void addVehicle(Vehicle v)` permet d'ajouter une véhicule à l'agence
  - `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.

[1] Attention donc à ne pas confondre les objets clients et leurs noms !

- `public void returnVehicle(Client client)` : le client `client` rend le véhicule qu'il a loué. Il ne se passe rien si il n'avait pas loué de véhicule.
- `public Collection<Vehicle> allRentedVehicles()` renvoie la collection des véhicules de l'agence qui sont actuellement loués.

- dans `RentalAgency` `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.

  L'exception `UnknownVehicleException` est levée si le véhicule n'existe pas dans l'agence et `IllegalStateException` s'il est déjà loué ou que le client loue déjà un autre véhicule.

Héritage

Q 1 . Créez la classe `UnknownVehicleException` puis complétez le code des classes `RentalAgency` et `InterCriterion` fournies en tenant compte des compléments au cahier des charges mentionnés ci-dessus.

N'oubliez pas les tests !

Q 2 . On considère le code suivant (fourni dans la classe `MainQ2`) :

```
RentalAgency agency = new RentalAgency();
Vehicle v = new Vehicle(...);
agency.addVehicle(v);
Client c1 = new Client("Tim Oleon",25);
float price = agency.rentVehicle(c1,v);
Client c2 = new Client("Tim Oleon",25);
boolean b = agency.hasRentedAVehicle(c2);
```

1. Selon vous quelle valeur devrait-on avoir pour `b` à l'exécution de ce code ?
2. Expérimentez en exécutant le `main` de `MainQ2`, est-ce le résultat avec votre code ?
   Si non, pourquoi ?
   Faites les corrections nécessaires pour obtenir le résultat attendu (si vous ne trouvez pas pourquoi demandez à votre enseignant !).

Q 3 . Créez une classe `Car` qui hérite de `Vehicle`. Une voiture a comme propriété additionnelle le nombre de passagers qu'elle peut accueillir. Cette information est ajoutée à la méthode `toString` des objets `Car`. La classe `Car` dispose de l'accesseur associé.

Q 4 . 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.

Q 5 . Dans un `main` créez une agence à laquelle vous ajouterez à la fois des objets `Vehicle`, `Car` et `Motorbike` et faites une sélection sur un prix dont vous afficherez le résultat.

Q 6 . Créez une classe `SuspiciousRentalAgency` qui hérite de `Agency` et qui applique un surcoût de 10% sur le prix de location pour les conducteurs dont l'âge est inférieur à 25.

Comment gérez au mieux ce surcoût dans le code ?

Q 7 . Dans un `main` expérimentez le fonctionnement de cette agence.