Secteur Tertiaire Informatique

Filière « Etude et développement »

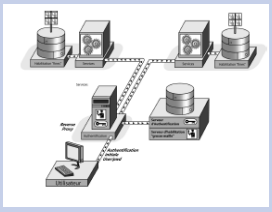
Entreprise EasyCar

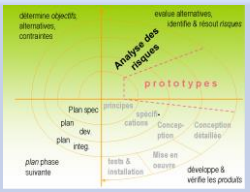
Programmation Orientée Objet

Apprentissage

Mise en situation

Evaluation





1. INTRODUCTION

L'objectif de ce projet est de concevoir un programme en console **basé sur une approche objet** et permettant de gérer **des locations de véhicules pour l'entreprise EasyCar.**

Ce cas se rapproche d'un cas concret de logiciel qui répond à un cahier des charge.

Le code attendu doit suivre les règles syntaxiques et les conventions de nommage du langage Java.

Il vous est demandé de **créer un nouveau projet basé sur un archétype Maven** existant.

2. CAHIER DES CHARGES

2.1 CONTEXTE

L'entreprise **EasyCar** vous contacte pour mettre en place une application permettant à des clients de louer des véhicules en ligne. Vous êtes en charge du développement des classes Java qui serviront, par la suite, à concevoir le code serveur.

2.2 REGLES METIER

L'entreprise **EasyCar** loue différents types de véhicules :

- Voitures ;
- Camions ;
- Vélo (non électriques).

Chaque véhicule a un prix de location à la journée et peut être loué pour une certaine période de temps.

Un véhicule est caractérisé par :

- Sa marque ;
- Son modèle ;
- Sa couleur ;
- Une date à laquelle l'entreprise EasyCar a fait son acquisition ;
- Un prix de location par jour.

Les véhicules motorisés (voitures, camion) sont caractérisés par, en plus :

- Un type de carburant (on considérera que le type « électrique » est également un type de carburant. Les types de carburant possibles sont les suivants : Essence, Diesel, électrique.
- Une consommation de carburant aux 100 km
- S'il y a un GPS intégré au véhicule ou non.

Les vélos sont caractérisés par une taille : S (petit), M (moyen), L (grand).

Les véhicules sont loués par des clients, un client est défini par les caractéristiques suivantes :

- un prénom ;
- un nom de famille ;
- une adresse postale ;
- une ville ;
- un code postal.

Tout client peut effectuer des réservations, les réservations sont définies par les caractéristiques suivantes :

- une date de début de réservation ;
- une date de fin de réservation ;
- une indication permettant de savoir si la facture liée à la réservation a été payée ou non.

Les fonctionnalités suivantes sont attendues :

- Possibilité d'ajouter une **réservation** à un **client** pour un véhicule spécifique (une réservation ne concerne qu'un seul véhicule et à lieu pour une période de temps spécifique) ;
- retrouver le prix total d'une réservation ;
- retrouver le total dépensé par un client ;
- gérer une liste de véhicules.

Attention

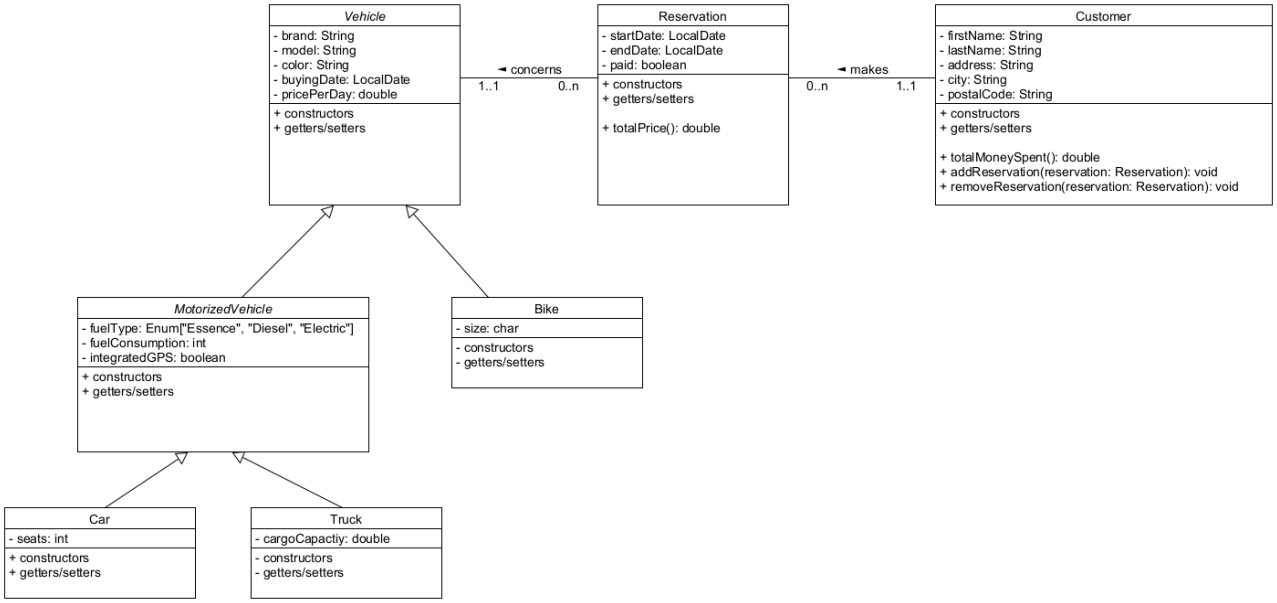
Dans un premier temps il n'est pas attendu de mettre en place une **interface utilisateur mais uniquement de mettre en place les classes permettant de répondre aux règles métier.**

Vous pourrez ajouter des saisies utilisateurs si le temps vous le permet.

3. CAHIER DE SPECIFICATIONS TECHNIQUE

3.1 MODELISATION UML

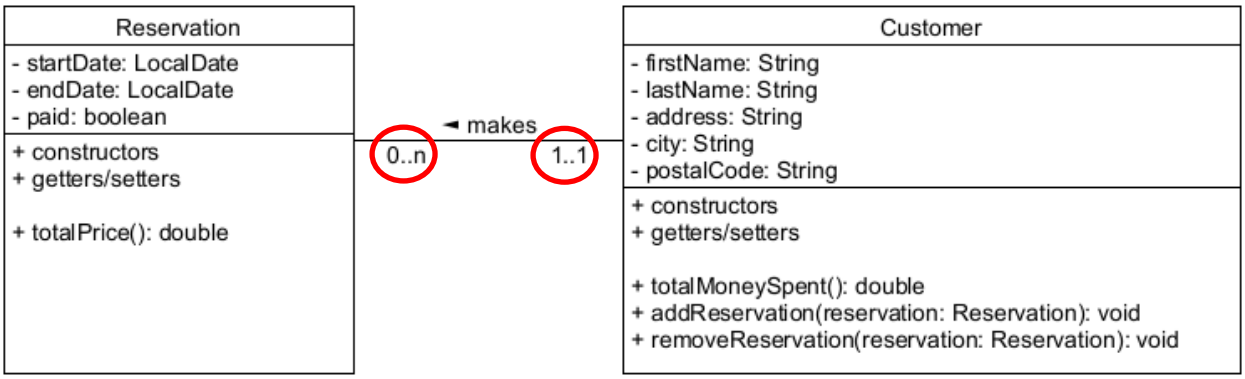
Afin d'implémenter les fonctionnalités attendues, l'« architecte technique » en charge du projet vous fournit le diagramme UML suivant :



Votre objectif est d'implémenter **toutes ces classes** et de **tester leur fonctionnement**.

Dans la suite du document vous seront donnés certains détails sur l'implémentation.

3.1.1 Relation entre « Customer » et « Reservation »



Nous observons une relation « **makes** » entre la classe « Customer » et la classe « Reservation ». Ceci indique qu'un client peut effectuer des réservations.

Notez les cardinalités (en rouge sur le diagramme), ceci s'interprète de la façon suivante :

- **0..n :** un client peut effectuer de 0 à un nombre indéterminé de réservations
- **1..1 :** une réservation est effectuée par un et un seul client

En code Java, cette relation peut être implémentée en ajoutant un attribut permettant de stocker une liste de réservations dans la classe « Customer ».

Par exemple en ajoutant l'attribut suivant :

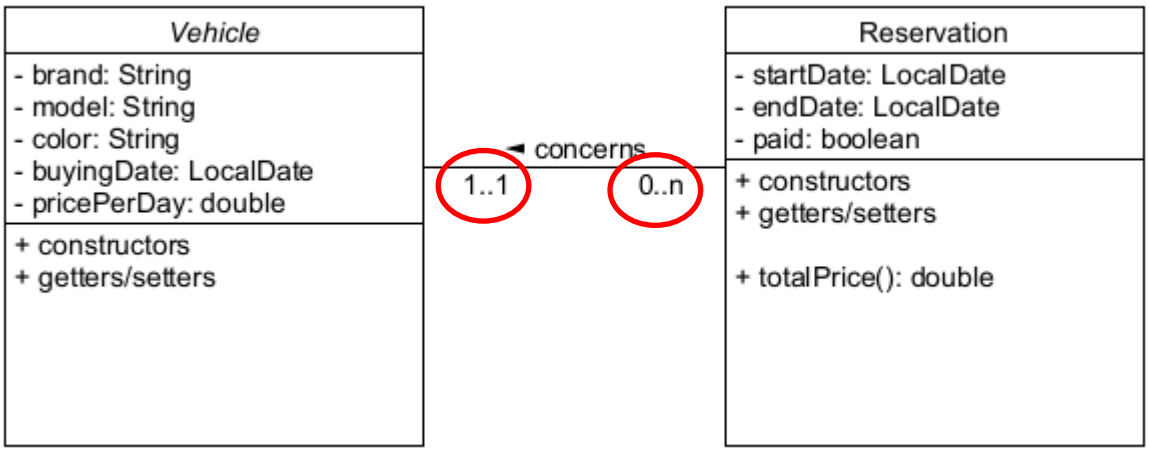
```
private ArrayList<Reservation> reservations = new ArrayList<Reservation>();
```

Voici maintenant quelques détails sur les méthodes :

- **totalMoneySpent() : double** : renvoie la somme totale de l'argent dépensé par l'utilisateur
- **addReservation(reservation : Reservation) : void** : ajoute une réservation au client
- **removeReservation(reservation : Reservation) : boolean**, supprime une reservation, renvoie **VRAI** si la réservation a pu être supprimée, renvoie **FAUX** sinon

Dans la classe réservation nous observons une méthode « **totalPrice() : double** ». Veuillez consulter la partie suivante pour en apprendre plus.

3.1.2 Relation entre « Reservation » et « Vehicle »



Nous observons une relation « **concerns** » entre la classe « Reservation » et la classe « Vehicle ». Ceci indique qu'une réservation concerne un véhicule.

Notez les cardinalités (en rouge sur le diagramme), ceci s'interprète de la façon suivante :

- **1..1 :** une réservation concerne **un et un seul véhicule**
- **0..n :** un véhicule peut être concerné **par 0 ou un nombre indéterminé de réservations**

En code Java, cette relation peut être implémentée en ajoutant un attribut permettant de cibler un véhicule dans la classe « Reservation », par exemple :

```
private Vehicle vehicle;
```

Si l'on souhaite retrouver toutes les réservation pour un véhicule, il est également **possible d'ajouter une liste dans la classe « Vehicle »** :

```
private ArrayList<Reservation> reservations = new ArrayList<Reservation>();
```

Détails sur la méthode « **totalPrice() : double** » : permet de retrouver le prix total d'une réservation pour la période de temps de la réservation.

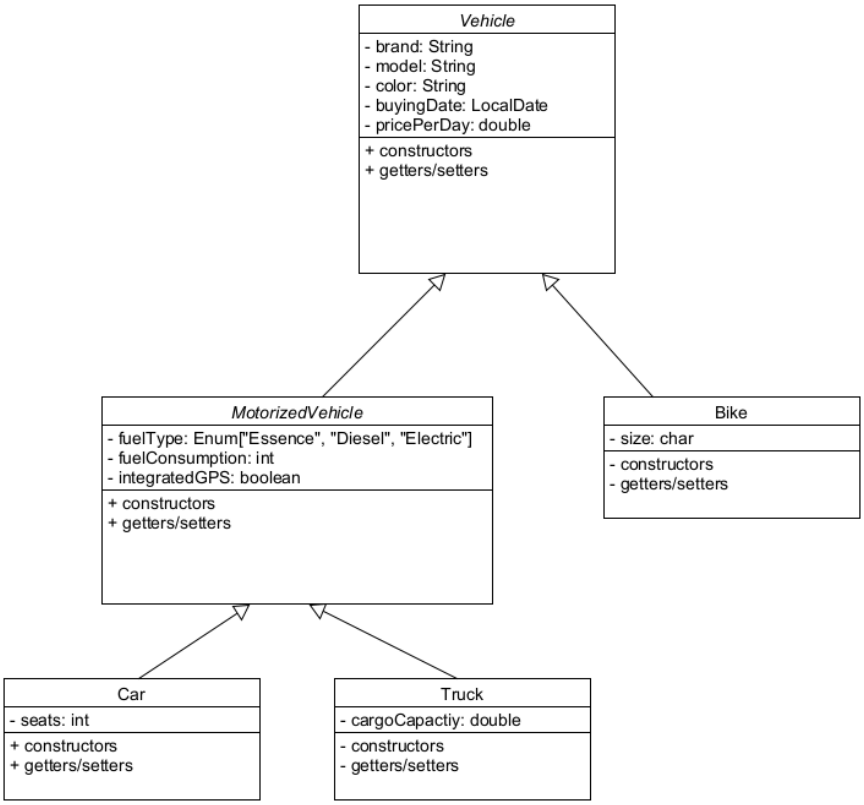
On observe qu'une réservation est notamment définie par 2 dates : « startDate » et « endDate ». De plus, nous avons la possibilité de retrouver le véhicule concerné par la réservation.

Ainsi, en calculant le nombre de jours séparant 2 dates il est possible de calculer le prix total.

Un indice sur la façon de calculer le nombre de jours entre deux dates : https://howtodoinjava.com/java/date-time/calculate-days-between-dates/

Pour un rappel sur le fonctionnement de la classe « **LocalDate** » : https://codegym.cc/fr/groups/posts/fr.1109.classe-java-localdate

3.1.3 Héritage des véhicules



Il vous est demandé d'implémenter l'arborescence ci-dessus. Notez le fait que les classes « **Vehicle** » et « **MotorizedVehicle** » soient abstraites.

Il y a une particularité sur ce **diagramme UML** : le type de l'attribut « **fuelType** » de la classe « **MotorizedVehicle** ».
Il s'agit d'un type « **enum** », lisez l'article suivant pour mieux comprendre le fonctionnement d'une enum et l'implémenter : https://codegym.cc/fr/groups/posts/fr.154.classe-enum-en-java

CREDITS

ŒUVRE COLLECTIVE DE l'AFPA
Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)
Michel Coulard – Formateur Evry
Chantal Perrachon – IF Neuilly sur Marne

Date de mise à jour : 01/07/2024