Object Oriented Software Design

MyUber Application

Rapport intermédiaire
de projet

Augustin Cuignet & Luca Thiébaud

Sommaire

Introduction

Ce projet est pour chacun des auteurs leur première grande utilisation du langage Java. Bien que fastidieux, ce projet était pour nous le moyen de nous immerger réellement dans la complexité du codage, de manière plus ample et en étant moins accompagnés qu'en Travaux Dirigés.

I - Première partie : le coeur de l'application
Analyse, Implémentation et Tests.

Cette première partie est celle sur laquelle se concentre ce rapport intermédiaire. Il s'agit alors de créer le coeur de l'application à venir. Pour ce faire, nous avons commencé par étudier le sujet en détail afin d'avoir un premier aperçu des principales classes et de leurs attributs, et de déterminer les patterns importants. Après avoir passé le sujet au peigne fin, nous avons construit un premier jet de diagramme UML. Bien que n'étant pas définitif, il nous a aidé à avoir un aperçu de notre structure future.Après avoir évolué au cours de la programmation, voilà ce qu'est devenu le diagramme UML du cœur de l'application :

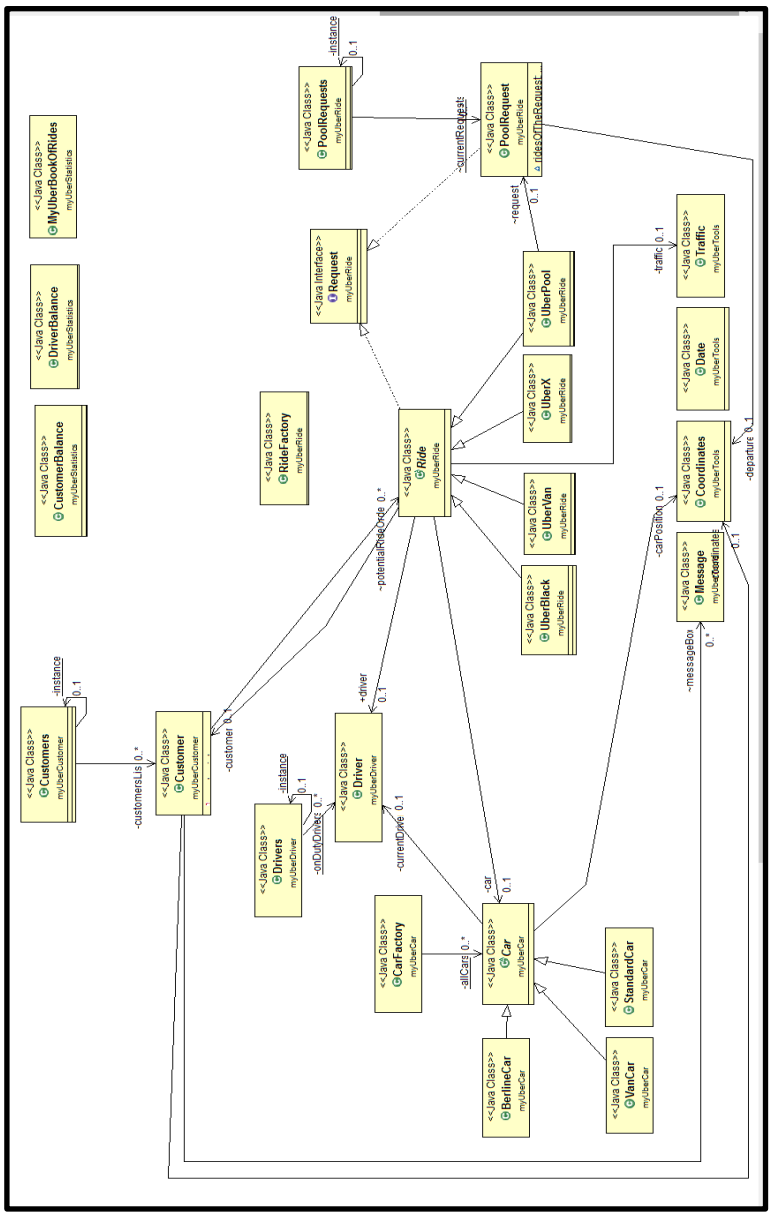
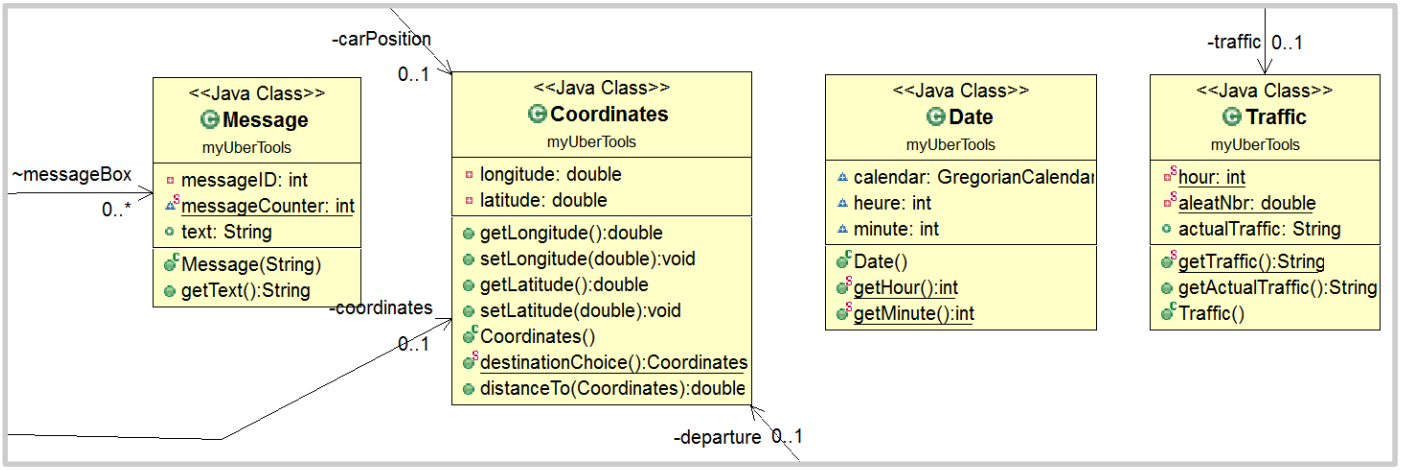


Diagramme UML du coeur de l'application

I - 1) Les classes d'outil

Pour commencer, nous avons réuni plusieurs "outils" dans un seul package : myUberTools. Nous avons Date, Traffic, Message et Coordinates.



Le package MyUberTools

Ces classes sont des outils auxiliaires d'attribution (Message, Coordinates) ou de détermination (Date, Traffic) de propriétés qui ne sont pas nécessairement uniques pour une classe.

- La classe Message sert à ajouter des messages dans la boîte de réception (messageBox) d'un Customer. On peut facilement ajouter une boîte de réception à la classe de conducteur (Driver) si on le désire.
- La classe Coordinates permet d'utiliser très facilement des coordonnées dans une requête de course (Ride ou PoolRequest). En théorie, c'est cette classe qu'il faudrait relier au GPS du téléphone pour que l'application soit utilisable en situation réelle. Ici, une coordonnée sera initiée au point (0,0) et nous demandons au consommateur les coordonnées GPS de sa destination.
- La classe Date utilise le calendrier Grégorien déjà implémenté dans eclipse afin de faciliter l'utilisation des dates dans le tri de statistiques et dans la génération de trafic.
- La classe Traffic utilise un nombre aléatoire et la date à l'instant t, associé aux probabilités données dans l'énoncé, pour simuler un type de trafic. Celui-ci sera pris en compte afin de déterminer le prix d'une Ride et sa durée.

I - 2) Packages MyUberCustomer et MyUberDriver

a) MyUberCustomer

Ce package est composé de deux classes. La première, Customers, permet de créer un unique univers de consommateurs (Customer). C'est ici que l'on peut gérer la liste de tous les Customer (ajout, suppression).
La seconde classe, Customer, représente un consommateur de l'application. Comme exigé, il a un identifiant unique (Singleton Pattern), un numéro de carte de paiement, un nom, surnom une boite de réception sous la forme d'ArrayList, ainsi que plusieurs attributs utiles pour le calcul de statistiques, des coordonnées GPS.
La commande d'une course débute dans cette classe, avec la méthode **comparePrices(Coordinates destination)**, qui va calculer le prix des différents types de Ride pour la destination entrée dans le GPS. Afin de ne pas privilégier un type de ride, le trafic à l'instant de commande leur est commun.
Après avoir pris connaissance des prix, le Customer peut choisir une des Ride proposées grâce à la méthode **selectRide(Ride chosenRide)** qui va transférer le traitement vers le package MyUberRide. Grâce à la méthode **cancelRide(Ride)**, le consommateur peut annuler sa course tant qu'aucun chauffeur ne l'a acceptée.

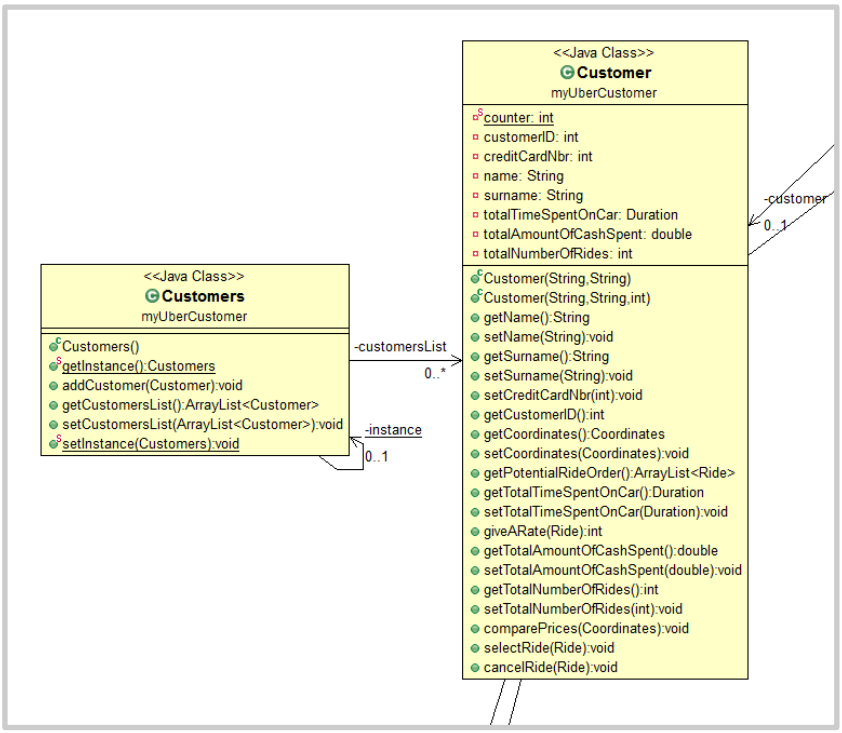


Diagramme du package MyUberCustomer

b) MyUberDriver

Ce package a le même forme que le précédent, avec une classe Drivers écrite avec un Singleton Pattern afin de créer un univers de Driver, et une classe Driver qui se rapporte à un seul conducteur. Celui-ci a tous les attributs requis pour le définir, notamment un état qui change en fonction de son activité. Un conducteur peut ainsi se connecter et se déconnecter, ce qui change son état. En se connectant, il doit s'associer à une voiture dont ill est propriétaire et il doit choisir le type de Ride qu'il veut effectuer (Black, Van, X, Pool). Si sa voiture ne le permet (Black avec une voiture Standard par exemple), un message d'erreur est affiché.
Quand une course lui est proposée, il peut l'accepter à travers la méthode **acceptRide(Ride)**, mais il peut aussi la décliner. A la fin de chaque course, ses statistiques et sa note sont mises à jour.

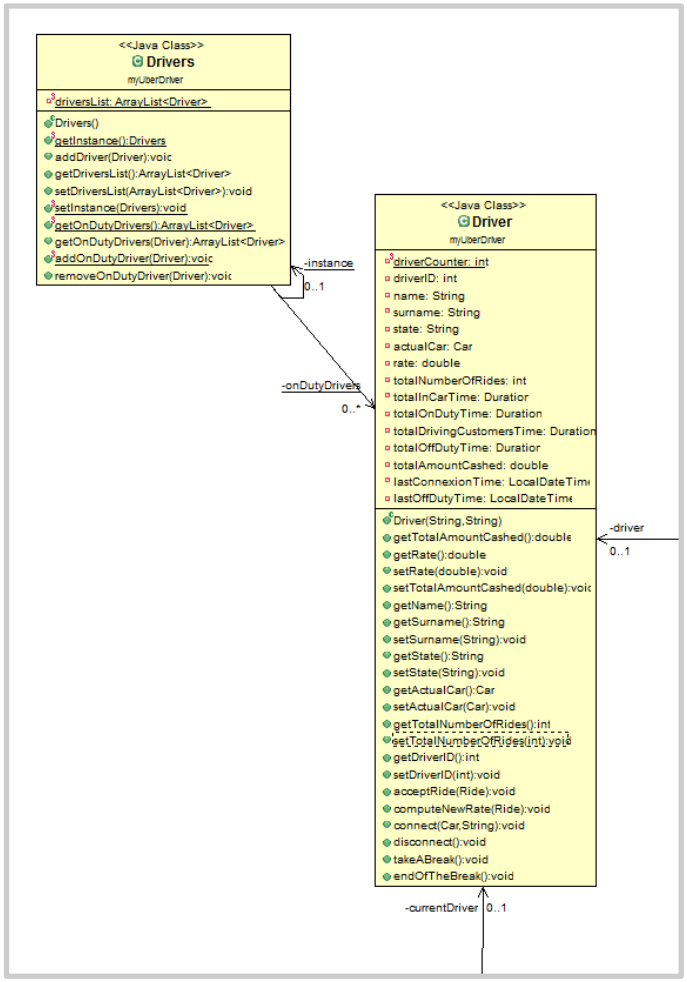


Diagramme du package MyUberDriver

I - 3) Package MyUberCar

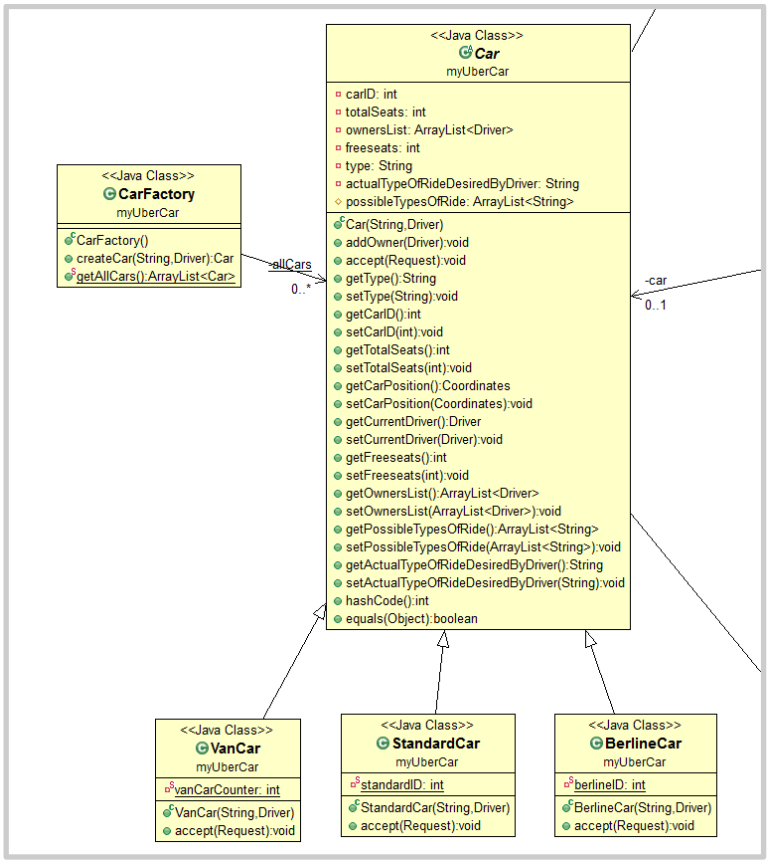


Diagramme du package MyUberCar

Afin d'utiliser et de pouvoir ajouter différents types de voitures dans l'application, nous avons utilisé un Factory Pattern. En effet, celui-ci permet à notre client d'instancier plusieurs types voitures facilement, en respectant l'*open-close principle*. Pour ce faire, nous avons implémenté une classe abstraite *car*, définissant tous les attributs d'une voiture de myUber, avec tous les *getters and setters* nécessaires puisque nous les avons déclarés en privé pour les protéger. Cette classe est étendue par trois sous-classes définissant chacune un type particulier de voiture : standard, van ou berline. Puis nous avons codé la *CarFactory*, permettant de créer une nouvelle voiture, selon le pattern classique de la factory. Notons qu'au début nous avons implémenté une abstract factory. Mais en réalité cela s'est avéré être superflu et nous avons préféré implémenter un factory pattern basique. Enfin il faut noter un élément important. La liste de toutes les voitures de myUber doit impérativement être unique. Nous avons donc eu recours à un singleton pattern dans la CarFactory.

I - 4) Package MyUberRide : le cœur de l'application

Ce package comporte les classes centrales de notre application.
Une Ride peut être de différents types : Black, Van, X, ou Pool. Nous avons défini les 3 premières comme étant "classiques" car elles sont toutes traitées exactement de la même manière tandis que les courses UberPool sont un peu différentes. Néanmoins, toutes les requêtes de course, qu'elles soient classiques ou Pool ont un point commun grâce à l'interface qu'elles implémentent : l'interface Request.

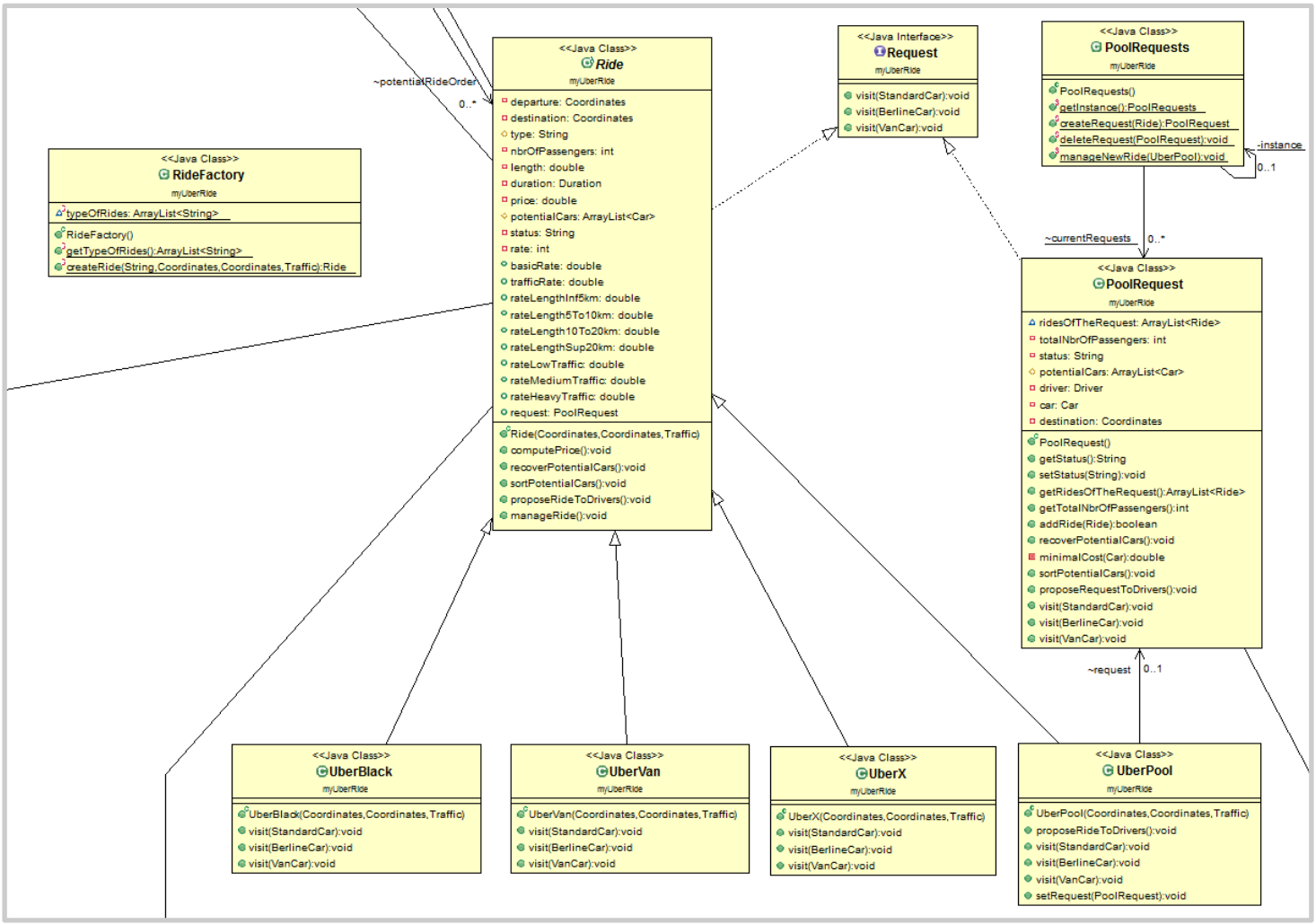


Diagramme du package MyUberRide

Nous avons choisi de définir la classe Ride comme une classe abstraite. Cela nous permet de nous assurer que nous ne pouvons pas l'instancier. En revanche, ses classes filles, concrètes, peuvent l'être. Afin de pouvoir être flexible sur les types de Ride à instancier, nous avons choisi d'utiliser un factory pattern. Le client peut ainsi facilement modifier les types de Ride proposés. Nous avons déclaré dans la classe Ride un identifiant qui lui est propre grâce à un singleton pattern, des attributs nécessaires en tant qu'informations basiques de la course (départ, arrivée, longueur, durée, trafic, état, nombre de passagers, prix), et tous les taux multiplicateurs du prix.

Comme ils dépendent du type de course choisi, ils sont réécrits dans les constructeur de chaque classe fille de Ride. Hormis les différents getters et setters, plusieurs méthodes de cette classe sont fondamentales.
computePrice() calcule le prix de la Ride en prenant en compte les bons indicateurs et paramètres de la course, ce qui permet au Customer de comparer les prix des différents types de Ride.

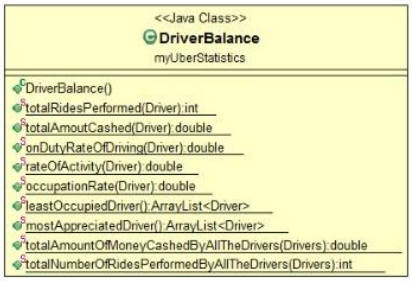
Voyons ce qu'il se passe quand le client choisit une course classique.
Une fois que le client a choisi la ride qu'il veut faire, **recoverPotentialCars()** recense toutes les voitures potentielles (driver "on-duty" dans une voiture compatible avec le choix du Customer) et les liste dans une ArrayList : potentialCars. Nous utilisons pour ce faire un visitor pattern. Le Visitor est la Ride et le Visitable est la Car.
sortPotentialCars() s'occupe alors de les trier, soit par ordre de plus voiture la plus proche du point de départ (pour les sous classes classiques), soit par ordre de coût le plus petit pour les UberPool, en utilisant @Override. Dans les deux cas, pour trier correctement les voitures, il n'aurait pas été judicieux d'utiliser un attribut propre à chaque voiture, puisque cet attribut changerait si deux courses étaient commandées en même temps à différents points de la ville. Ainsi, ces méthodes de tri sont certes coûteuses en calcul, mais comme elles augmentent en $O(n^2)$ avec n le nombre de voitures dans la liste, il suffit de limiter ce nombre pour limiter les calculs.
La course va ensuite être proposée aux Driver dans l'ordre de cette liste, grâce à la méthode **proposeRideToDrivers()**. Ceux-ci peuvent soit l'accepter, soit la refuser. Une fois qu'un conducteur l'accepte, elle n'est plus disponible aux autres conducteurs.
La méthode **manageRide()** se chrge alors du bon déroulé de la course. Finalement, le client peut noter son conducteur (méthode **GiveARate(Ride ride)** dans la classe Customer), qui verra sa note personnelle changer.

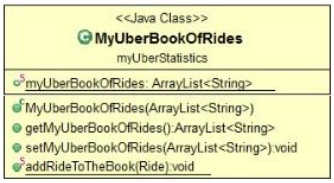
Si un client choisit une course UberPool, celle-ci est prise en charge par la méthode **manageNewRide()** dans la classe PoolRequests. Si la course peut être ajoutée à une PoolRequest existante, elle l'est, sinon une nouvelle PoolRequest est créée. Dès qu'une PoolRequest est pleine, soit par le nombre de courses (3) ou le nombre de personnes (que nous avons limité à 4, en nous disant qu'un van ne ferait pas ou peu de Pool), celle-ci est traitée de la même manière que précédemment : obtention de la liste des voitures potentielles, tri, et envoi aux chauffeurs. Notons que les Ride UberPool sont ensuite gérées indépendamment les unes des autres, ce qui permet à chaque utilisateur de noter son chauffeur, et d'avoir des statistiques qui lui sont propres.

Finalement, nous pouvons voir une Request comme la définition d'une course pour un chauffeur, et une Ride comme la définition d'une course pour un Customer.

I-5) Package MyUberStatistics

Le package MyUberStatistique est un outil destiné au calcul de statistiques par le client. Il est relativement indépendant du reste du code car il ne concerne pas directement le processus même de réservation de course, contrairement à tous les autres packages. Il se divise en trois classes : CustomerBalance, DriverBalance et MyUberBookOf Rides.





a) CustomerBalance

Cette classe est destinée au calcul de statistiques à partir des données concernant l'utilisateur. Elle se compose de deux grands types de méthodes : celles portant sur un client en particulier, et celles portant sur l'ensemble de l'univers des utilisateurs. Les premières, comme **totalAmountOfCashSpent()**, **totalTimeOnCar()**,... ont ainsi pour attribut un utilisateur en particulier, et renvoient la valeur actuelle de l'un des attributs de l'utilisateur à l'aide de *getters and setters*. Les secondes, comme **mostChargedCustomers()**, utilisent directement la liste de tous les utilisateurs. A chaque fois ces méthodes parcourent la liste en utilisant une méthode de tri classer les utilisateurs selon un critère particulier. A l'avenir, nous utiliserons la méthode **sort()** qui est bien plus rapide à implémenter.

b) DriverBalance

Cette classe est relativement similaire à *CustomerBalance*. Les méthodes se séparent elles aussi en deux parties, selon si elles prennent en argument un driver donné ou l'ensemble des drivers. Parmi les méthodes se focalisant sur un conducteur donné, notons celles du type **onDutyRateOfDriving()**,... Ces méthodes s'appuient sur des durées, qui sont des attributs des conducteurs et qui sont calculées à partir de chaque ride dans la classe *Ride.*

c) MyUberBookOfRide

Une des contraintes du cahier des charges du client était de tenir un historique de toutes les rides effectuées. Nous les avons stockées à l'aide de cette classe, *MyUberBookOfRide.* Nous avons commencé par stocker les données dans un fichier texte. Mais la recherche d'information dans le fichier étant laborieuse à implémenter,

nous avons finalement opté pour une autre solution : stocker les informations dans un ArrayList. Cette solution était beaucoup plus simple à coder. Pour protéger ces données, très précieuses pour le client, nous avons utilisé un *singleton pattern* pour ne générer qu'un *bookOfRides.* Nous avons également codé une méthode permettant d'ajouter une *ride* au *bookOfRide.* Nous appelons cette méthode dans la méthode **manageRide()** de la classe *Ride.*