Report

Antoine Paris & Yassine Abbahaddou

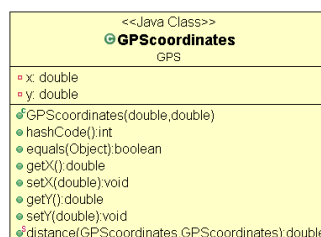# Introduction and first description of our solution

The aim of the myUber Project we did develop is to implement a java code to model all the interactions which are required for a system like Uber to work. It means all the following operations should be implemented.

A customer can enter the GPS coordinates of the place he wants to go and then can choose the type of ride he wants, knowing all the prices. Then the nearest driver should be notified for the ride and can accept or refuse it. If the driver refuses the ride, the system should notify other drivers until one accept it. If no driver can be found the ride is canceled. The customer can also cancel the ride if he did not get into the car. If a driver accepts a ride, he should pick up the customer and notify the myUber system that the ride has started. He should also notify the system when the ride is over, and the customer can then give a mark to the driver.
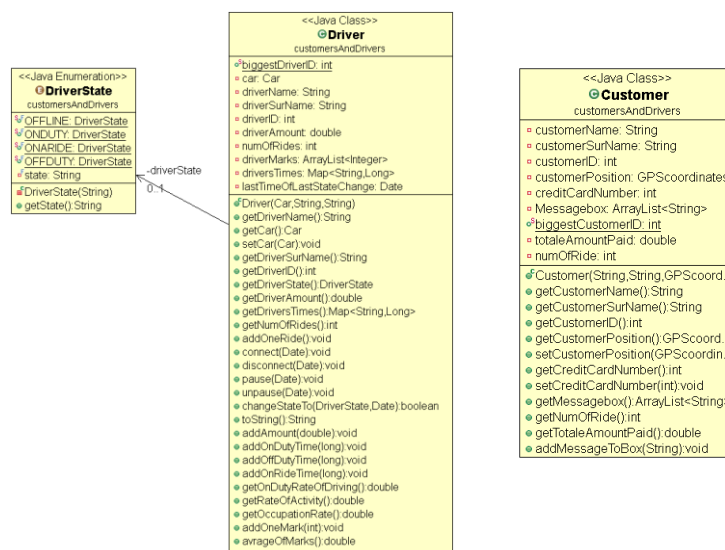
To perform all these operations, we must also do hidden operations to store the status of each, driver, car or ride, so that an operation which is not allowed cannot occur.

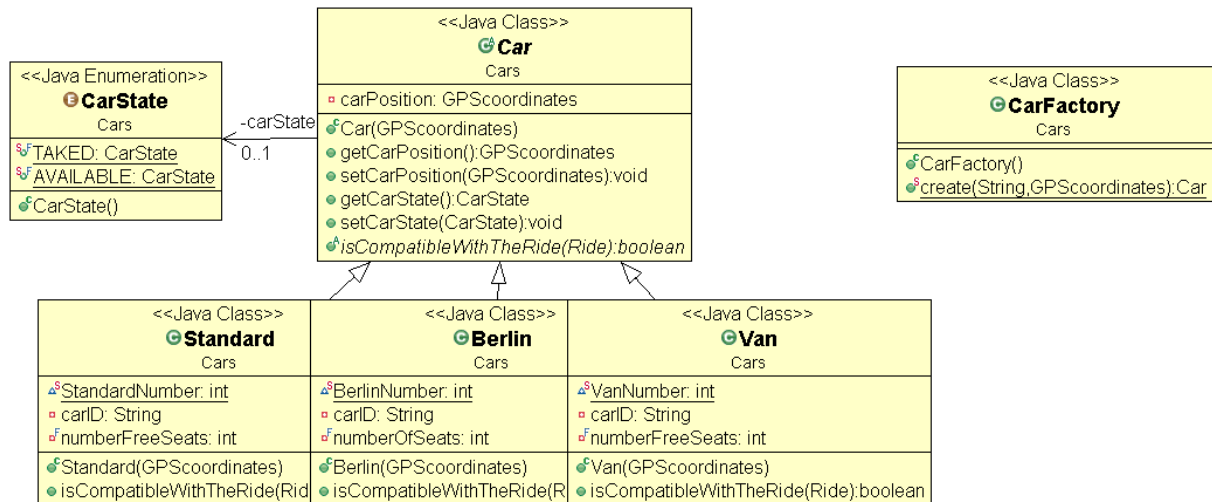To do so we choose to implement the following classes.

First of all, we choose to implement a GPScoordinates class which allowed us to create and store coordinates of customers, cars or arrival points of rides. We implemented also a method to compute the distance between two points, or a method to find the closest point in a list. We did also Override the hashCode and equals methods to define the equality between points.

```
<<Java Class>>
 GPScoordinates
        GPS

 x: double
 y: double

 GPScoordinates(double,double)
 hashCode():int
 equals(Object):boolean
 getX():double
 setX(double):void
 getY():double
 setY(double):void
 distance(GPScoordinates,GPScoordinates):double
```

The Customer class which should contain all the information of one customer, like his name, his ID, the total amount he paid for rides, his position or his message box.

```
<<Java Enumeration>>
    DriverState
  customersAndDrivers

 OFFLINE: DriverState
 ONDUTY: DriverState
 ONARIDE: DriverState
 OFFDUTY: DriverState
 state: String

 DriverState(String)
 getState():String
```

```
<<Java Class>>
    Driver
 customersAndDrivers

 biggestDriverID: int
 car: Car
 driverName: String
 driverSurName: String
 driverID: int
 driverAmount: double
 numOfRides: int
 driverMarks: ArrayList<Integer>
 driversTimes: Map<String,Long>
 lastTimeOfLastStateChange: Date

 Driver(Car,String,String)
 getDriverName():String
 getCar():Car
 setCar(Car):void
 getDriverSurName():String
 getDriverID():int
 getDriverState():DriverState
 getDriverAmount():double
 getDriversTimes():Map<String,Long>
 getNumOfRides():int
 addOneRide():void
 connect(Date):void
 disconnect(Date):void
 pause(Date):void
 unpause(Date):void
 changeStateTo(DriverState,Date):boolean
 toString():String
 addAmount(double):void
 addOnDutyTime(long):void
 addOffDutyTime(long):void
 addOnRideTime(long):void
 getOnDutyRateOfDriving():double
 getRateOfActivity():double
 getOccupationRate():double
 addOneMark(int):void
 avrageOfMarks():double
```

```
-driverState
0..1
```

```
<<Java Class>>
   Customer
 customersAndDrivers

 customerName: String
 customerSurName: String
 customerID: int
 customerPosition: GPScoordinates
 creditCardNumber: int
 Messagebox: ArrayList<String>
 biggestCustomerID: int
 totaleAmountPaid: double
 numOfRide: int

 Customer(String,String,GPScoord...
 getCustomerName():String
 getCustomerSurName():String
 getCustomerID():int
 getCustomerPosition():GPScoord...
 setCustomerPosition(GPScoordin...
 getCreditCardNumber():int
 setCreditCardNumber(int):void
 getMessagebox():ArrayList<String>
 getNumOfRide():int
 getTotaleAmountPaid():double
 addMessageToBox(String):void
```
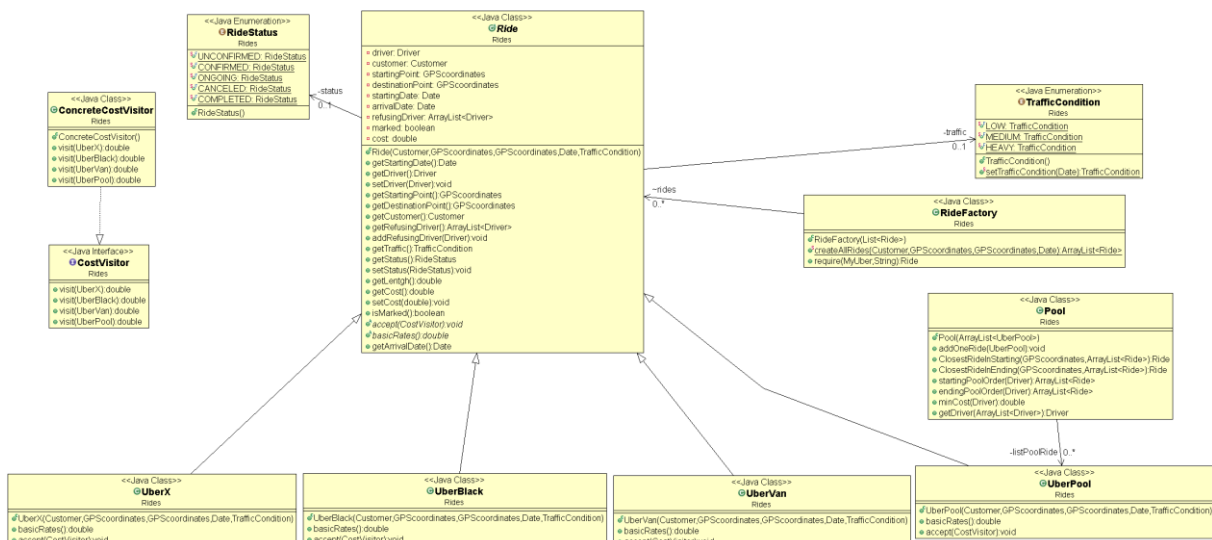
The Driver class should contain the information of one driver, like his name, his ID, his car, his balance, the list of his marks or his status which can be OFFLINE, OFFDUTY, ONDUTY and ONARIDE and shows that the driver can accept a new ride, if he is ONDUTY. The different states also allow the system to compute statistics about the time a driver is driving customers and the time is waiting for them. To implement these states, we choose an enumeration class.

<<Java Enumeration>>
CarState
Cars
- ᔢF TAKED: CarState
- ᔢF AVAILABLE: CarState
- CarState()

-carState 0..1

<<Java Class>>
Car
Cars
- carPosition: GPScoordinates
- Car(GPScoordinates)
- getCarPosition():GPScoordinates
- setCarPosition(GPScoordinates):void
- getCarState():CarState
- setCarState(CarState):void
- *isCompatibleWithTheRide(Ride):boolean*

<<Java Class>>
CarFactory
Cars
- CarFactory()
- create(String,GPScoordinates):Car

<<Java Class>>
Standard
Cars
- ᔢS StandardNumber: int
- carID: String
- numberFreeSeats: int
- Standard(GPScoordinates)
- isCompatibleWithTheRide(Rid

<<Java Class>>
Berlin
Cars
- ᔢS BerlinNumber: int
- carID: String
- numberOfSeats: int
- Berlin(GPScoordinates)
- isCompatibleWithTheRide(R

<<Java Class>>
Van
Cars
- ᔢS VanNumber: int
- carID: String
- numberFreeSeats: int
- Van(GPScoordinates)
- isCompatibleWithTheRide(Ride):boolean

The Car abstract class should contain all the information of the car. The car has a position which is used to find the nearest driver, and a status because two driver can use the same car but not at the same time. The method *changeStateTo(DriverState,Date)* in the Driver class use the car state attribute to check if a driver can connect with his car.

The car class is extended by 3 implementable classes which are corresponding to each type of car. These classes also implement the abstract method *isCompatibleWithTheRide(Ride)* which return a Boolean representing if the car do the specified ride, given its type.

We also choose to implement a factory pattern for cars, which we will discuss in the next part of the report.

<<Java Enumeration>>
RideStatus
Rides
- UNCONFIRMED: RideStatus
- CONFIRMED: RideStatus
- ONGOING: RideStatus
- CANCELED: RideStatus
- COMPLETED: RideStatus
- RideStatus()

-status 0..1

<<Java Class>>
ConcreteCostVisitor
Rides
- ConcreteCostVisitor()
- visit(UberX):double
- visit(UberBlack):double
- visit(UberVan):double
- visit(UberPool):double

<<Java Interface>>
CostVisitor
Rides
- visit(UberX):double
- visit(UberBlack):double
- visit(UberVan):double
- visit(UberPool):double

<<Java Class>>
Ride
Rides
- driver: Driver
- customer: Customer
- startingPoint: GPScoordinates
- destinationPoint: GPScoordinates
- startingDate: Date
- arrivalDate: Date
- refusingDriver: ArrayList<Driver>
- marked: boolean
- cost: double
- Ride(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
- getStartingDate():Date
- getDriver():Driver
- setDriver(Driver):void
- getStartingPoint():GPScoordinates
- getDestinationPoint():GPScoordinates
- getCustomer():Customer
- getRefusingDriver():ArrayList<Driver>
- addRefusingDriver(Driver):void
- getTraffic():TrafficCondition
- getStatus():RideStatus
- setStatus(RideStatus):void
- getLength():double
- getCost():double
- setCost(double):void
- isMarked():boolean
- accept(CostVisitor):void
- basicRates():double
- getArrivalDate():Date

-rides 0..*

<<Java Enumeration>>
TrafficCondition
Rides
- LOW: TrafficCondition
- MEDIUM: TrafficCondition
- HEAVY: TrafficCondition
- TrafficCondition()
- setTrafficCondition(Date):TrafficCondition

-traffic 0..1

<<Java Class>>
RideFactory
Rides
- RideFactory(List<Ride>)
- createAllRides(Customer,GPScoordinates,GPScoordinates,Date):ArrayList<Ride>
- require(MyUber,String):Ride

<<Java Class>>
Pool
Rides
- Pool(ArrayList<UberPool>)
- addOneRide(UberPool):void
- ClosestRideInStarting(GPScoordinates,ArrayList<Ride>):Ride
- ClosestRideInEnding(GPScoordinates,ArrayList<Ride>):Ride
- startingPoolOrder(Driver):ArrayList<Ride>
- endingPoolOrder(Driver):ArrayList<Ride>
- minCost(Driver):double
- getDriver(ArrayList<Driver>):Driver

-listPoolRide 0..*

<<Java Class>>
UberX
Rides
- UberX(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
- basicRates():double
- accept(CostVisitor):void

<<Java Class>>
UberBlack
Rides
- UberBlack(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
- basicRates():double
- accept(CostVisitor):void

<<Java Class>>
UberVan
Rides
- UberVan(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
- basicRates():double
- accept(CostVisitor):void

<<Java Class>>
UberPool
Rides
- UberPool(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
- basicRates():double
- accept(CostVisitor):void

The Ride abstract class contains the attributes and methods that characterize a Ride, like the driver, the customer, the starting and ending point of a ride, its cost or the starting and ending date. We use

the RideStatus and the TrafficCondition enumeration class to represent the status and traffic attribute of the class.

4 classes extend the Ride class to represents all the concrete possible rides, and these Rides can be created using the RideFactory class, which is inspired of the factory pattern. We are also using a visitor pattern to compute the cost of a ride and we will discuss the reason and the implementation of these patterns in the next part of the report.

UberPool ride is very particular from all other kind of rides. In fact, several rides can have the same driver and the same car at the same moment, the search of a driver differs also from other cases because it's based on the aggregation of multiple trajectories.

That is why we created the Pool class where we defined the attributes and methods needed for all the functionalities:
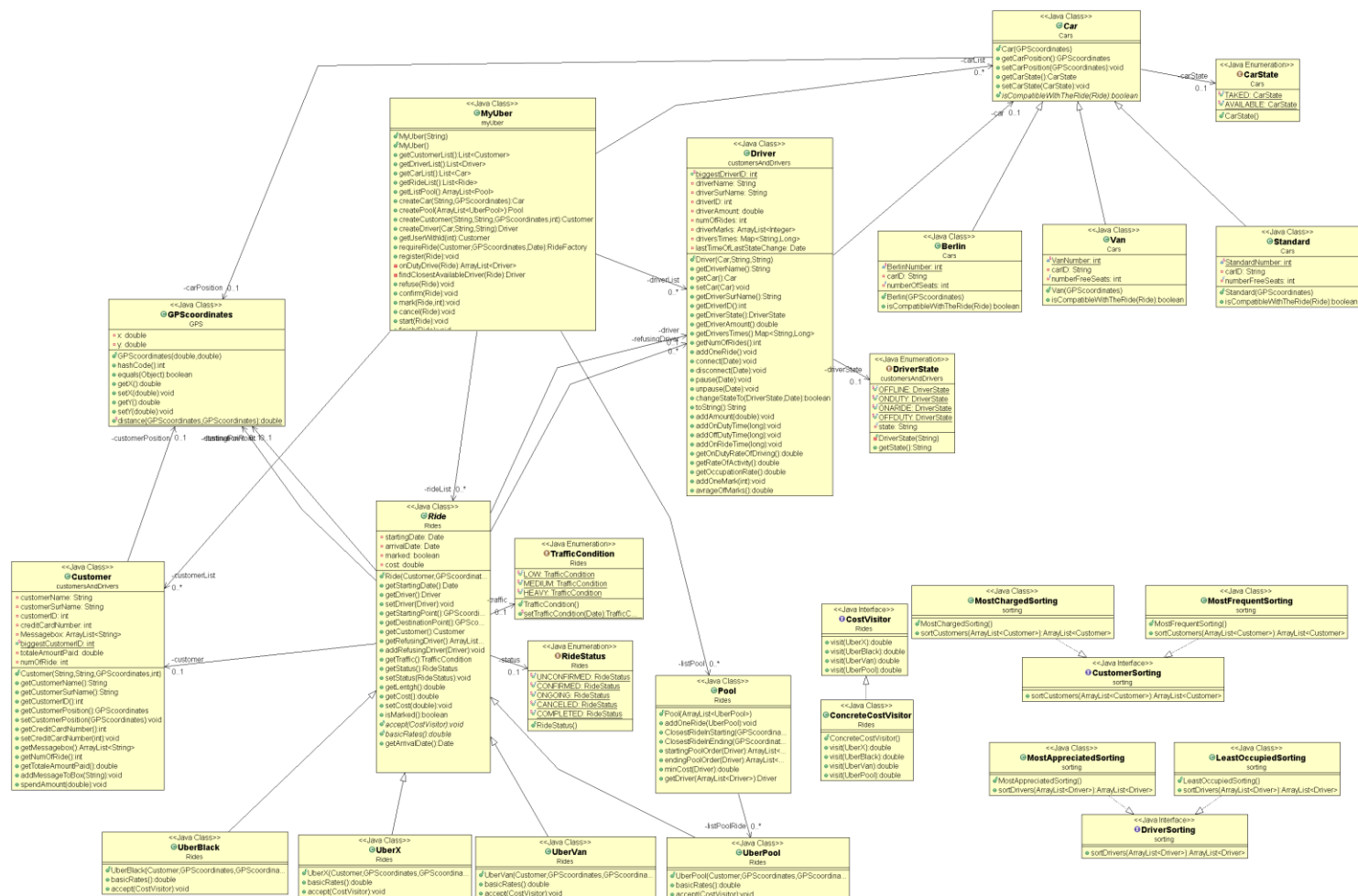
- As principle attribute, we put an ArrayList containing all the shared ride.
- The addOneRide(Ride ride) method to add a new ride to list of all shared rides.
- The ClosestRideInStarting() method to determine the nearest point to the driver between all starting points.
- The ClosestRideInEnding() method to determine the nearest point to the driver between all starting points.
- The StartingPoolOrder to determine the optimal order to follow for taking the customers from their starting points.
- The EndingPoolOrder() to determine the optimal order to follow for dropping off the customers to their destination point starting by a specific position.
- The minCost() method to calculate the minimum cost for a driver from all possible orders.
- The getDriver() method that return the driver with the minimum cost.

Finally, the MyUber class should store the list of all the cars, drivers, customers and rides created for the instance of the class. This class enable many myUber instance to run at the same time with different drivers, cars, customers and rides, what would not be possible if the list of drivers would be a static attribute of the driver class for example.

The myUber class also provides all the methods that a user can use to interact with the system, like methods to create a car, a customer or a driver, to require a ride or to change the status of a ride like confirm, refuse, start and finish for a driver and cancel and mark for a customer.

A MyUber instance can also be created with a text file with the CreateMyUberFromText class. You can then use keywords to create a big number of car, customer and/or drivers and to require a big number of rides with random values. This is a practical way to create an instance with many rides to test, for example, the sorting function.

The UML diagram for all the classes is then the following.

Car (Java Class) — Cars
- Car(GPScoordinates)
- getCarPosition():GPScoordinates
- setCarPosition(GPScoordinates):void
- getCarState():CarState
- setCarState(CarState):void
- isCompatibleWithTheRide(Ride):boolean

CarState (Java Enumeration) — Cars
- TAKED: CarState
- AVAILABLE: CarState
- CarState()

MyUber (Java Class) — myUber
- MyUber(String)
- MyUber()
- getCustomerList():List<Customer>
- getDriverList():List<Driver>
- getCarList():List<Car>
- getRideList():List<Ride>
- getListPool():ArrayList<Pool>
- createCar(String,GPScoordinates):Car
- createPool(ArrayList<UberPool>):Pool
- createCustomer(String,String,GPScoordinates,int):Customer
- createDriver(Car,String,String):Driver
- getUserWithId(int):Customer
- requireRide(Customer,GPScoordinates,Date):RideFactory
- register(Ride):void
- onDutyDrive(Ride):ArrayList<Driver>
- findClosestAvailableDriver(Ride):Driver
- refuse(Ride):void
- mark(Ride,int):void
- cancel(Ride):void
- start(Ride):void

Driver (Java Class) — customersAndDrivers
- biggestDriverID: int
- driverName: String
- driverSurName: String
- driverID: int
- driverAmount: double
- numOfRides: int
- driverMarks: ArrayList<Integer>
- driversTimes: Map<String,Long>
- lastTimeOfLastStateChange: Date
- Driver(Car,String,String)
- getDriverName():String
- getCar():Car
- setCar(Car):void
- getDriverSurName():String
- getDriverID():int
- getDriverState():DriverState
- getDriverAmount():double
- getDriversTimes():Map<String,Long>
- getNumOfRides():int
- addOneRide():void
- connect(Date):void
- disconnect(Date):void
- pause(Date):void
- unpause(Date):void
- changeStateTo(DriverState,Date):boolean
- toString():String
- addAmount(double):void
- addOnDutyTime(long):void
- addOffDutyTime(long):void
- addOnRideTime(long):void
- getOnDutyRateOfDriving():double
- getRateOfActivity():double
- getOccupationRate():double
- addOneMark(int):void
- avrageOfMarks():double

Van (Java Class) — Cars
- VanNumber: int
- carID: String
- numberFreeSeats: int
- Van(GPScoordinates)
- isCompatibleWithTheRide(Ride):boolean

Standard (Java Class) — Cars
- StandardNumber: int
- carID: String
- numberFreeSeats: int
- Standard(GPScoordinates)
- isCompatibleWithTheRide(Ride):boolean

Berlin (Java Class) — Cars
- BerlinNumber: int
- carID: String
- numberOfSeats: int
- Berlin(GPScoordinates)
- isCompatibleWithTheRide(Ride):boolean

DriverState (Java Enumeration) — customersAndDrivers
- OFFLINE: DriverState
- ONDUTY: DriverState
- ONRIDE: DriverState
- OFFDUTY: DriverState
- state: String
- DriverState(String)
- getState():String

GPScoordinates (Java Class) — GPS
- x: double
- y: double
- GPScoordinates(double,double)
- hashCode():int
- equals(Object):boolean
- getX():double
- setX():double):void
- getY():double
- setY(double):void
- distance(GPScoordinates,GPScoordinates):double

Ride (Java Class) — Rides
- startingDate: Date
- arrivalDate: Date
- marked: boolean
- cost: double
- Ride(Customer,GPScoordinat...
- getStartingDate():Date
- getDriver():Driver
- setDriver(Driver):void
- getStartingPoint():GPScoordi...
- getDestinationPoint():GPSco...
- getCustomer():Customer
- addRefusingDriver():ArrayList...
- addRefusingDriver(Driver):void
- getTraffic():TrafficCondition
- getStatus():RideStatus
- setStatus(RideStatus):void
- getLength():double
- getCost():double
- setCost(double):void
- isMarked():boolean
- accept(CostVisitor):void
- basicRates():double
- getArrivalDate():Date

TrafficCondition (Java Enumeration) — Rides
- LOW: TrafficCondition
- MEDIUM: TrafficCondition
- HEAVY: TrafficCondition
- TrafficCondition()
- setTrafficCondition(Date):TrafficC...

RideStatus (Java Enumeration) — Rides
- UNCONFIRMED: RideStatus
- CONFIRMED: RideStatus
- ONGOING: RideStatus
- CANCELED: RideStatus
- COMPLETED: RideStatus
- RideStatus()

Customer (Java Class) — customersAndDrivers
- customerName: String
- customerSurName: String
- customerID: int
- creditCardNumber: int
- Messagebox: ArrayList<String>
- biggestCustomerID: int
- totaleAmountPaid: double
- numOfRide: int
- Customer(String,String,GPScoordinates,int)
- getCustomerName():String
- getCustomerSurName():String
- getCustomerID():int
- getCustomerPosition():GPScoordinates
- setCustomerPosition(GPScoordinates):void
- setCreditCardNumber(int):void
- getMessagebox():ArrayList<String>
- getNumOfRide():int
- getTotaleAmountPaid():double
- addMessageToBox(String):void
- spendAmount(double):void

Pool (Java Class) — Rides
- Pool(ArrayList<UberPool>)
- addOneRide(UberPool):void
- ClosestRideStarting(GPScoordina...
- ClosestRideEnding(GPScoordinat...
- startingPoolOrder(Driver):ArrayList<...
- endingPoolOrder(Driver):ArrayList<...
- minCost(Driver):double
- getDriver(ArrayList<Driver>):Driver

CostVisitor (Java Interface) — Rides
- visit(UberX):double
- visit(UberBlack):double
- visit(UberVan):double
- visit(UberPool):double

ConcreteCostVisitor (Java Class) — Rides
- ConcreteCostVisitor()
- visit(UberX):double
- visit(UberBlack):double
- visit(UberVan):double
- visit(UberPool):double

MostChargedSorting (Java Class) — sorting
- MostChargedSorting()
- sortCustomers(ArrayList<Customer>):ArrayList<Customer>

MostFrequentSorting (Java Class) — sorting
- MostFrequentSorting()
- sortCustomers(ArrayList<Customer>):ArrayList<Customer>

CustomerSorting (Java Interface) — sorting
- sortCustomers(ArrayList<Customer>):ArrayList<Customer>

MostAppreciatedSorting (Java Class) — sorting
- MostAppreciatedSorting()
- sortDrivers(ArrayList<Driver>):ArrayList<Driver>

LeastOccupiedSorting (Java Class) — sorting
- LeastOccupiedSorting()
- sortDrivers(ArrayList<Driver>):ArrayList<Driver>

DriverSorting (Java Interface) — sorting
- sortDrivers(ArrayList<Driver>):ArrayList<Driver>

UberBlack (Java Class) — Rides
- UberBlack(Customer,GPScoordinates,GPScoordina...
- basicRates():double
- accept(CostVisitor):void

UberX (Java Class) — Rides
- UberX(Customer,GPScoordinates,GPScoordina...
- basicRates():double
- accept(CostVisitor):void

UberVan (Java Class) — Rides
- UberVan(Customer,GPScoordinates,GPScoordina...
- basicRates():double
- accept(CostVisitor):void

UberPool (Java Class) — Rides
- UberPool(Customer,GPScoordinates,GPScoordina...
- basicRates():double
- accept(CostVisitor):void

Patterns use in our implementation

To achieve a code that respects the Open Close principle, we used multiple design patterns:

Factory pattern for creating cars:

We create a factory named CarFactory to generate objects of Car classes based on a given string. The factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily add a new type of car by adding a class implementation and another "if" block in the CarFactory class. The new type of car should then override the isCompatibleWithRide to easily specify the type of ride it can perform.

This pattern makes it also easier to create a Car in the myUber class because we only need a single method *createCar* that call the CarFactory *create* method and the createCar method of the MyUber class respect then the open-close principle.
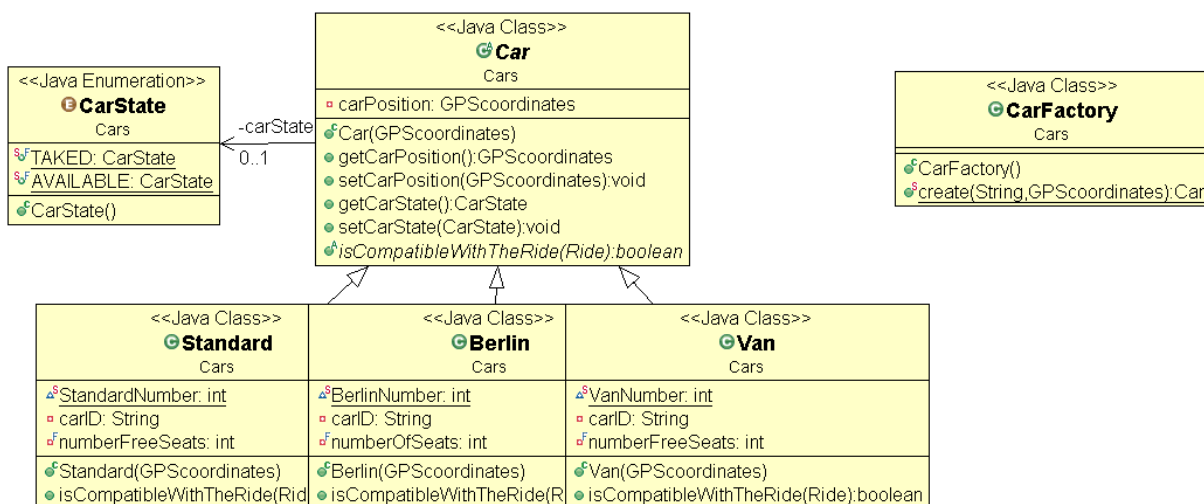
Visitor pattern of rides:

As the fare computation function depends on the kind of ride booked, we decided to use a visitor pattern which consists of:

- Visitable abstract class **"Rides"**: It declares the accept() method that invokes the visit() method of the visitor object passed as parameter. The choice of an abstract class instead of an interface was due to the number of similar and identical methods and attributes used in the UberX, UberBlack, UberVan and UberPool classes.

- Visitor interface **"CostVisitor"**: It declares the visit() method

That way, our components (UberX, UberBlack, UberVan, UberPool) will allow the visitor implementation to "visit" them and perform the required action on that element.

Consequently**, we make good use of the Open/Closed principle** as we won't modify the code to add a new functionality, but we'll still be able to extend the functionality by providing a new *Visitor* implementation.

Ride (Java Class) — Rides

driver: Driver
customer: Customer
startingPoint: GPScoordinates
destinationPoint: GPScoordinates
startingDate: Date
arrivalDate: Date
status: RideStatus
refusingDriver: ArrayList<Driver>
marked: boolean
cost: double
traffic: TrafficCondition

Ride(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
getStartingDate():Date
getDriver():Driver
setDriver(Driver):void
getStartingPoint():GPScoordinates
getDestinationPoint():GPScoordinates
getCustomer():Customer
getRefusingDriver():ArrayList<Driver>
addRefusingDriver(Driver):void
getTraffic():TrafficCondition
isCompatibleWithTheRide(Ride,Car):boolean
getStatus():RideStatus
setStatus(RideStatus):void
getLength():double
getCost():double
setCost(double):void
isMarked():boolean
accept(CostVisitor):void
basicRates():double
getArrivalDate():Date

ConcreteCostVisitor (Java Class) — Rides

ConcreteCostVisitor()
visit(UberX):double
visit(UberBlack):double
visit(UberVan):double
visit(UberPool):double

CostVisitor (Java Interface) — Rides

visit(UberX):double
visit(UberBlack):double
visit(UberVan):double
visit(UberPool):double

UberX (Java Class) — Rides

UberX(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
basicRates():double
accept(CostVisitor):void

UberPool (Java Class) — Rides

UberPool(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
basicRates():double
accept(CostVisitor):void

UberBlack (Java Class) — Rides

UberBlack(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
basicRates():double
accept(CostVisitor):void

UberVan (Java Class) — Rides

UberVan(Customer,GPScoordinates,GPScoordinates,Date,TrafficCondition)
basicRates():double
accept(CostVisitor):void

Factory pattern for creating rides:

RideFactory (Java Class) — Rides

rides: List<Ride>

RideFactory(List<Ride>)
createAllRides(Customer,GPScoordinates,GPScoordinates,Date):ArrayList<Ride>
require(MyUber,String):Ride

The RideFactory class is inspired of a factory pattern, even if it is not actually one, because the ride creation only occurs in a specific case: the customer makes a ride request, the system send him back the prices for all the existing rides and the customer finally choose its ride.

So, when we create a ride, we must create all type of rides, then compute the prices for all of them and finally register the chosen one in the MyUber instance.

Therefore, we have two methods in the RideFactory class.

The createAllRides static method takes in argument all the arguments of the Ride constructor and return a list of all type of ride constructed with the given arguments.

With this list, the system can compute the prices for all rides with the visitor pattern.

Then we can create an instance of RideFactory, with the above list in arguments.

And then we can choose the ride by calling the require method on this instance of RideFactory and specifying the type of ride with a String in parameter.

So, if we want to add a type of Ride, we just have to create the class extending Ride, add it to the returned list of the *createAllRides* method and add an "if" case in the require method to match with the String in parameter. So, the RideFactory makes the Ride class more flexible, because new subclasses are easy to create.

Strategy pattern for sorting drivers and customers:

Our requirement was to design a family of algorithms in order to sort

1. Customers based on the number of completed rides and the total amount of charges for all completed rides.
2. Drivers based on their occupation rate and the level of appreciations.

For that, we used two strategy patterns: We defined two sorting algorithms for each case, encapsulate each one, and make them interchangeable.
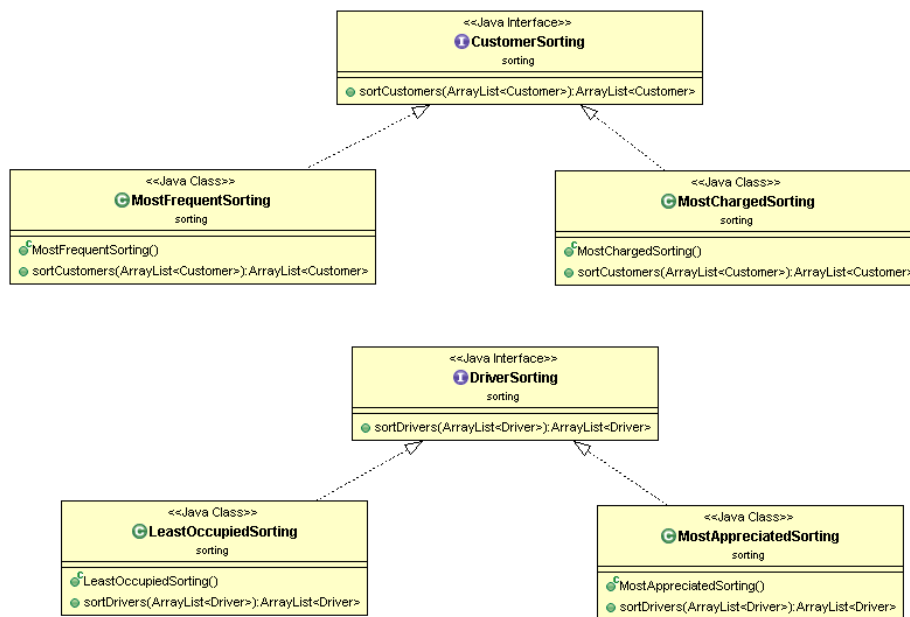
If we want to sort drivers or customers with new criteria, the only thing to do is adding new classes with its implementations. This is compatible with the open/closed principle (OCP), which proposes that classes should be open for extension but closed for modification.