Software Engineering Part 1

Group 43

Progress and division of labor

At first, we worked together to define the UML diagram we needed and to choose the patterns we will use. Then, we decided to find a way to work efficiently on the project without getting in trouble with version management. To do so, we used github to make our collaboration easier.
After that, we splitted the classes we had to implement.
Vincent worked on the rides, the costs and the statistics while I developed the car Factory, the customer environment including the Request class.
In addition, we met quite often to make sure that we were doing well and that the project was on the right track.
We both worked on the myUber class to develop the main functionalities.
Besides, we tried to document our functions and use JUnit cases as soon as possible when we add functions.

Components of myUber system

Car : A car is defined by its owner, its ID and the type of ride it is made for. Moreover, due to the obligation to be able to add new kind of cars easily, we have decided to use a factory pattern for this class. Thus, creation of new car is made by the car factory and the logic creation behind each car is hidden to the client.

Driver : A driver is defined by his ID, his name and surname and his state. No pattern is used for this class.

Customer : Détailler le Observer pattern used here. Rides : Due to the similarities on the requirements between cars and rides we have decided to implement rides using a factory pattern. Indeed, thanks to this pattern the client can't see what is behind rides creation and we can easily add a new type of ride (which is an obligation if we can easily add new types of car).

Distance computation : We decided to use the euclidean distance. Indeed, that was not possible to get a real map so every model would have been simplified and the euclidean distance was quite natural if we consider the distance as the crow flies.

Booking of a car ride

Booking process : We created a class request that manage the whole booking process from the client destination setting to the ride type decision. This class enable us to make sure that the order of the task is respected : set destination, receive prices, choose ride type, find a driver…

Driver decision : We currently supposed that every driver accept the ride. We made this choice because we stated that our drivers are not making decision.

Uber Pool

We used a queue that stores the pool requests. When this queue contains 3 requests, the algorithm sort the "on-duty" uberPool drivers by the distance they would need to drive to realize the pool ride. So we first offer the ride to the first driver and if he refuses we offer it to the second etc… To do so, we compute the minimal distance required for every driver to realize the ride. This minimal distance is calculated by testing every order of customers pick up and drop off.

Time and Movement

We decided not to manage the time in our application. Indeed, we consider that every ride is realized instantly because we considered that the aim of project is rather to book rides than to build a real time application. Although we could do it by created one thread by ride and using the command sleep to interrupt its execution until the ride is finished.

About cost and statistics

Calculation of the cost of a ride

The specification of this calculation was :

— To be able to define different fare function depending on the type of ride, the length of the ride, the state of the traffic and the hour of the reservation.

— To be able to define different fare function depending on the type of ride, the length of the ride, the state of the traffic and the hour of the reservation.

— To be able to add new fare function very easily.

Considering those requirements, we decided to use a visitor pattern to implement cost calculation. Indeed, definition of fare function depends on the object we are performing, thus using this pattern permits to define new fare calculation without changing classes of the rides.

Statistics

After each ride, relevant statistics on drivers, customers and system in general are computed and they can be consulted thanks to the methods driverBalance, customerBalance and systemBalance.

Concerning the treatment of those statistics, they can be sorted thanks to four different methods (see myUber for more details). Those methods were implemented thanks to the comparator technology which enables to define how to compare to elements (for instance how to compare two drivers) and then provide an already sorting method.

Explanations about use case scenario

Use case 1 : UberPool Three customers set their destination so the uberPool ride is launched and drops the three customers off their destination.

Use case 2 : The customer c1 (Baptiste Andrieu) wants to go to the destination dest1.
The four type of rides are proposed and he chooses the UberVan type.
The ride is completed and the customer can note the driver.

Use case 3 : Customer c1 (Baptiste Andrieu), c2 (Noé Mikati) and c3 (Christophe Gallon) want to go respectively to destinations dest 6, dest7 and dest3.
The four type of rides are proposed and they all choose the UberPool type.
The ride is completed and customers can note the driver.

Use case 4 : Statistics about two rides in addition to the precedent rides are collected. Then driver and customers are sorted in order to print :

— the most charged customer

— the customer the most using the application

— the most appreciated driver

— the most occupied driver

File .ini

Unfortunately, we didn't succeed in using a .ini file to initialize the platform. As a consequence, we had to put the definitions in the test scenario files. We are sorry for this but we didn't find any clue about it in the lecture and couldn't find any clear explanation on the internet.