MY UBER

Version 1+2 : Core + CLUI + GUI

Lily Monnier, Timothy Meehan

Table des matières

I. Introduction

Today's increasing mobility is a huge issue to be dealt with. With the rise of smartphone, all services of mobility must be adapted. It is in this context that Uber has been created.

My Uber is a simplified version of the mobile application Uber: customers book rides that are submitted to available drivers.

This java project is complex because of the diversity of the interactions between very different entities. Indeed, a ride implies a car, one or several customers and a driver, that can all interact with each other: a car can be rented by a driver, a ride is requested by a customer and accepted by a driver. Because so many different entities must be created, the Object Oriented Software Design fits well this project. Nevertheless, its complexity requires a real reflexion for the core structure of the code.

I. Core Design

1) Main characteristics

Because we want to dissociate the action of creating a new object from the user's request, we thought of the Factory Pattern to tackle this issue. My uber implies the creation of very different objects (car, ride, customer, driver). We decided to implement an Abstract Factory Pattern that will handle the instantiation of a Car Factory, a Customer Factory, a Driver Factory and a Ride Factory. This allows to extend the family of product with minor modifications. Nonetheless, some modifications would still have to be done, especially in the other factories.

Our project is divided mainly in the following units:

- Factories: objects that create instances of different entities (Customer, Car, Ride, Driver),
- Customer: person that books rides,
- Driver: person that accepts rides and rents cars,
- Car: object rented by drivers in order to execute a ride,
- Ride: object created during Booking, requested by a customer and accepted by a driver,
- Booking: process of booking a ride,
- PriceCalculator: object that compute the ride's characteristics (cost, length etc.),
- Balances: general statistics (Customer, Driver, System).

2) Main processes

a) Ride's characteristics

This part is treated in the package PriceCalculator. With a Visitor Pattern, we compute the characteristics of the different type of rides (cost, length etc.), by visiting each instance created and initialised with the parameters of the customer. This allows few modifications if we want to add another type of ride with a different process of computing those values.

For the computation of the length, we found this expression in the literature:

$$d = \mathrm{Arcos}(\sin(\mathrm{latitude1}) * \sin(\mathrm{latitude2}) + \cos(\mathrm{latitude1}) * \cos(\mathrm{latitude1}) * \cos(\mathrm{longitude1} - \mathrm{longitude2})) * \mathrm{R}_{\mathrm{Terre}}$$

We had to create a class GPS for the manipulation of the coordinates. Moreover, as we need to compute the distance between 2 GPS that belongs to different types of objects (Customer, Car, Driver), it is even more convenient to have a special class to treat all kind of actions regarding the characteristics of the ride.

For the attribution of traffic state, we used the random functionalities of Java: considering the time of reservation, we have assigned the matching probability. We created an attribute containing this table, so it would be easier to change those probabilities. Indeed, the method does not depend on the values of this table but on this table itself. We implemented an interface for the Visitor Pattern and a superclass for the different type of TrafficCalculators (Low, Medium, Heavy). We chose a superclass rather than an interface because a lot of attributes and methods were shared. We can still add easily a different type of TrafficCalculator.

b) Booking a ride

We decided to implement a superclass called Booking rather than an interface, as a lot of attributes and methods were shared between the 2 subclasses ClassicBooking and BookingUberPool. It still allows us to create new processes of booking.

The process of booking a ride is described below:

1- The customer starts a requestRide method (parameters: departure, destination, number of passengers, distance maximum to the driver);

2- A booking object is created and associated to this customer (number of passengers is set).

The request method computes the characteristics (cost, length etc.) of each ride type after creating it and stocking it in the booking's attribute "rideTypes" (array of rides).

3- The customer starts a treatRequest method (parameters: choice of ride type, distance maximum between the driver and the customer, and between 2 customers for a shared ride). It generates either a ClassicBooking or a BookingUberPool, instances of the subclasses of Booking, and the ride chosen is associated to it. The customer and the compatible drivers among the drivers previously found are registered as observers of this ride.

In BookingUberPool, we had to ensure the fact that an UberPool ride needs at least 2 customers compatibles (small distance to each other). In order to do that, we created an attribute called listOfRequests in BookingUberPool. When a second ride is compatible, we look if there is a third ride compatible. If not, the 2 rides are set with all the parameters of the other customer.

In both, the drivers found are sorted by the minimum distance to the customer and the nearest is selected for the ride.

4- Once the ride is accepted by a driver, its state changes and all the observers are notified. Messages are sent to the customers to inform them about the state of the ride.

5- At the end of the ride a message is sent asking the customer to rate the ride and the driver. This mark will be added to the system balances.

3) Design Pattern

a) Observer Pattern

Because users concerned by a ride needs to know its state and may act when it changes, we decided to implement an Observer Pattern: a ride is observed by the drivers that are likely to accept it, and by the client that has requested it. This allows us to notify all the users concerned by a change of the ride's state.

We created an interface ObserverRide in the package users that is implemented by concrete Observers (Customers and Drivers). In the package rides, we created an interface called Observable that is implemented by the superclass Ride (concrete observables). By doing this, 2 types of observers can be registered on the 4 types of ride. Moreover, we can specify the action of the observers according to type of ride.

b) Strategy Pattern

As the price of a ride depends on the traffic, we decided to implement a Strategy Pattern in the priceCalculator package. Indeed, by creating an interface PriceCalculator that is implemented by the superclass TrafficCalculator, we can specify in the 3 following subclasses: HeavyTrafficCalculator, LowTrafficCalculator, MediumTrafficCalculator, the method to follow in order to calculate the price of a ride regarding the traffic state.

The advantage of this pattern is that the user is not concerned by the choice of the method implemented. That is to say that the customer doesn't have to know how the procedure work for the different subclasses.

c) Visitor Pattern

Because the price of a ride depends on its type and must be calculated simultaneously for each type, we decided to implement a Visitor Pattern. Indeed, it allows us to specify how to calculate the price regarding the ride's type, but also to execute this action simultaneously for each type.

The interface PriceCalculator of the package priceCalculator is the visitor and the visitable is an interface implemented by the superclass Ride in the package rides. The concrete visitable is the different type of rides (UberBlack, UberPool, UberVan, UberX).

d) Singleton Pattern

As we wanted a unique instance of each factory or of MyUber or of SystemBalance, we thought of the Singleton Pattern to tackle this issue. In fact, it guarantees us that there is no duplicated object. Nonetheless, due to a lack of time, we did not implement the thread-safe and serializable Singleton Pattern.

For the different ID, as we use it as a unique attribute of an instance of a class, we didn't implement the Singleton Pattern. In fact, it would have required to create a class IDGenerator following a Singleton Pattern design. Instead, for each class, we have created a private static attribute called "compteur", which is incremented each time we create a new ID in that class.

As we did not create a class request corresponding to the unconfirmed rides, we had to be careful with the rides' ID. Indeed, we only give an ID to the ride once it is accepted by a driver.

II. CLUI Design

To design the Command Line User Interface, we used a switch and cases in order to differentiate the processes.

For the different cases we treated, we used the scanner functionalities of Java to retrieve the command instructions. We also implemented try-and-catch statements in order to handle the exceptions mostly generated by errors in the command instructions or in the inputs are handled.

1) "help"

This command is used when the user wants information about the commands available. He will find information like the name of the commands, the parameters needed, and a brief description of the actions of each command.

2) "initialisation"

This command is used to initialize the system by creating several cars regarding to the type and drivers associated, also several customers. All the coordinates are initialised randomly in the given area.

3) "setup"

This command allows the user to initialise interactively the system. With a scanner, all the inputs are read, and the system is initialised the same way as during the previous command. The output here is the state of the system such as the list of cars and their current driver, the list of drivers and the list of customers. It uses the toString methods of the different classes.

4) "displayState"

This command is used when general information about the system are needed. The output is the state of the system (list of cars, list of drivers and list of customers). It uses he toString methods of the different classes.

5) "displayCustomer"

This command displays the list of customers sorted either by charge or by frequency. The choice is read by a scanner, then the right sorting method is called. Here also, the toString method of the Customer class is used.

6) "displayDriver"

This command displays the list of drivers sorted either by occupation or by popularity. The choice is read by a scanner, then the right sorting method is called. The toString method of the Driver class is called.

7) "displayTotalCashed"

This command displays the total amount of money spent on rides. To do so, we retrieve this value in the System Balance class.

8) "addCustomer"

This command is used to add a customer to the system. The inputs required are read by a scanner such as the name or the surname. Their coordinates are generated randomly in the given area. The output is the list of the customers, using the toString method of their class.

9) “addCarDriver”

This command is used to add a customer and its car to the system. To read the required inputs, we used a scanner. Their coordinates are generated randomly in the given area but are the same since they are associated. The output is the list of drivers and the list of cars using their toString methods. In car.toString() the current driver is displayed, so we had to handle the case when a car is not rented.

10) “addDriver”

This command is used to add a driver to the system and precise which car of the system he is renting. We used a scanner to read the inputs necessary. Their coordinates are generated randomly in the given area. The output is the same as the previous command. Here 2 NullPointerExceptions must be handled: one if the car does not exist and an another one if it is not available.

11) “setDriverStatus”

This command is used to change the status of the driver (on duty to off duty or the contrary). The driver is retrieved by its ID. A NullPointerException must be handled if the driver doesn’t exist.

12) “moveCar”

This command is used to move a car from its localisation to a destination. The car is retrieved by its ID and can generate a NullPointerException.

13) “moveCustomer”

This command is the same as the previous but for customers.

14) “ask4price”

This command realises the process requestRide of the core code. A scanner is used to read the inputs required. The outputs are the prices of the different ride types using the last messages sent to the customer (method in MessageBox class). As a client is retrieved by its ID, a NullPointerException must be handled when the client doesn’t exist.

15) “simRide_i”

This command is used to simulate a ride. It implements the processes requestRide, treatRequest, addRideMark, addDriverMark of the core code. The inputs are read by a scanner. The output is a summary of the ride and of the state system. NullPointerExceptions must here also be handled.

16) “simRide”

This command is the same as the previous one except that the ride type is entered before knowing the price.

17) “exit”

This command allows us to exit the system.

18) “default”

This command informs the user that he entered a wrong command.

III. GUI

The GUI was realized thanks to the CardLayout which allowed us to display different input tables according to the command selected. A list of subpanels have been created and added as attributes of the myUberGUI instance.

Each Subpanel is associated to a method which edits the content of the output panel.

IV. Realization of the workload

1) Design structure

After reading the subject, we started thinking about the design structure of the code. First, we wrote down the different class we would need, precising major functions they will implement. Then for each class we listed the design patterns that could work. Finally, we chose the design patterns that we would implement.

2) Repartition of the code

At first, we agreed to work on the foundation of the code (create classes, interfaces etc.) simultaneously and separately. We quickly realized that it was hard to work with another person on a code. We tried to use Git, but as we didn't know how it worked, we gave up this idea. We decided to split the work and add the updates on a drive. Nonetheless, merging the documents was difficult, as each class had to be checked to see the updates. Even with a good repartition of the work, as we worked simultaneously, we were not as efficient as we wish due to the dense interactions among the classes.

For the precise repartition, all basic classes or functions were handled by both of us at the beginning. Then when we had to code the difficult functions, we assigned each task:

- Timothy: the priceCalculator package, the users package, the balances package, the others package, GUI.

- Lily: the booking package, the rides package, the factories package, CLUI.

Finally, we spent a lot of time gathering and merging all the work we did: each week we did a briefing about the modifications we had done. Especially at the end, it was really long to put together everything. We didn't manage to fix in the time given the CLUI with reading and writing in a file.