A.Y 2015-2016

Software Engineering 2: "myTaxiService"

Design **D**ocument

Version 1.0

Massimiliano Paci (mat. 852720)
Giovanni Patruno (mat. 852658)

December 4th 2015

Contents

1 Introduction

1.1 Purpose

This is the Design Document and its scope is to provide the design of our application with an explanation of our choices. This is a technical document useful to the future developing of the system.

1.2 Scope

The aim of this project is to create a platform, named **myTaxiService**, to optimize the taxi service of a large city.

Passengers will be able to book a taxi for a certain route, view a cost preview and the relative waiting time.

Taxi drivers will be able to inform the system about their availability in a certain zone, confirm or deny a ride and consult the feedback of the passengers.

The city is divided in taxi zones and each one is associated to a queue of taxis. The system assigns every booking to an available driver in the ride's zone. If there is not a driver available in that specific zone, the system automatically assigns another available driver from another zone.

Taxi drivers will have access to the platform too: they will be able to check their queue state and and also see statistics on their routes.

The platform will be scalable and it could be integrated with other applications to implement new functionalities.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Ride: a taxi booking performed by a passenger.
- Zone: A delimited square area (0.2 km^2) at which belongs addresses.
- Passenger: who really travel with the taxi.
- Profile: Every registered user has got a profile which contains all necessary info needed by the platform. We have three types of profile corresponding to three type of users.

1.3.2 Acronyms

- DD: Design Document
- EWT: Estimated Waiting Time

1.3.3 Abbrevations

- iff: if and only if

1.4 Reference Documents

- *Stakeholder specification document*
- *DD TOC.pdf* for a template of the document
- *myTaxiServicePP RASD*

1.5 Document Structure

The document is structured in the following sections:

1. This section contains a generic introduction to the DD
2. This section describes the architectural design proposed for the application. It contains useful diagrams to understand better the system.
3. This section contains some algorithms to satisfy some system's functionalities
4. This section contains the Design of the User Interface of the application, linked to the mockups of the RASD document.
5. In this section we explain how we have satisfied the requirementes illustrated in rhe RASD document

2 Architectural Design

2.1 Overview

In this chapter we will discuss about our design choices proposed for myTaxiService also with the help of some diagrams.

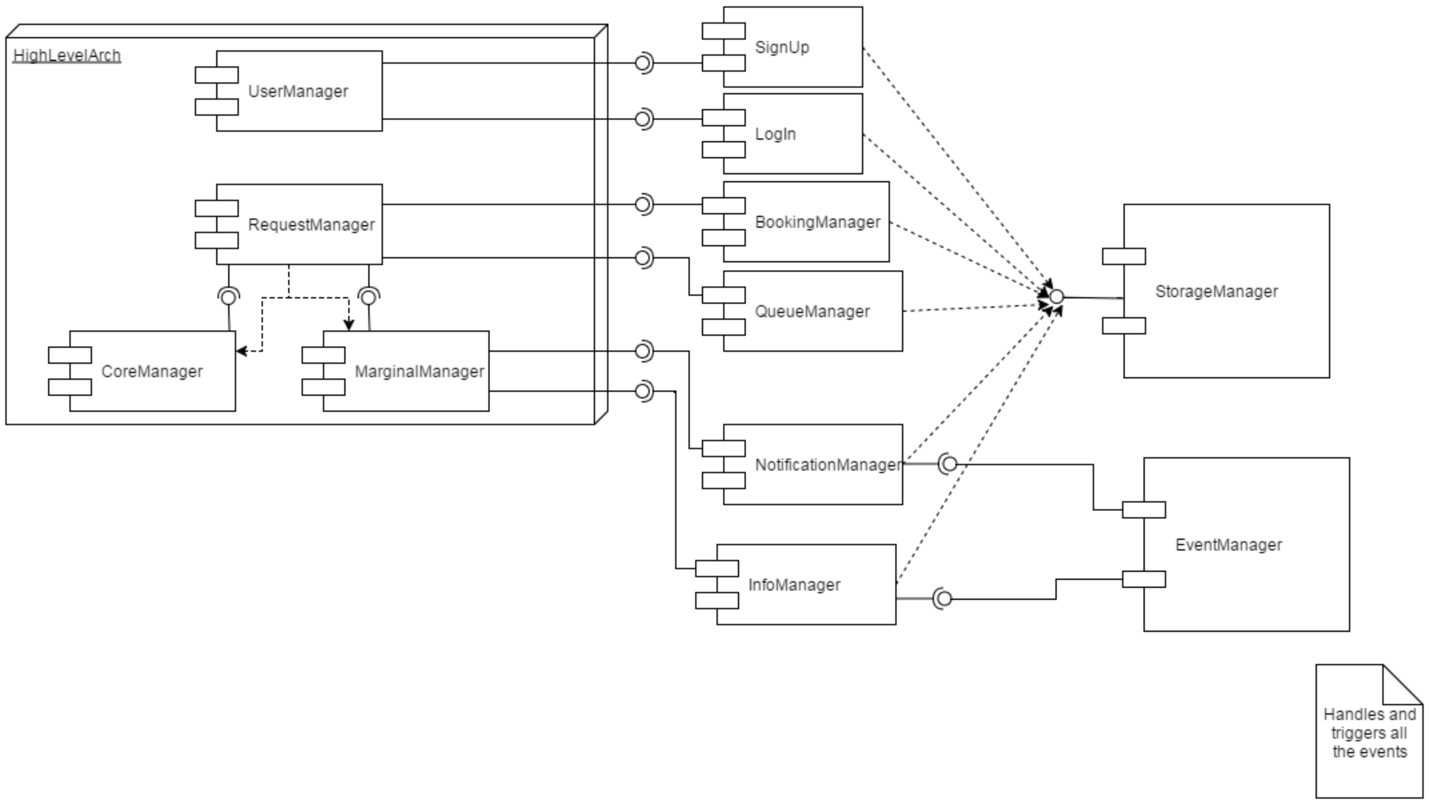
2.2 High Level components and their interaction

We have chosen to centralize the system, as a set of controllers, in order to handle all requests type and keep updated all the application's informations (notifications, bookings and so forth) for all the "guests" (Taxi driver, passenger and administrator) that are involved in the entire system. There will be a subscription on specific topics that are bookings and there will be a broker (System) that will notify in different ways the subscribers. In order to be as integrated as possible, for this application, will be available API to get info about rides, booking and so forth.

2.3 Component View

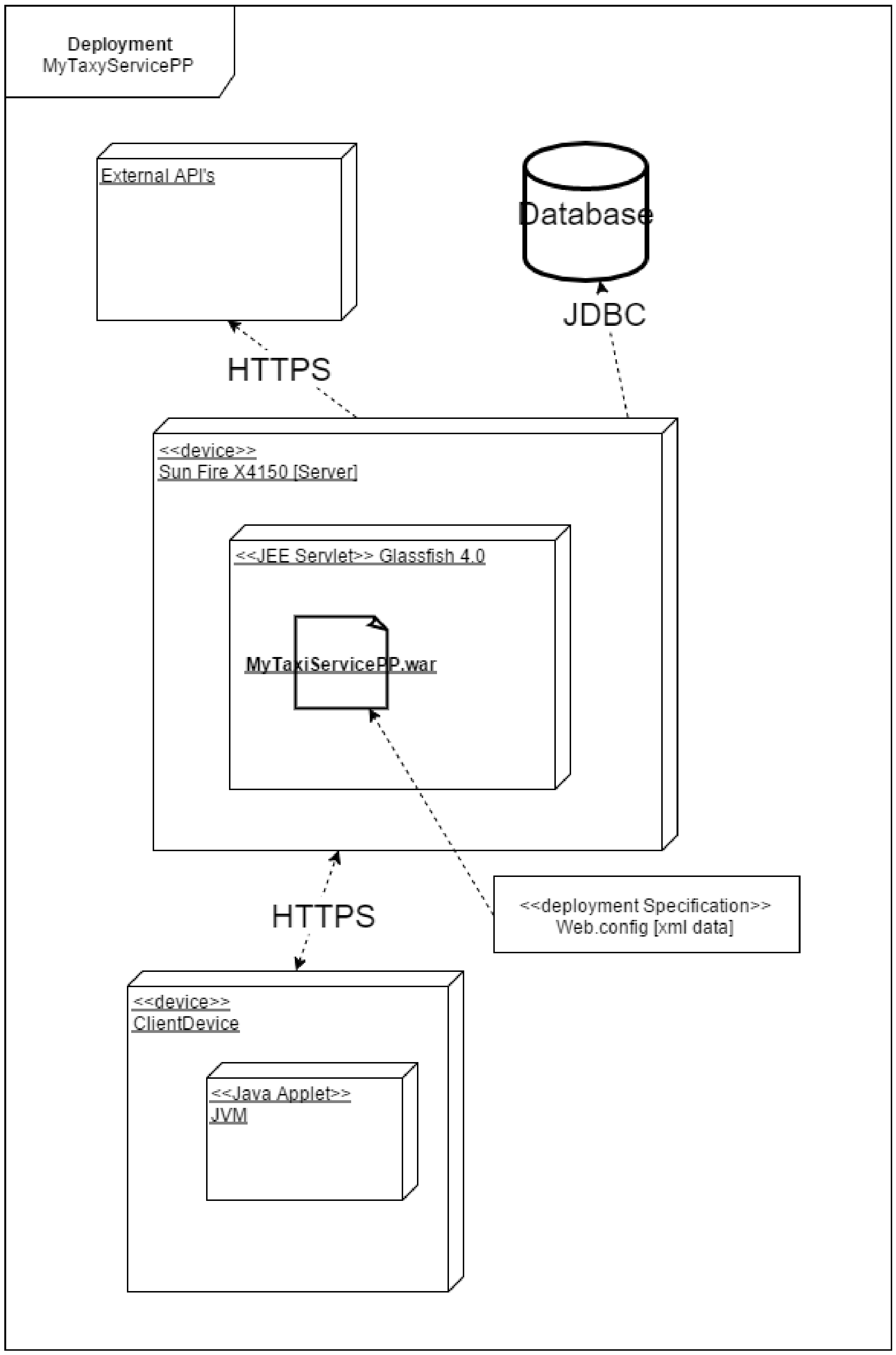
With this diagram we want to give an high level idea on what has guided us to create the entire software architecture and how we will reach this goal by connecting software functionalities thanks to the interfaces.

Notes: StorageManager will be the framework that will handle the storage of the information of the system on the database. InfoManager will be the functionalities that will evaluate the EWT and the estimated price (see Class Diagram → Controller → BookingManager)



2.4 Deployment View

Below we can see the deployment diagram that explores the architecture of embedded systems (middleware used to connect the disparate machine each other).

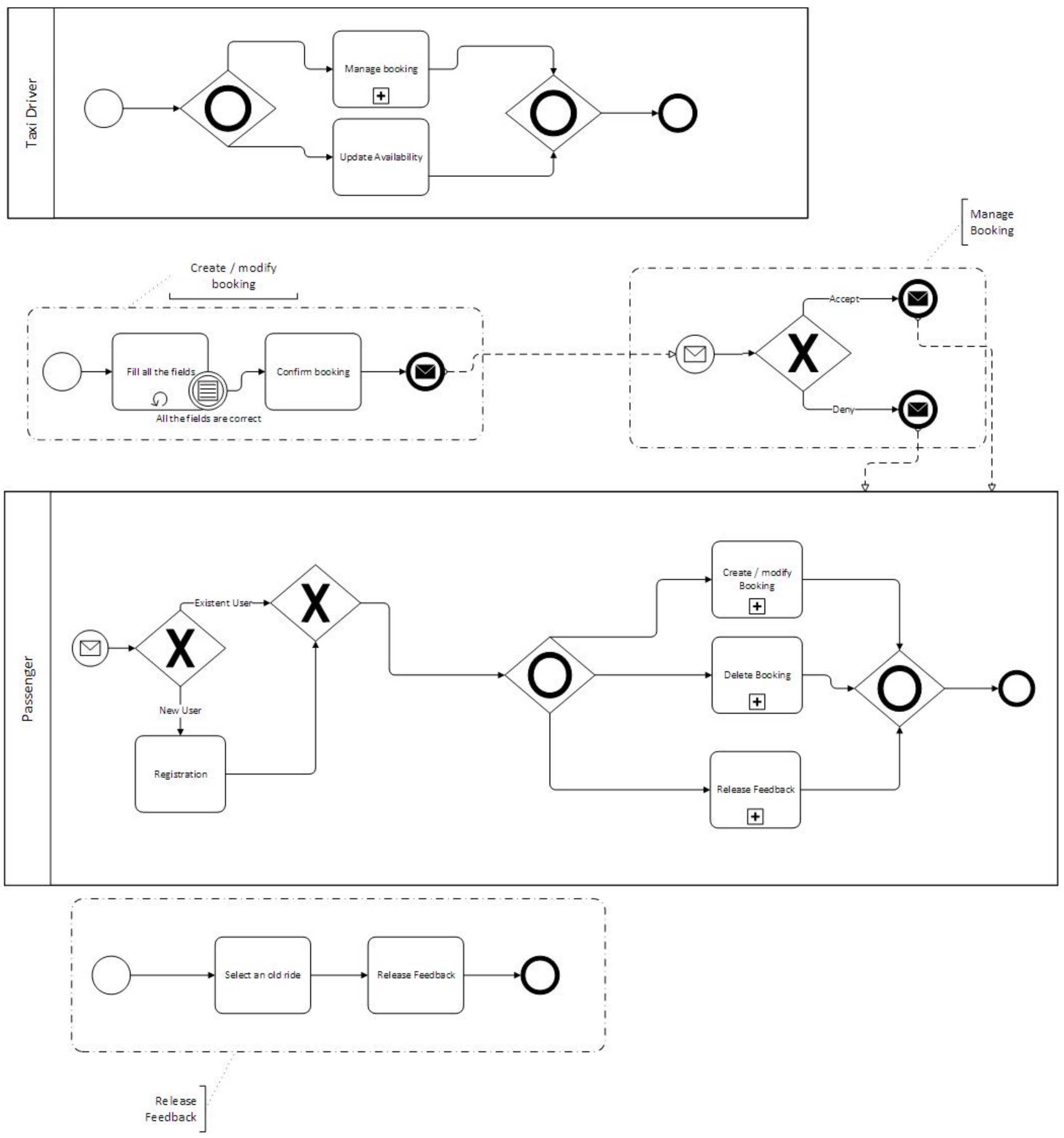


Deployment
MyTaxyServicePP
External API's
Database
JDBC
HTTPS
<<device>>
Sun Fire X4150 [Server]
<<JEE Servlet>> Glassfish 4.0
MyTaxiServicePP.war
HTTPS
<<deployment Specification>>
Web.config [xml data]
<<device>>
ClientDevice
<<Java Applet>>
JVM

2.5 Runtime View

2.5.1 BPMN Diagram

In this section we will propose a BPMN Diagram (Business Process Modeling Notation) to describe the flow of the activities of our application.
We have 3 pool corresponding to the 3 author but in the following diagram we have represented only the two most significative: Passenger and Taxi Driver.

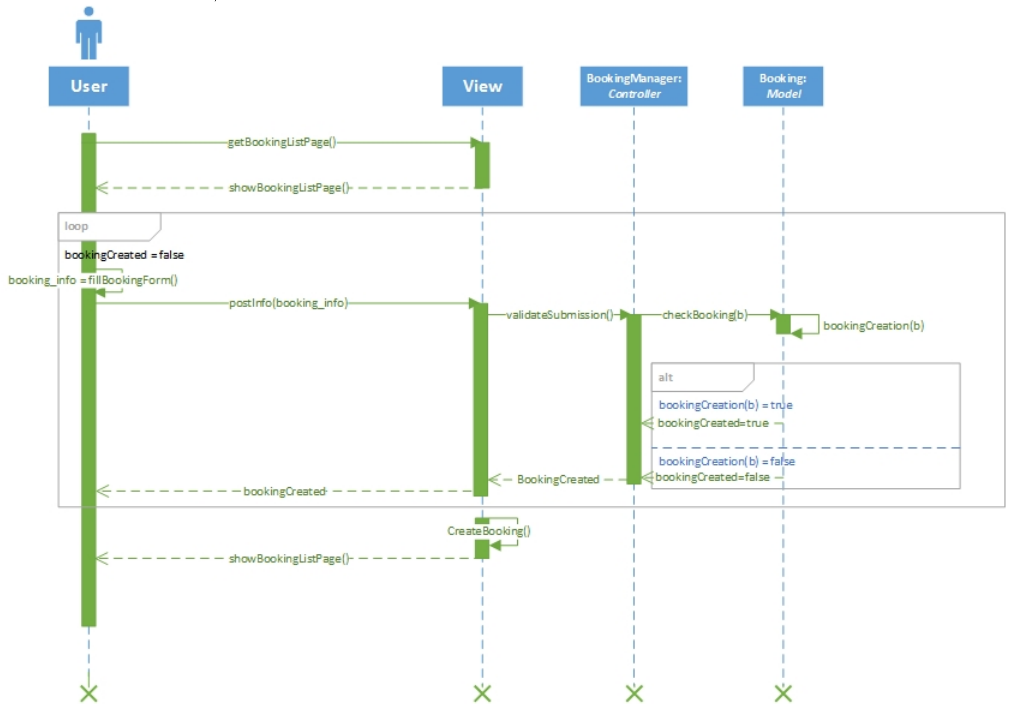

Taxi Driver
Manage booking
Update Availability
Create / modify booking
Fill all the fields
All the fields are correct
Confirm booking
Manage Booking
Accept
Deny
Passenger
Existent User
New User
Registration
Create / modify Booking
Delete Booking
Release Feedback
Select an old ride
Release Feedback
Release Feedback

2.5.2 Detailed Sequence Diagrams

We propose some detailed sequence diagrams in order to clarify and point out the way in which the application handles some kind of situations.

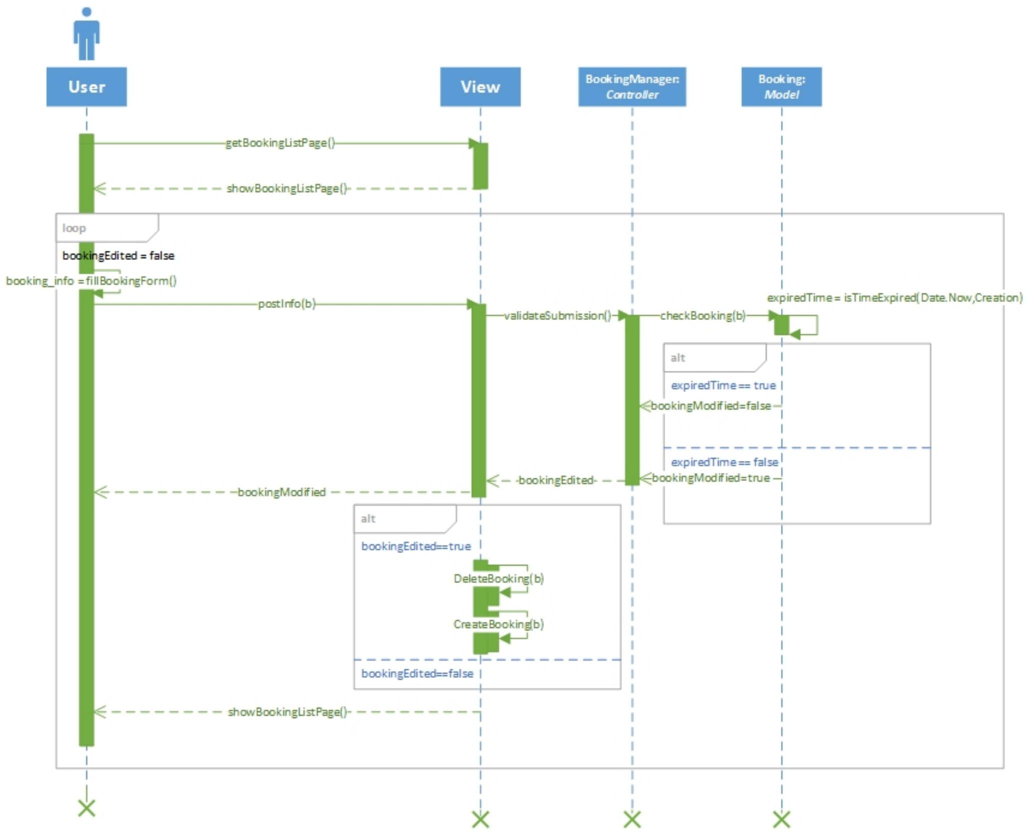
2.5.2.1 Booking Creation

This sequence diagram describes the effective creation of a booking specifying how the controller, view and model interact with each other.



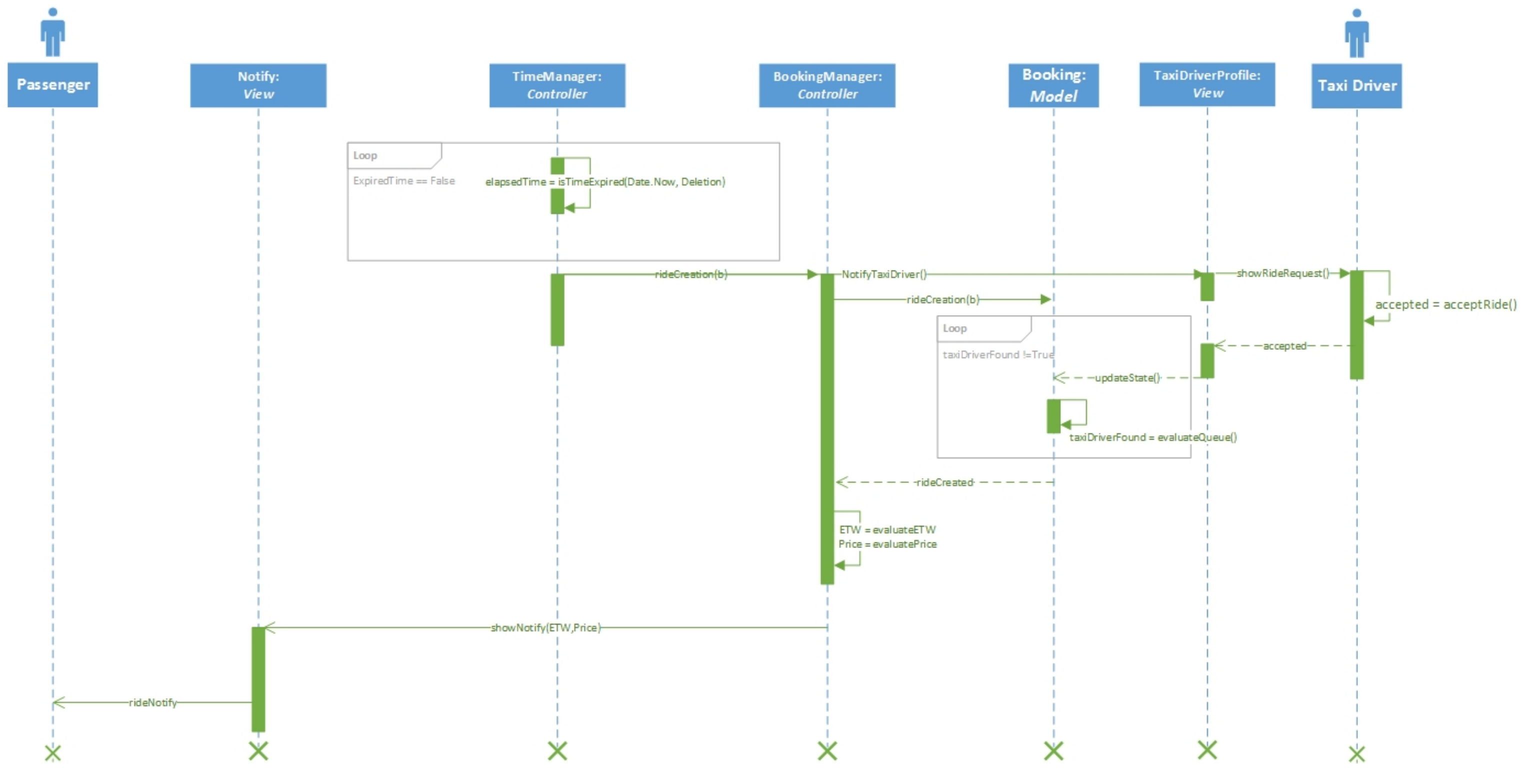
2.5.2.2 Booking Editing

This sequence diagram describes the booking editing done by a user, isTime-Expired method evaluates a boolean variable that becomes true iff the time for the date, taken as parameter, has terminated for the request (also this value is taken as a parameter). In this example we check if the time is expired for the selected request.



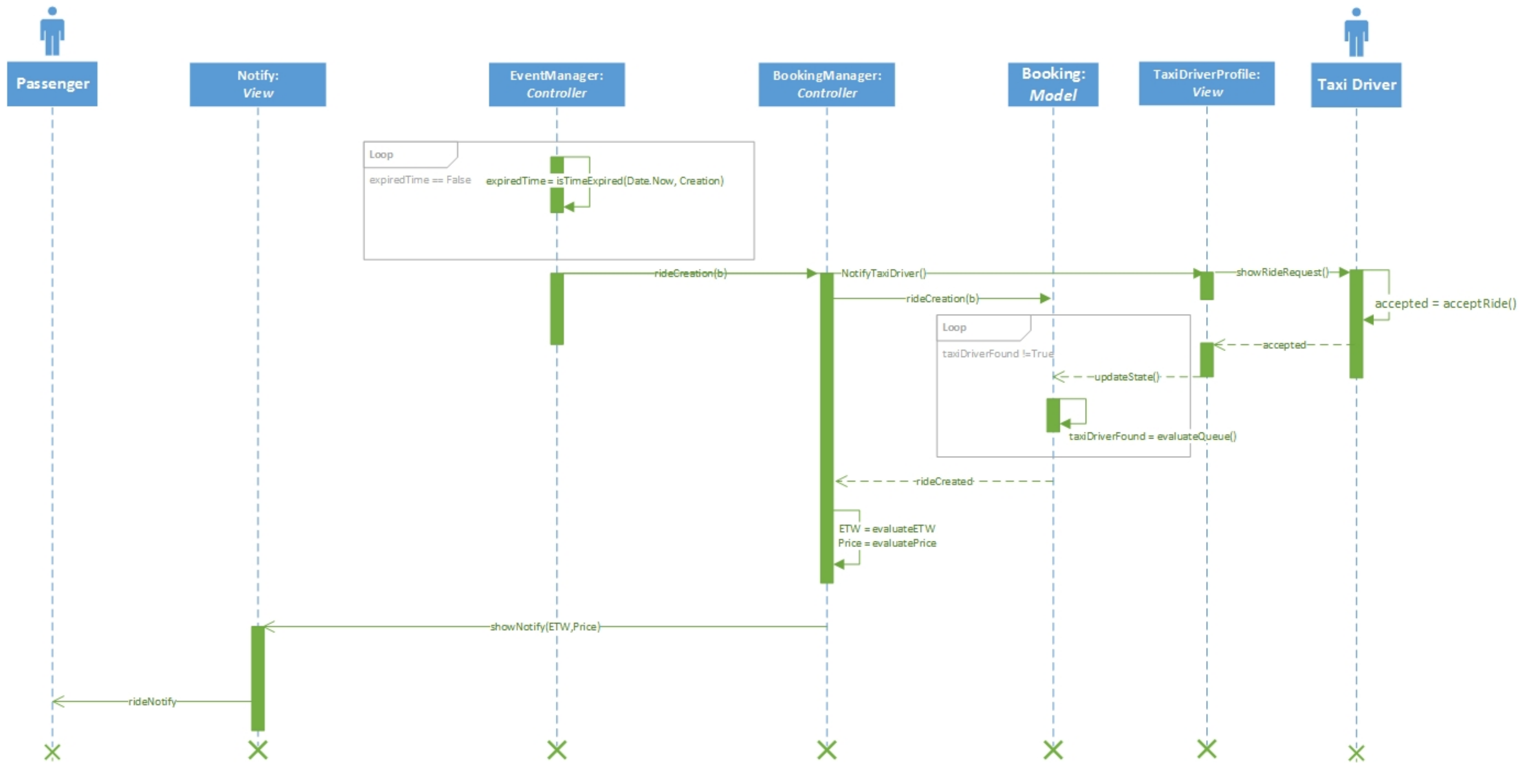
2.5.2.3 Booking Deletion

This sequence diagram describes the booking deletion done by a user checking if the time is expired for the selected (deleting) request



2.5.2.4 Booking Management

This sequence diagram describes the booking management of the system that checks constantly the moment in which it will try to find a taxi driver to associate at that booking. Note that the time controller is a multi-thread process because of the possible interleaved bookings.



2.6 Selected architectural styles and patterns

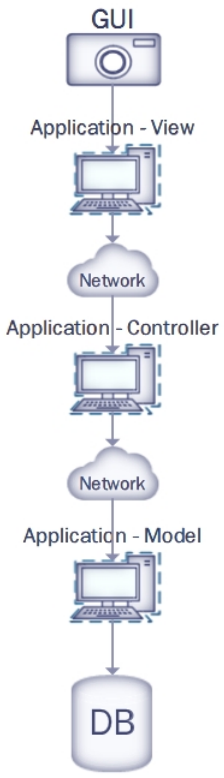
Supplementary definitions

- Guest: Represents the part of the application (Front-end) that allows a user (Passenger, Taxi Driver or Administrator) of the platform to perform a generic request, can be a *Subscriber* or a *Client* (see below for these definitions)
- Host: Represents the part of the application (Back-end) that handles the requests and provides all the information, can be a *Broker* or a *Server* (see below for these definitions)
- Client: Represents the application part that performs a request for a specific service (e.g. create a bookings, get a page and so far) to the Server
- Subscriber: Represents the part of the application that is in hearing of some multicast message send by the broker
- Server: Represents the part of the application that handles the requests sent by the clients

- Broker: Represents the part of the application that sends all the message for a subset of subscriber, this subset, in theory, could coincide with the entire set of the application's users.

2.6.1 Multi-level architecture

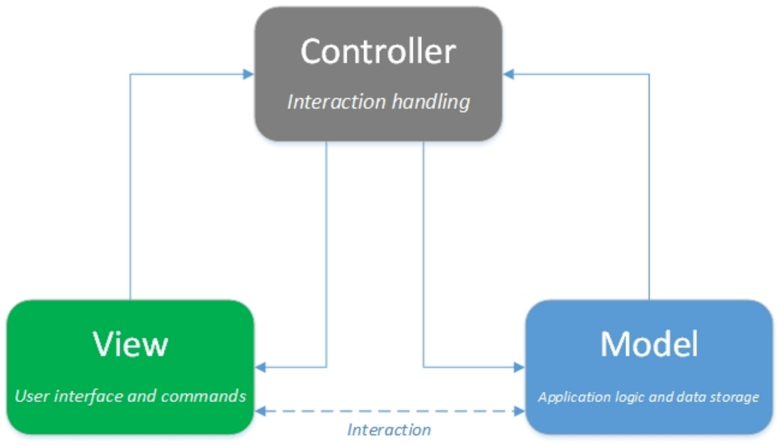
There will be three tier to fit as well as possible the MVC design patterns (described below): The View will interact with the GUI state catching inputs and requests by the clients and will communicates with the controller side that handles these types of request; The model will handle the software logic and will save all data in the database working with a certain framework.



2.6.2 Architectural patterns

2.6.2.1 MVC

We decided to use the classic MVC (Model-View-Controller) to split the business logic, presentation layer and data layer.


Controller
Interaction handling
View
User interface and commands
Model
Application logic and data storage
Interaction

Model
In the model there will be the functional logic of the components of the system and interactions with the storage manager in order to save in the database the information about the environment.
View
The view will strictly handle the communication between the user inputs and the application catching all the interactions and sending them through a controller.
Controller
Controller will be divided in two sections. The first one will manage the communication between the logic (provided by the Model) and the view interactions; the second will handle the external API requests from other applications. The two section will work in parallel.

2.6.2.2 Event based

The event based patter that we have chosen is the publisher-subscriber pattern will be useful to manage all types of notifications. For scalability reasons this pattern will allow the system (broker) to publish all kind of topic for the subscriber guests that will be used from passengers or taxi drivers. These guests application will subscribe to some topics in order to be notified.

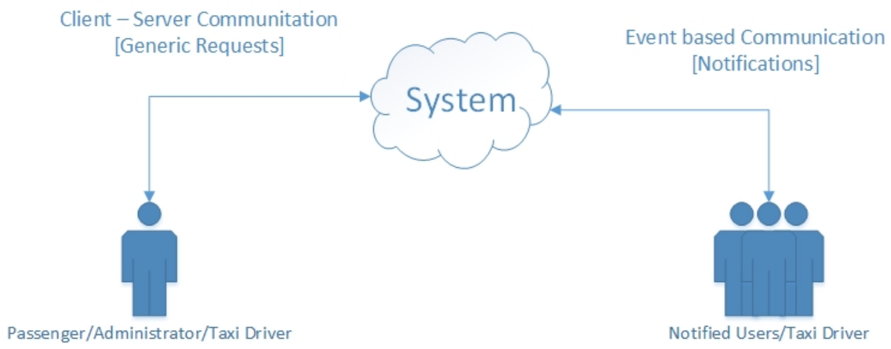
2.6.2.3 Client-Server

The Client-Server pattern will be useful to manage the request/response messages.Clients, situated in the guests' machines will contact the server for a specific requests and then will wait for the response message.

2.6.3 Design patterns

Referencing to the class diagram that has been showed we will highlight the design pattern that will be implemented:

- Singleton - RequestManager, EventManger (controller)
- Observer - Bookings as topics and Users as observer (model)
- Strategy - Various pages (view)
- Factory - RequestManager as factory and APIRequestHandler as products.

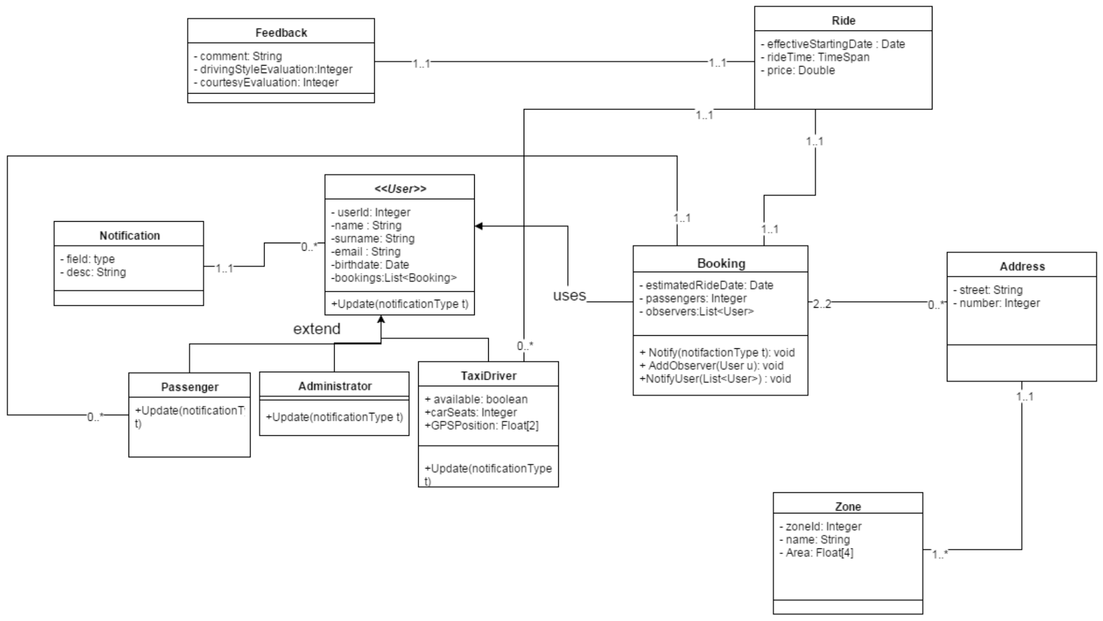

Client – Server Communitation
[Generic Requests]
System
Event based Communication
[Notifications]
Passenger/Administrator/Taxi Driver
Notified Users/Taxi Driver

2.7 Class Diagrams

We decided in general to split the diagram in three parts (Model, View and controller) and to highlight only the arrows that represent the askings by the classes, to keep it orderly.

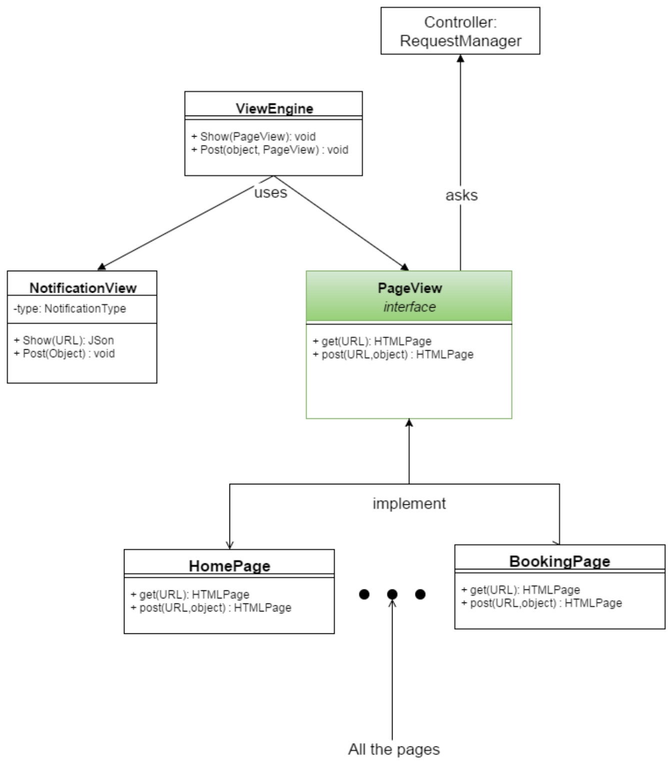
2.7.1 Model

In the section of the model we decided to omit all the auto-explaining class such as *Request*, *Response* or HTMLPage and so forth. Those classes will be constructed by the developer to let the freedom to manage all the parameters, methods and structures. Note that a *Ride* is created after the creation of a *Booking* and it represents the effective travel of the passenger, so if a feedback is associated to a ride, this ride must exist to respect integrity constraints.



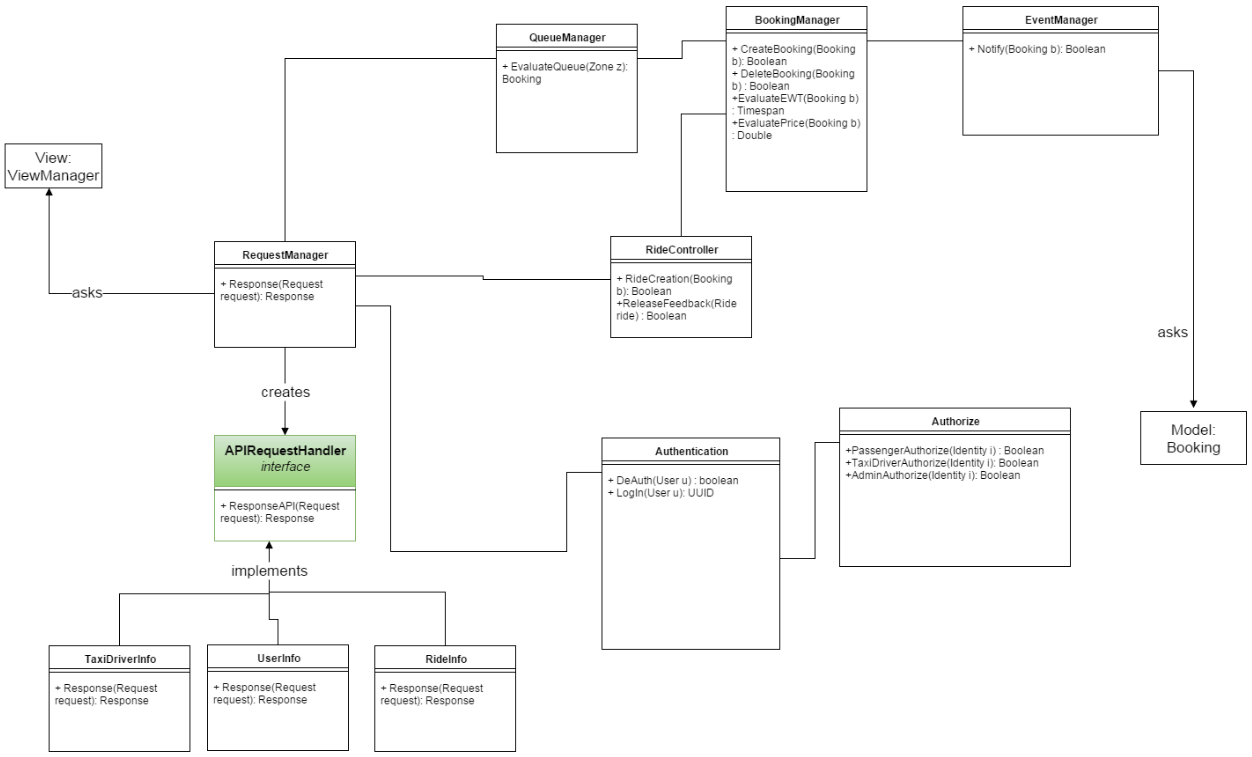
2.7.2 View

All application pages will implement *PageView* interface, for scalability reasons the classes will have standard get and post methods.



2.7.3 Controller

In the controller all the classes will interact with model features freely, we do not specifying this in the graphic (e.g., BookingManager will use the Booking methods to store the information into the database and to check the parameters). API request handler is designed to provide server-side access method from the applications clients. Authentication and authorize will handle the authorization controls and will convert the identity in an active user object of our system.


QueueManager
+ EvaluateQueue(Zone z): Booking
BookingManager
+ CreateBooking(Booking b): Boolean
+ DeleteBooking(Booking b) : Boolean
+EvaluateEWT(Booking b) : Timespan
+EvaluatePrice(Booking b) : Double
EventManager
+ Notify(Booking b): Boolean
View: ViewManager
RequestManager
+ Response(Request request): Response
asks
RideController
+ RideCreation(Booking b): Boolean
+ReleaseFeedback(Ride ride) : Boolean
asks
creates
APIRequestHandler
interface
+ ResponseAPI(Request request): Response
Authentication
+ DeAuth(User u) : boolean
+ LogIn(User u): UUID
Authorize
+PassengerAuthorize(Identity i) : Boolean
+TaxiDriverAuthorize(Identity i): Boolean
+AdminAuthorize(Identity i): Boolean
Model: Booking
implements
TaxiDriverInfo
+ Response(Request request): Response
UserInfo
+ Response(Request request): Response
RideInfo
+ Response(Request request): Response

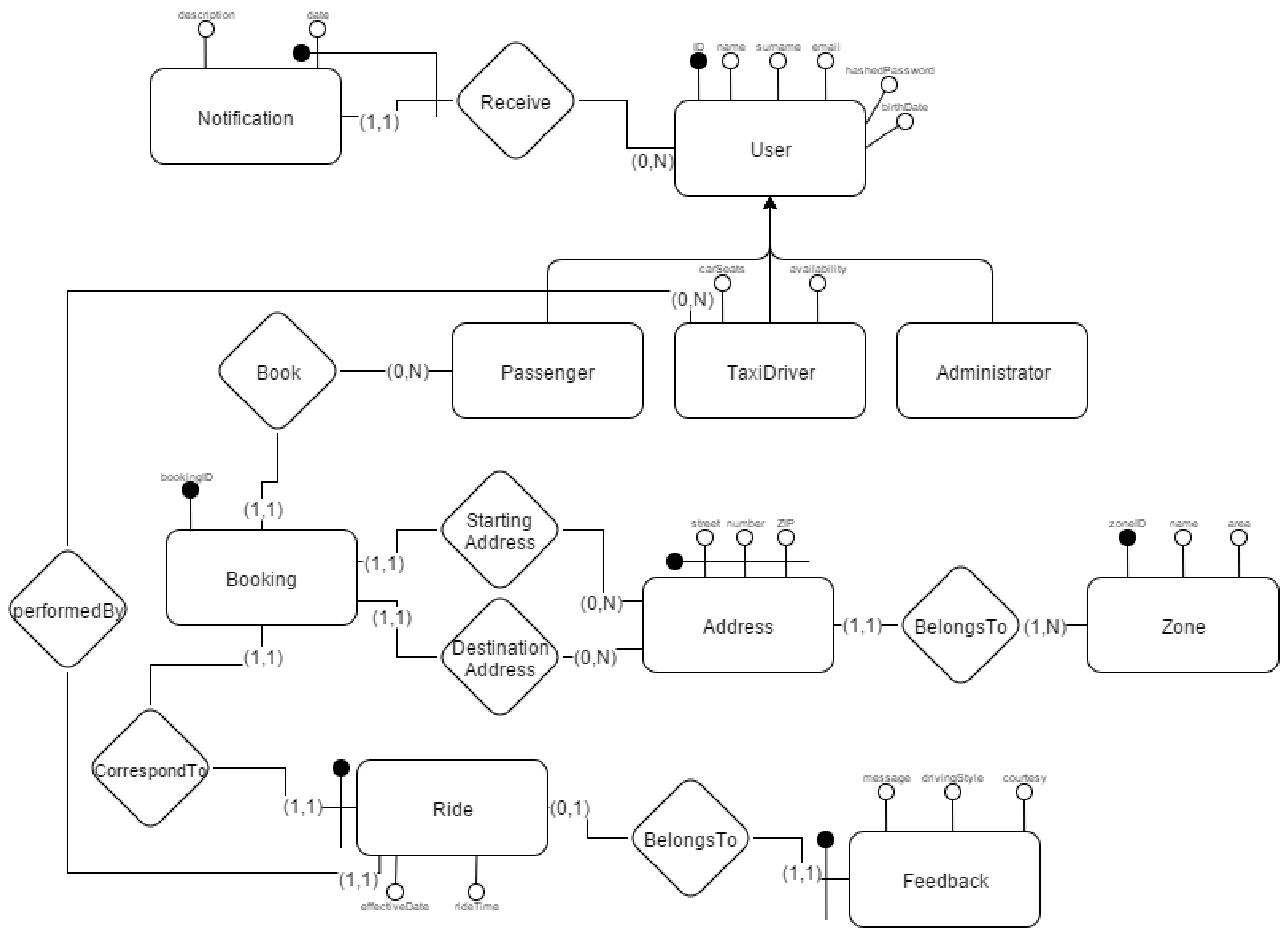
2.8 Other design decisions

2.8.1 Database Design

In this section we will propose a design of our database by an Entity Relationship Diagram.

2.8.1.1 Conceptual Design

With the following diagram we will give a a conceptual design that represents how the data will be stored and the relationship between them.



2.8.1.2 Logical Model

The Logical Design is directly derived from the conceptual one and it represents the database structure of the platform.

PASSENGER (<u>passengerId</u>, name, surname, email, hashedPassword, birthDate)
ADMINISTRATOR (<u>adminId</u>, name, surname, email, hashedPassword, birthDate)
TAXIDRIVER (<u>taxiDriverId</u>, name, surname, email, hashedPassword, birthdate, carSeats, availability)
NOTIFICATION (<u>date</u>, <u>user</u>, description)
BOOKING (<u>bookingId</u>, estimatedDate, passenger, startingCity, startingStreet, startingNumber, startingZIP, destinationCity, destinationStreet, destinationNumber, destinationZIP)
ADDRESS (<u>city</u>, <u>street</u>, <u>number</u>, <u>zip</u>, zone)
ZONE (<u>zoneId</u>, name, area)

RIDE (bookingId, effectiveDate, duration, taxiDriver)
FEEDBACK (bookingId, message , drivingStyle, courtesy)

3 Algorithm Design

3.1 Fare Algorithm

We propose the following formula to estimate the cost of a ride. There are a set of parameters that can be customizable by the Taxi Driver Society

Customizable parameters:

- initialCharge = [€]   Initial charge of the ride. It depends by day and time of the ride
- chargeDistance = [€/km]   Charge per kilometer
- chargeTime = [€/h]   Charge per hour
- distanceTrigger= [km]   It indicates after how many kms the totalCost will be updated
- timeTrigger= [h]   It indicates after how many hours the totalCost will be update

Fixed parameters:

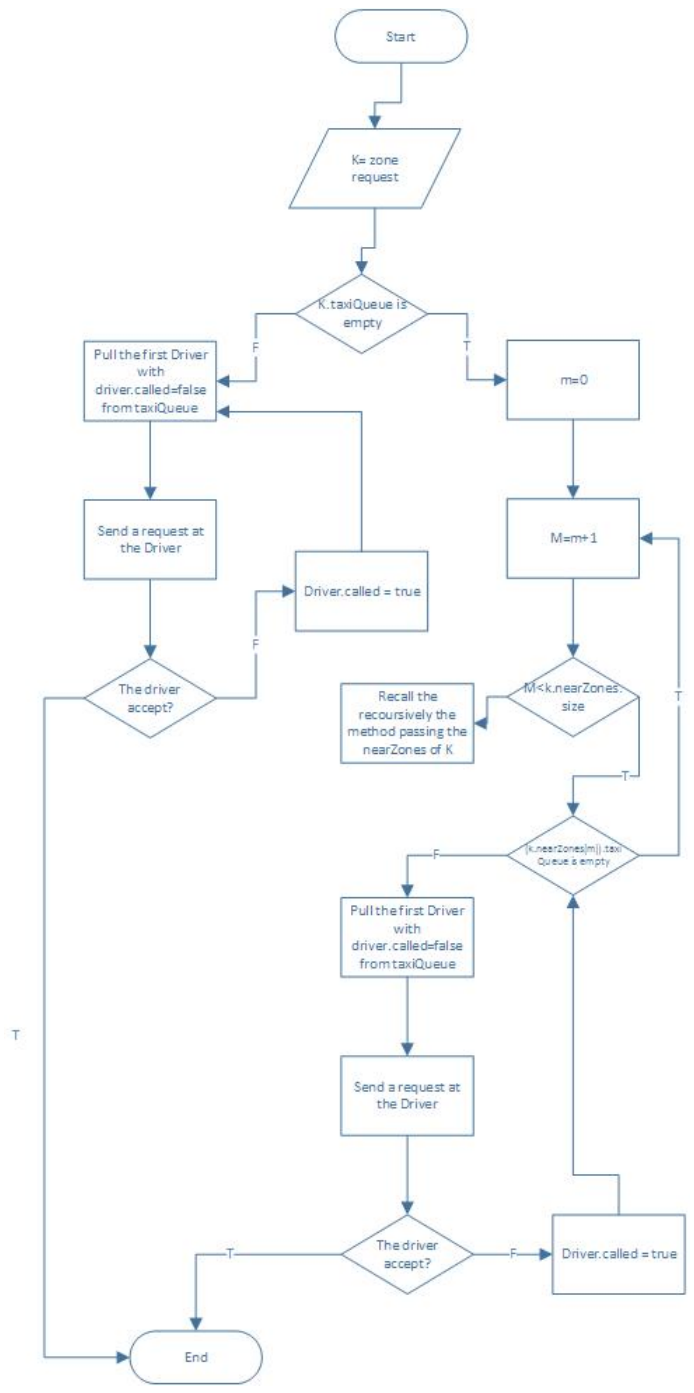
- distance = [km]   Ride's length
- time = [h]   Ride's duration

If for a specific hour and date more initial costs are associated (e.g during an holiday night in which are applied both Holiday and Night fare), in the the formula we will use the most expensive one.

$$TotalCharge = initialCharge + \lceil \frac{space}{spaceCharge} \rceil \cdot (costSpace \cdot spaceCharge) + \lceil \frac{time}{timeCharge} \rceil \cdot (costTime \cdot timeCharge)$$

3.2 Queue Algorithm

We propose the following flowchart for the queue algorithm proposed for the system:


Start
K= zone request
K.taxiQueue is empty
F
T
Pull the first Driver with driver.called=false from taxiQueue
m=0
Send a request at the Driver
M=m+1
Driver.called = true
F
The driver accept?
T
M<k.nearZones.size
Recall the recoursively the method passing the nearZones of K
T
(k.nearZones[m]).taxi Queue is empty
F
Pull the first Driver with driver.called=false from taxiQueue
Send a request at the Driver
T
The driver accept?
F
Driver.called = true
T
End

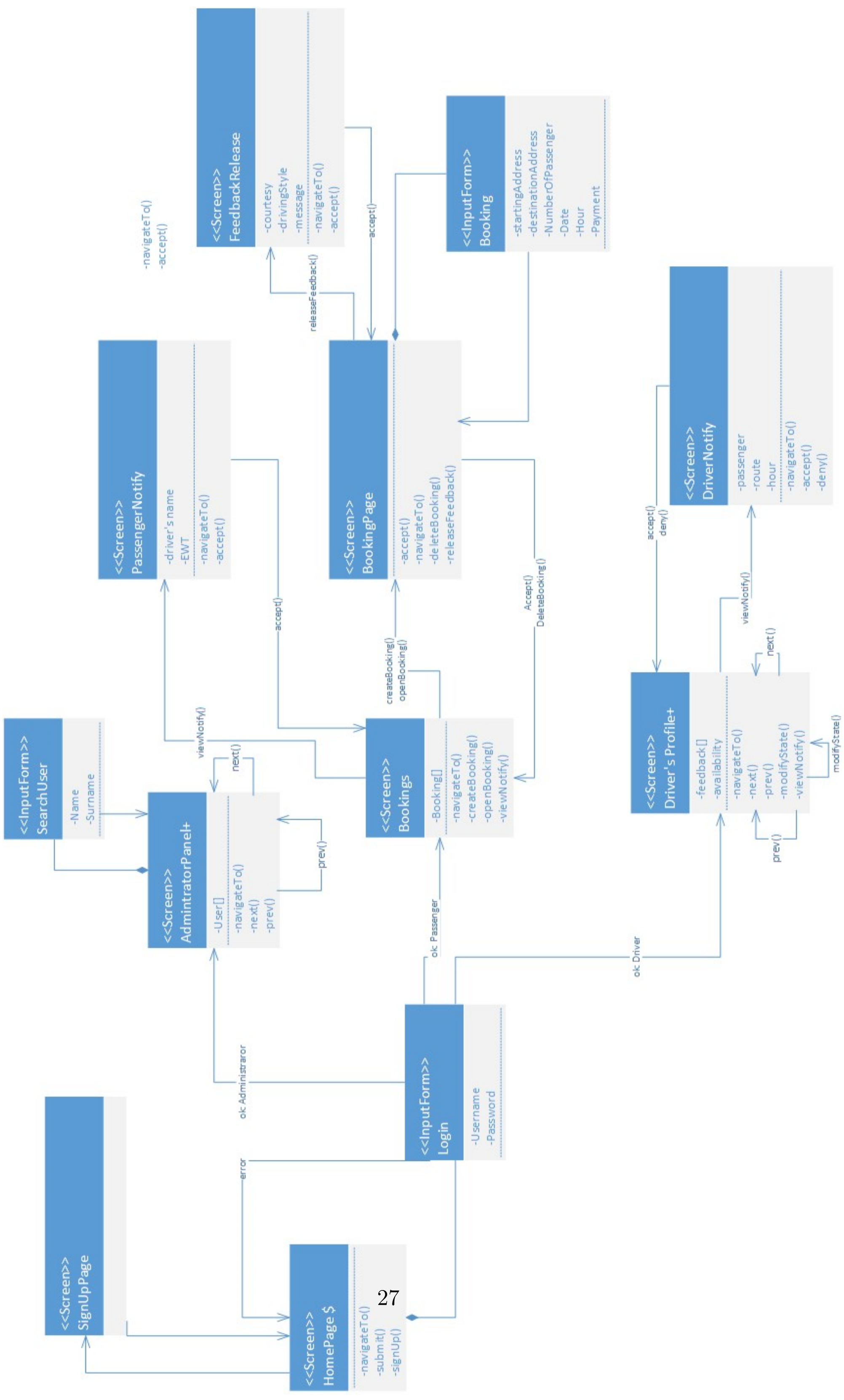
3.3 EWT elaboration

The EWT will be simply provided by the Open Street Map API: it's the time from the GPS coordinates of the driver that accepts the request and the relative passenger.

`http://wiki.openstreetmap.org/wiki/Routing`

4 User Interface Design

In this chapter we analyze the User's experience of myTaxiService by representing the usability of the application with the following UX Diagram.



<<Screen>> HomePage $
-navigateTo()
-submit()
-signUp()
<<Screen>> SignUpPage
<<InputForm>> Login
-Username
-Password
error
ok Administraror
ok Passenger
ok Driver
<<Screen>> AdmintratorPanel+
-User[]
-navigateTo()
-next()
-prev()
next()
prev()
<<InputForm>> SearchUser
-Name
-Surname
<<Screen>> Bookings
-Booking[]
-navigateTo()
-createBooking()
-openBooking()
-viewNotify()
viewNotify()
accept()
createBooking()
openBooking()
<<Screen>> PassengerNotify
-driver's name
-EWT
-navigateTo()
-accept()
<<Screen>> BookingPage
-accept()
-navigateTo()
-deleteBooking()
-releaseFeedback()
Accept()
DeleteBooking()
releaseFeedback()
<<Screen>> FeedbackRelease
-courtesy
-drivingStyle
-message
-navigateTo()
-accept()
-navigateTo()
-accept()
<<InputForm>> Booking
-startingAddress
-destinationAddress
-NumberOfPassenger
-Date
-Hour
-Payment
<<Screen>> Driver's Profile+
-feedback[]
-availability
-navigateTo()
-next()
-prev()
-modifyState()
-viewNotify()
next()
prev()
modifyState()
viewNotify()
<<Screen>> DriverNotify
-passenger
-route
-hour
-navigateTo()
-accept()
-deny()
accept()
deny()

A visitor begins the navigation in the Homepage and can sign in if is already registered or can sign up in the dedicate page (SignUpPage). After performing the login, the user is redirected to a different page based by the type of User.

A passenger is redirected to the page with the list of his bookings. He can create open an existent booking or create a new one and can manage that from the bookingPage. Here he can fill all the fields for the selected booking or eliminate it. If he selects a past booking, he can release a feedback to the relative driver. The passenger receives the notification about a ride when a taxi driver accept the request.

A taxi driver is redirected to his profile page with the list of feedback received. From this page he can also modify his availability. When a new booking is performed, 10 minutes before the driver receives the relative notify with the passenger name, the route and the hour of the ride and he can accept or deny the request.

An administrator is redirected to the AdministratorPanel where he can search and manage all the users of the platform.

5 Requirements Traceability

In this section we will illustrate how the application's requirements, explained in the RASD document, will be satisfied. For each requirement we provide a link to the relative reference of this document.

5.1 [G1] Allow the registration on the platform

- **Visitor can only log in or register on the platform**
  Passenger pool in the BPMN diagram - 2.5.1

5.2 [G2] Allow user to log-in on the platform

- **Passenger must be already registered**
  Passenger pool in the BPMN diagram - 2.5.1

5.3 [G3] Allow user to book a new trip

- **Passenger must insert a starting address, a destination address, number of passengers and date of the booking.**
  UX Diagram - 4
- **Passenger must confirm the booking.**
  UX Diagram - 4
- **Starting address and destination address must be different.**
  Sequence Diagram → Booking creation → validateSubmission method - 2.5.2.1

5.4 [G4] Allow user to modify one of his booked trip

- **User must select one of his future booking**
  UX Diagram → Bookings → openBooking() - 4
- **User can modify a booking until two hours before the ride**
  Sequence Diagram → Booking Editing → isTimeExpired() -2.5.2.2

5.5 [G6] Allow user to receive a cost preview for a selected route

EWT Algorithm - 3.3

5.6 [G8] Allow user to release a feedback on a specic taxi driver

Class diagram → Model → association ride - feedback - 2.7.1

5.7 [G9] The application will notify taxi driver for a new ride

Sequence Diagrem → Booking Management . 2.5.2.4

5.8 [G10] The application will notify taxi driver for a new ride

UX Diagram - 4

5.9 [G11] Allow taxi driver to accept or deny ride's requests

UX Diagram - 4

5.10 [G12] Allow administration to inspect all users' profiles

UX Diagram - 4

6 References

7 Appendix

7.1 Software and tool used

In order to redact this document we have used the following softwares:

- ∗ Name: Visio
  ∗ Version: 2013
  ∗ Scope: Draw diagrams
  ∗ Reference: `https://products.office.com/it-it/visio/`
- ∗ Name: draw.io
  ∗ Version: -
  ∗ Scope: draw diagrams
  ∗ Reference: `https://www.draw.io/`

7.2 Working Hours

The Design Document was written in more or less 50 hours divided between the two elements of the group:

- Massimiliano Paci: 25 hours
- Giovanni Patruno: 25 hours

Each element of the group didn't work on specific sections but worked on the whole document