Politecnico di Milano
A.Y. 2015/2016

My Taxi Service

Design Document

Bernardis Cesare matr. 852509 Dagrada Mattia matr.852975

December 4, 2015

Contents

1 Introduction

1.1 Purpose

The main goal of this document is to describe the system in terms of architectural design choices, going in details about the architectural styles and patterns, and also to show how these are implemented via pseudo-code utilizing meaningful examples.

1.2 Scope

MyTaxiService is a system designed in order to improve the user experience of a taxi service and also improving the quality of life of taxi drivers. It was decided to design both a web application and a mobile application for the customers. These applications allow customers to easily request taxis and also to reserve a taxi ride, even if this feature is only for registered users. Taxi drivers on the other hand have access to the system only via mobile application. The system sends them requests that they can either accept or refuse and they are also able to set themselves available or unavailable. The system holds taxi queues in different areas of the city and depending on the position of the customers will forward requests to a specific queue instead of another.

1.3 Reference Documents

We used as starting point

- MyTaxiService RASD
- Structure of the design document

1.4 Document structure

This document is structured as following:

1. **Introduction**: this section represents a generic description of the document.
2. **Architectural Design**: this section gives informations about the architectural choices, showing also which styles and patterns have been selected.
3. **Algorithm Design**: this section focuses on the definition of the most relevant algorithmic part of this project.
4. **User Interface Design**: provides an overview on how the user interface(s) of the system will look like.

5. **Requirements Traceability**: explains how the requirements defined in the RASD map into the design elements.

6. **References**: shows from which documents we take informations.

7. **Other Info**: contains the hours of work and the tool used to write this document.

2 Architectural Design

2.1 Overview

In this section we will talk about the system in its complex. We will show how we think that it should work, in particular we will present which should be its structure, the most important components and how they interact each other. We thought our system with a four tier architecture, in order to be compatible with a JEE Architecture implementation.

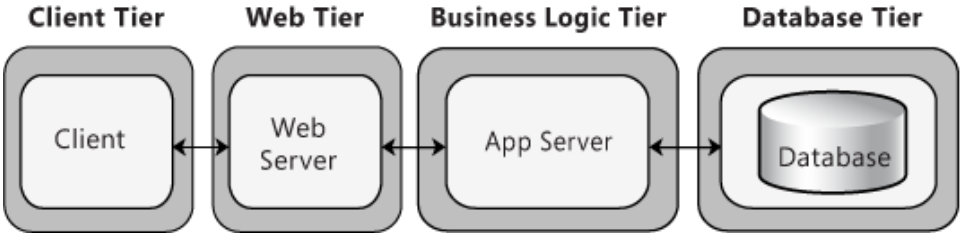

Figure 1: Four tier architecture

However, we do not want to limit the ways it could be implemented, so we will present the JEE as an example, **not as a constraint**. JEE is useful because provides some services "on the shelf" (ready to use) and we recommend to reuse the existing code to avoid a useless waste of time, but the possibility is always open for a more specific, and performing, implementation of the system.

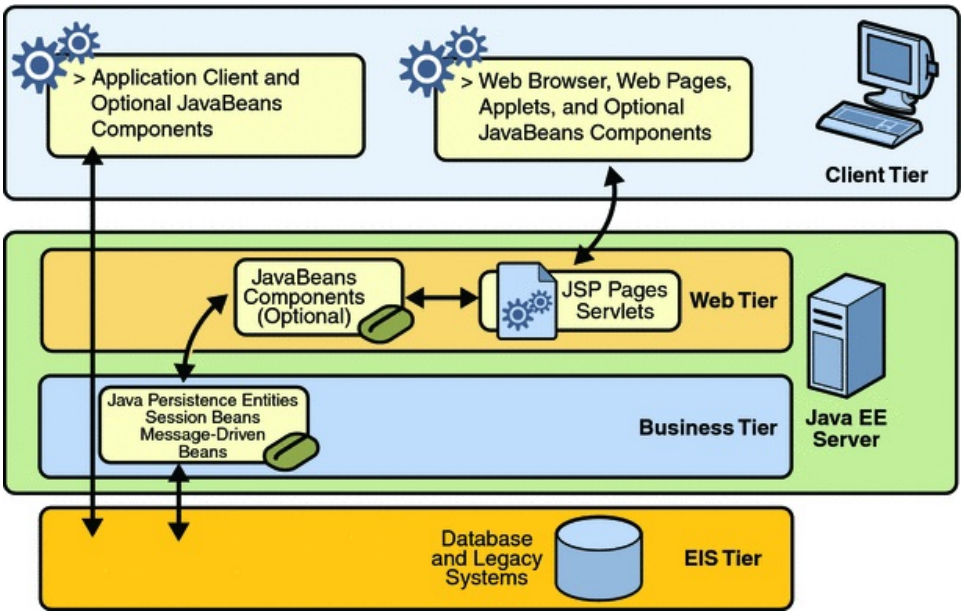

Figure 2: Java Enterprise Edition architecture

Client tier contains Application Clients and Web Browsers and it is the layer that interacts directly with the actors. In our case this tier contains both of them, because we have a usual web application and a mobile app.

Web tier contains the Servlets and Dynamic Web Pages that needs to be elaborated. It could also include some optional JavaBeans. This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier correctly formatted

Business tier contains the Java Beans (the business logic of the application) and Java Persistence Entities which represent the physical data of the system.

EIS tier contains the persistent data source and destination. It is another way to refer to the database of our system.

2.2 High Level Components and their interaction

We start introducing how we thought the four tier architecture of our system. The top block represents the Client, where the application or the web

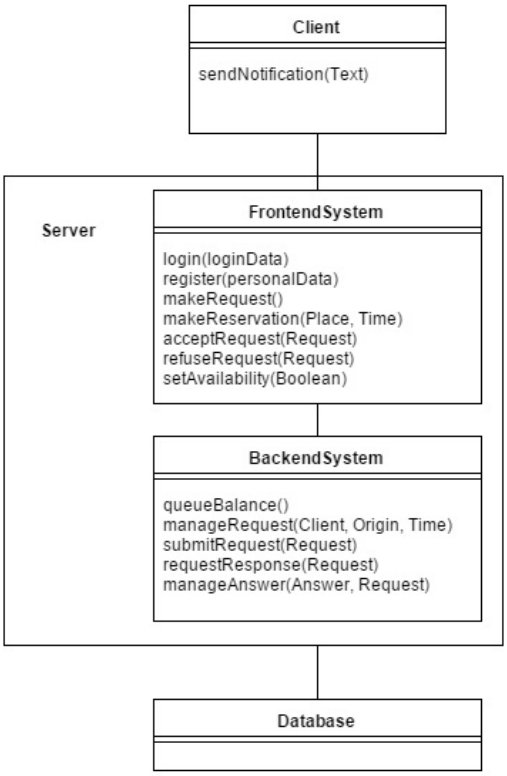

Figure 3: High Level Component representation

browser runs with our clients. The Client is connected to the Server of our application, which is divided in two subsystems:

Frontend which represents the part of our system that provides to the client all the necessary functions to perform all the functionalities that are required by the application

Backend which represents the beating hearth of our system, where all the requests, reservations, and the services that our system should provide, are created and maintained.

The end point is the Database, where all the persistent data is stored for a long term availability.

2.3 Component view

This is a diagram that represents the whole system. We are focusing on the global components and their connections, providing also a preview of the interfaces that the various components provide for the interaction. A more detailed explanation of the single components follows the diagram.

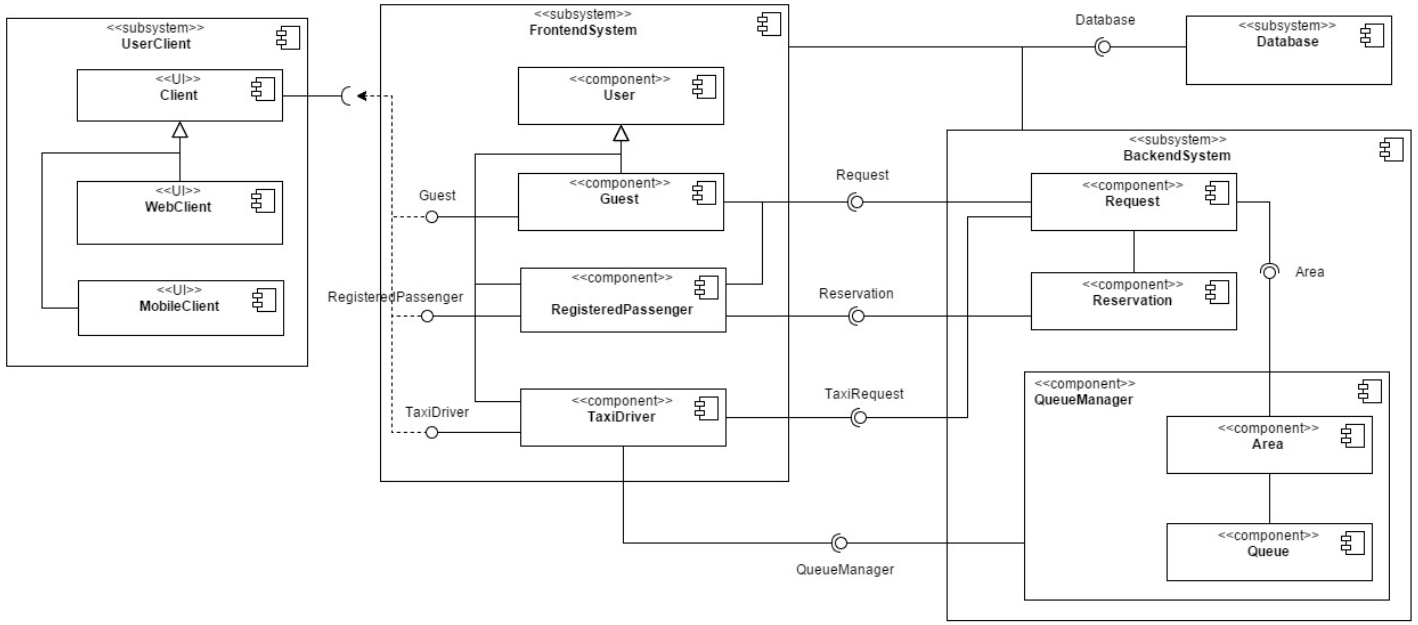

Figure 4: Component Diagram

2.3.1 Client

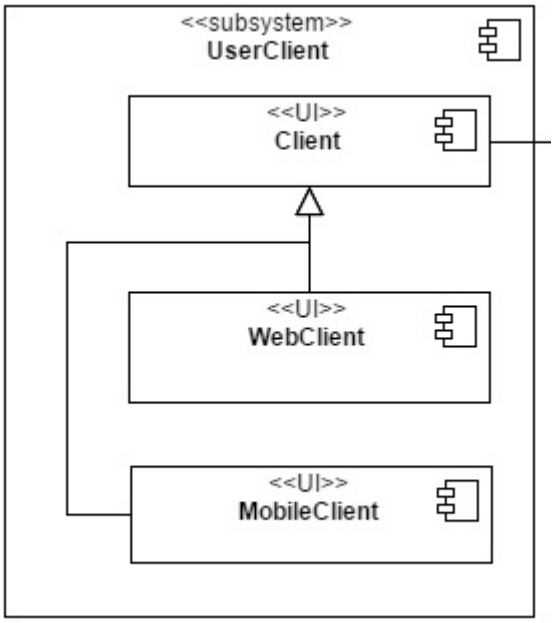

Figure 5: Client components

In this subsystem we have the components in direct contact with the user. It can be considered the user's "access interface" of our system.

2.3.1.1 Client

user interface represents the generic type of client that interact with the system. It will most likely be an abstract class extended by WebClient and MobileClient.

2.3.1.2 WebClient, MobileClient

user interfaces represent a generalization of the Client since they both share the same functionalities except for the fact that a Taxi Driver can only use the MobileClient to interact with the system. They main difference between these two user interfaces is the different type of communication protocols they use to interact with the system.

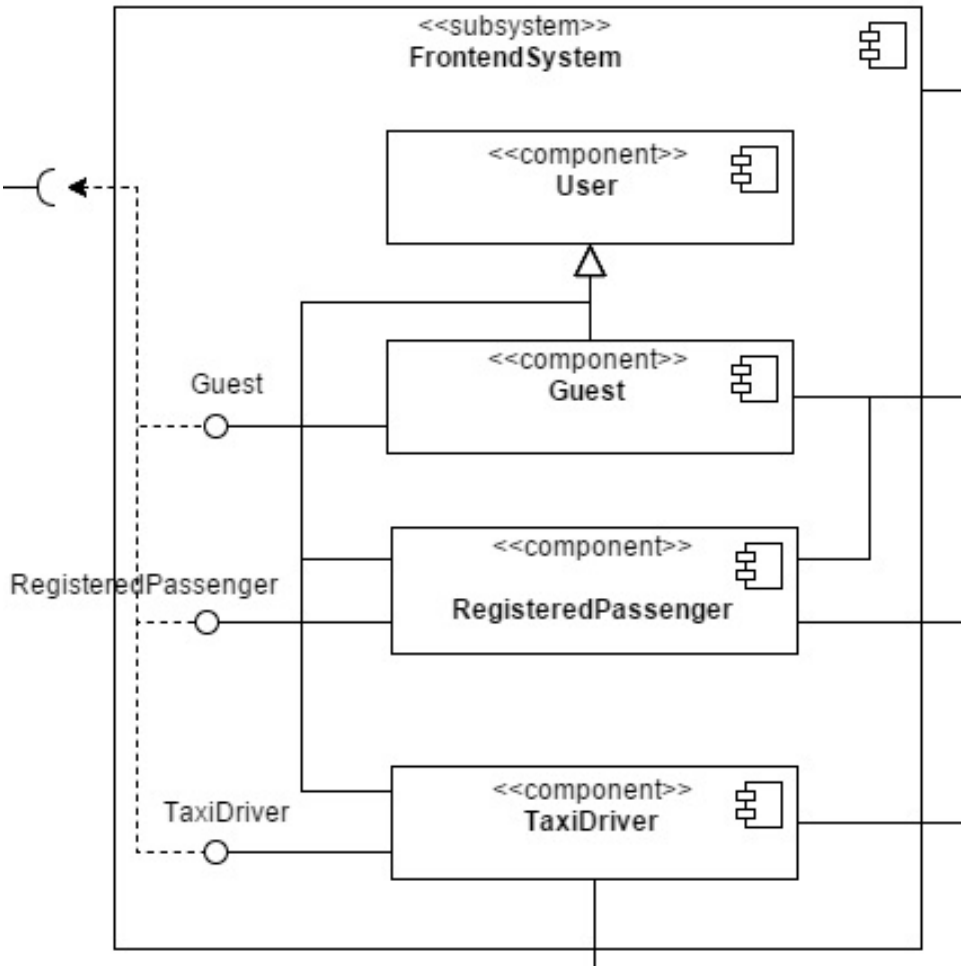

Figure 6: Backend components

2.3.2 Frontend

This subsystem represents the access to the real functionalities of our system. The application requires a different type of user (that depends on the login) and, through the components of the Frontend, calls the services of the backend.

2.3.2.1 User

component is the generic type of user that can be created by the system. It is the representation of the client from the server point of view. It will most likely be an abstract class extended by Guest, RegisteredPassenger and TaxiDriver. User component is stateful since it is strictly related to a

Client but once the session is over its instance is destroyed.

2.3.2.2 Guest

component is the representation from the server point of view of one of the possible actors that interact with the system. Guests can only perform requests. Guest component is stateful, as for the User, since it is strictly related to a Client but once the session is over its instance is destroyed.

2.3.2.3 RegisteredPassenger

component is the representation from the server point of view of one of the possible actors that interact with the system. RegisteredPassengers can perform requests or reservations and their data is stored in the database. This is due to the fact that a client that is a registered passenger has filled a registration form and has to log in before being able to access all these features. RegisteredPassenger component is stateful, as for the User, since it is strictly related to a Client but once the session is over its instance is destroyed.

2.3.2.4 TaxiDriver

component is the representation from the server point of view of one of the possible actors that interact with the system. TaxiDrivers can receive requests from the server and can accept or refuse them. They also have a status which can be available or unavailable and affects their presence in a queue. Their data is stored in the database as for the RegisteredPassengers. A client that is a TaxiDriver has to log in first to access all these features. TaxiDriver component is stateful, as for the User, since it is strictly related to a Client but once the session is over its instance is destroyed.

2.3.3 Backend

In this subsystem we have placed all the components (mainly entities) that we do not want to let the client interact directly. It is in direct communication with the frontend, responsible of the user management, and the database. This is practically the core of our system.

2.3.3.1 Request

This is a stateful component. A User (Guest or RegisteredPassenger) can ask to create an instance which will be strictly related to the User himself and to the TaxiDriver who is asked to satisfy the request. A record in the database will be created to keep track of the operation, but when a taxi driver accepts or when the user closes the session, the request has no sense

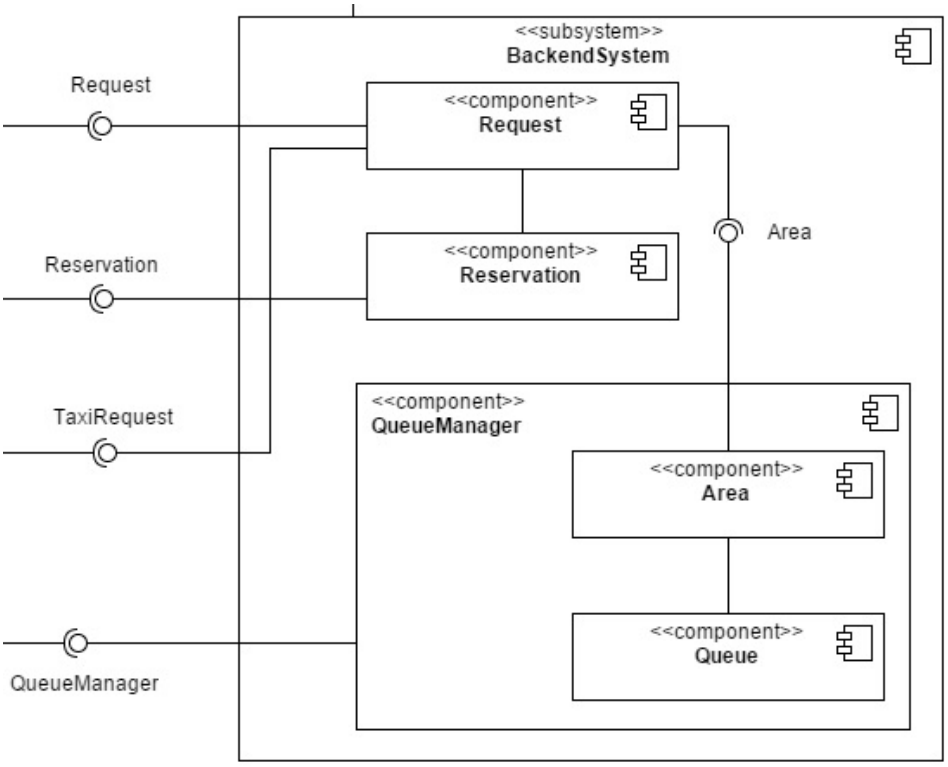

Figure 7: Backend components

to exist anymore. In case a request is created automatically by a reservation, it will be related to the same user that booked it.

2.3.3.2 Reservation

This is a stateful component. A RegisteredPassenger can ask to create an instance which will be strictly related to the User himself. A record in the database will be created to keep track of the operation, but when the User closes his session, the reservation has no sense to exist anymore. At the right time a job will be launched, creating a Request (even if the original instance does not exist anymore, the informations are stored in the database).

2.3.3.3 QueueManager

This is a singleton component. Its interface is used by all the taxi drivers to interact with the areas and the queues. It is also responsible for the maintenance and the balancing of the queues.

2.3.3.4 Area

This component is stateless. It is not connected to a single User, but, simply because it represents an Area, it has to serve all the users that require that area in the same way, there is no reason to relate the life of an area to the user's one. The main goal of this component is to manage the access to the one and only queue it contains.

2.3.3.5 Queue

This is not truly an independent component, because we want to keep it strictly connected to the area, but we also want to be able to manage it without passing through the Area component all the times. It is easier fort he QueueManager to operate with a direct access to the queues. It is also sometimes useful to go back to areas from queues, so we keep a two way connection between them.

2.3.4 Database

Figure 8: Backend components

2.3.4.1 Database

component is the database of our system. It stores all the important data that are often required by the Backend and the Frontend servers. Database component is singleton, it is created once and it is used during the whole life cycle of the system.

2.4 Deployment view

We provide a sight of how our components should be deployed in the various physical devices that compose the infrastructure of our system. In this case we assume that our Frontend and Backend subsystems run on the same server, so they are deployed in the same block. They could also be executed on different machines, the developer should simply define how the two subsystems should connect each other. On the application server will also be available the web pages accessible through a web browser.

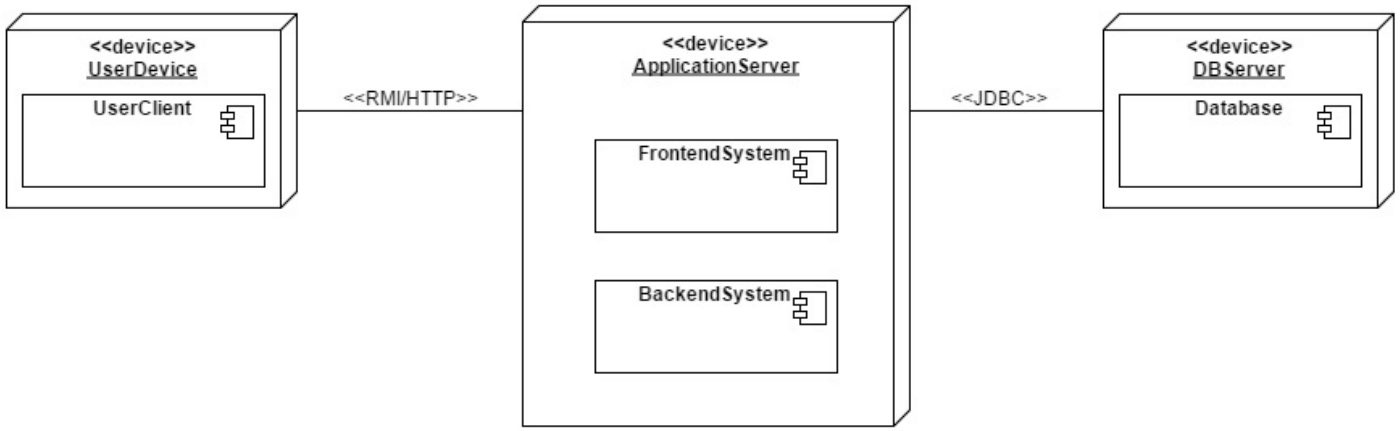

Figure 9: Deployment Diagram

2.5 Runtime view

This is a "photograph" of the system during a hypothetical execution. The instances of the components and their interconnections are well visible inside the respective blocks where they are executed.

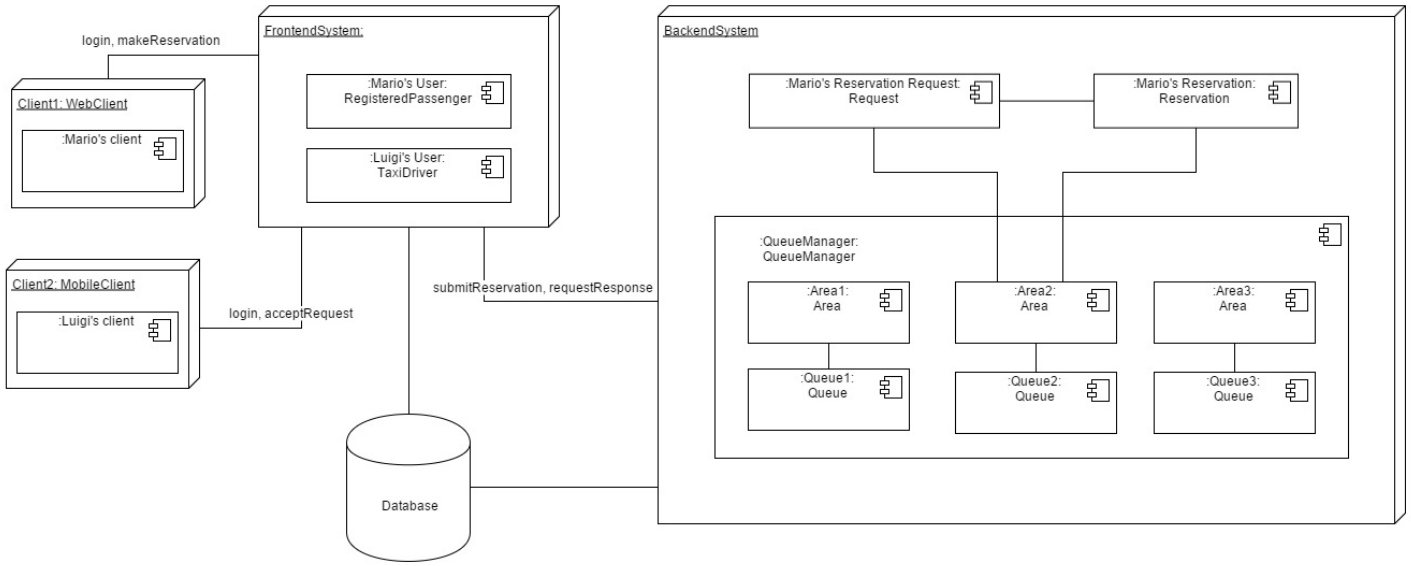

Figure 10: Runtime Diagram

2.6 Sequence Diagrams

2.6.1 Guest submit request

This diagram shows how the system works when a user approaches the application a tries to submit a request as a guest user (not logged). We break the representation at the makeRequest method call because we will show how it works only one time in Figure 13.

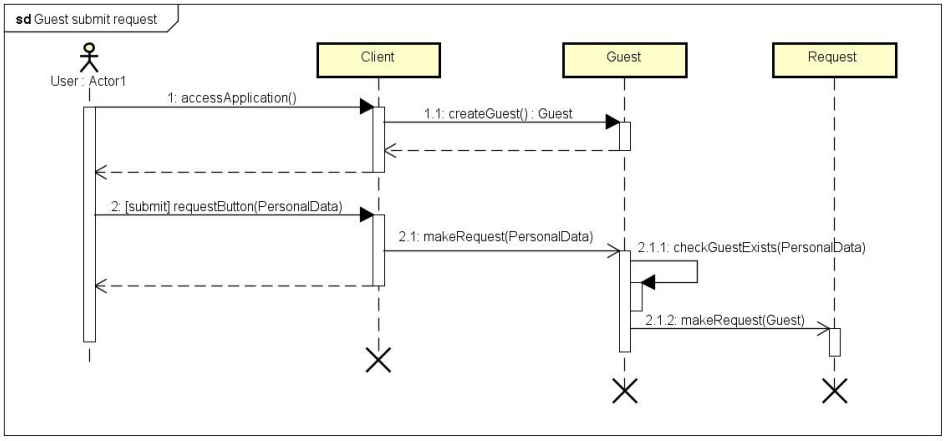

Figure 11: Guest submit request

2.6.2 RegisteredPassenger submit request

This diagram shows the behaviour of the system when a registered passenger logs into the system (so we have also a sequence diagram example of the login process) and makes a reservation as a Registered Passenger. As before, we break the representation at the makeRequest method call because we will show how it works only one time in Figure 13.

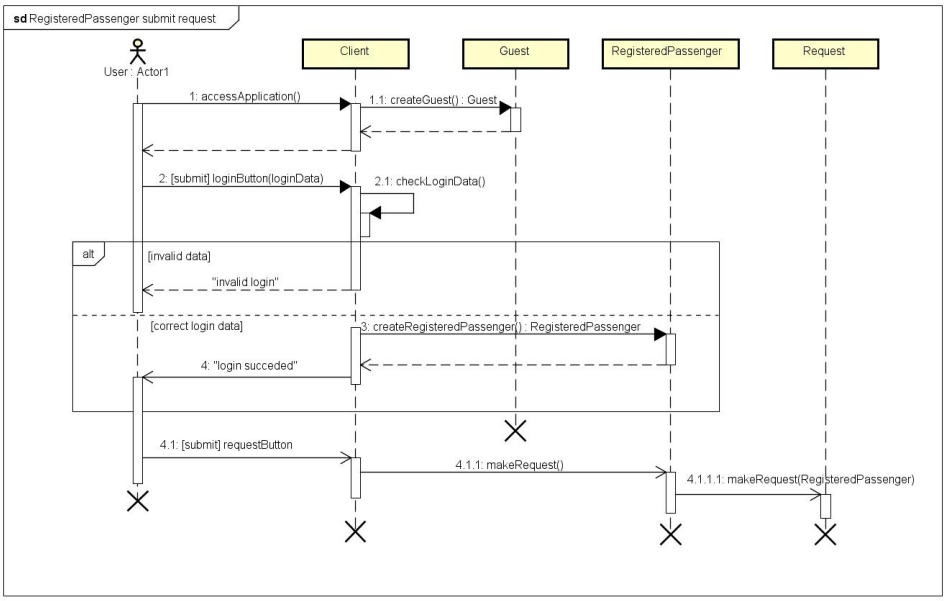

Figure 12: RegisteredPassenger submit request

2.6.3 Make Request

This is how the system processes a Request submission.

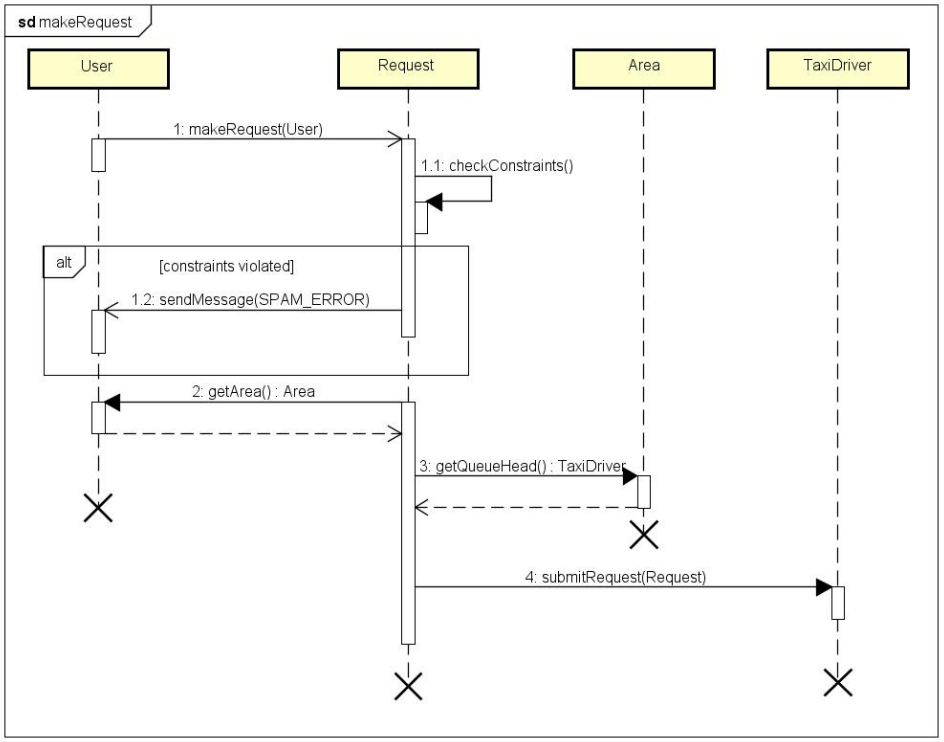

Figure 13: Make Request

2.6.4 Request response and Answer management

When a request has been created and the first notification is sent to a taxi, the system should manage the answer of the interested TaxiDriver bringing the necessary consequences to the Request (in order to finally satisfy it) and the TaxiDriver itself.

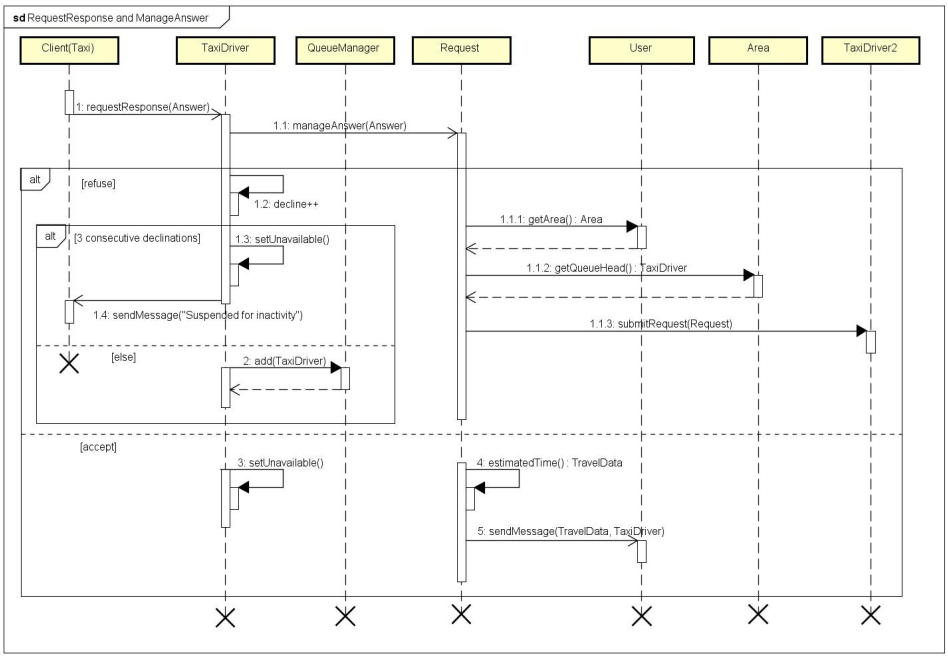

Figure 14: Request response and answer management

2.6.5 Reservation

This is how the system processes a Request submission. As usual, we refer at the makeRequest diagram at Figure 13 for the creation and management of the Request relative to this Reservation.

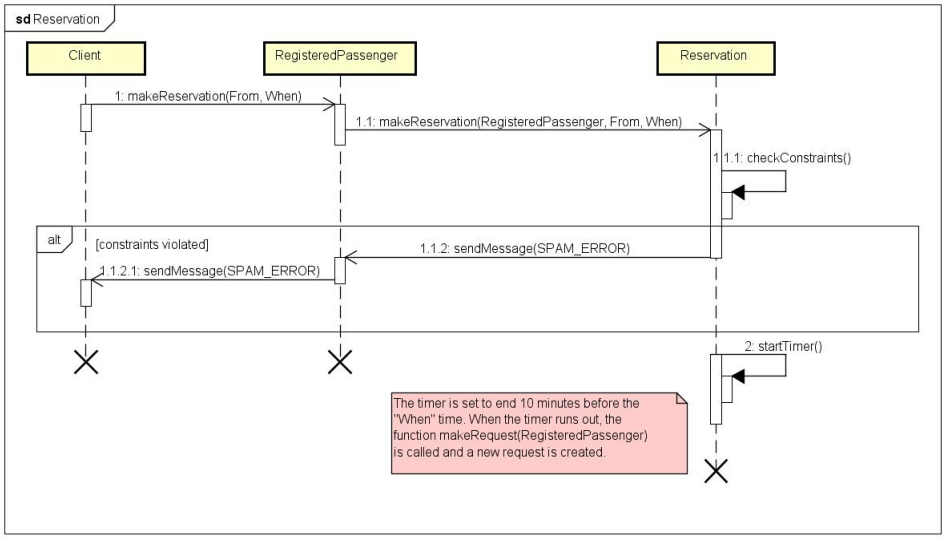

Figure 15: Reservation

2.7 Component interfaces

2.7.1 Guest

This is the basic interface assigned to client when the user accesses the application. It provides methods to perform all the operations that a "Guest" user can do:

- **register**: this method gives the possibility to an unregistered passenger to submit his personal data to perform a registration. If the registration succeeds, he will be stored in the database and will be able to log into the environment and be considered a registered passenger.

  Input Personal Data required for the registration

  Output The result of the registration (success or failure in case of error)

- **login**: this method gives the possibility to login the environment to be considered a registered passenger. If this method succeeds, the Client will automatically substitute its Guest interface with a Registered Passenger interface related to the user that just performed the login.

  Input Login Data required for the login

  Output The result of the login (success or failure in case of error)

- **makeRequest**: this method gives the possibility to a guest to make a single request. Some personal data and the consent for the automatic position recovery are required for the submission of the request. It also checks that all the constraints are respected before creating the Request object.

  Input Personal Data and consensus

  Output Nothing is returned directly, but the client will receive notifications about the request status.

2.7.2 RegisteredPassenger

This interface substitutes the Guest one in the client when the login is successfully completed. It provides all the functionalities that a Registered Passenger can exploit.

- **makeRequest**: like in the Guest, this method gives the possibility to make a single request. No personal data is required for the submission, because the system will automatically use the informations stored in the database during the registration process. This method also checks that all the constraints are respected before creating the Request object.

Input Nothing

Output Nothing is returned directly, but the client will receive notifications about the request status.

- **makeReservation**: this method gives the possibility to a logged user to make a single reservation. For a reservation some additional informations are required, that must be provided at the submission. This method also checks that all the constraints are respected before creating the Reservation object.

  Input The origin time and place where the taxi ride will begin

  Output Nothing is returned directly, but the client will receive notifications about the reservation and the consequent request statuses.

- **deleteReservation**: this method gives the possibility to delete a reservation. The time constraints are checked and, if all the test are passed and no Request related to this reservation already exists, the record is deleted from the database. No Requests for this Reservation will be forwarded.

  Input The Reservation that has to be deleted

  Output Nothing is returned directly, but the client will receive notifications about the result of the operation.

2.7.3 TaxiDriver

This interface is provided by the TaxiDriver. With this interface a client is able to access the system via login. In particular this interface will be used only by mobile clients since the taxi drivers can't access the system via web. Once logged in they will be able to set themselves as available or unavailable and to see, accept or refuse requests.

- **login** this method gives the possibility to login the environment to be considered a taxi driver and will access the taxi driver interface

  Input Login Data required for the login

  Output The result of the login (success or failure in case of error)

- **setAvailable** this method is used to change the taxi driver status to available

  Input Nothing

  Output Nothing is returned but the taxi driver will see on his application that his status has changed

- **setUnavailable** this method is used to change the taxi driver status to unavailable

  Input Nothing

  Output Nothing is returned but the taxi driver will see on his application that his status has changed

- **submitRequest** this method is used to send a request to a taxi driver

  Input Request that will be sent to the taxi driver

  Output The result of the submit (success or failure in case of error)

2.7.4 Request

This interface is provided to the User instances to create, and eventually interact, with a request.

- **makeRequest**: this method receives a User (a Guest o a RegisteredPassenger instance) and creates an instance of a Request, launching also all the necessary processes to satisfy it.

  Input User

  Output A Request instance

2.7.5 Reservation

This interface is provided to the User instances to create, and eventually interact, with a reservation.

- **makeReservation**: this method receives a User (a Guest o a RegisteredPassenger instance) and the place and time informations necessary for the creation of a Reservation and instances a Reservation object, launching also all the necessary processes to satisfy it. This method fundamentally launches the makeRequest functionality at the right time.

  Input User, When and Where the request will take place

  Output A Reservation

2.7.6 TaxiRequest

This interface is provided by the Request. With this interface TaxiDrivers receive requests that they can accept or refuse them.

- **requestReply** this method is used by the taxi driver to accept or refuse the request they have received

  Input Request that the taxi driver has received and his answer

  Output The result of the reply (success or failure in case of error)

2.7.7 QueueManager

This interface is provided by the QueueManager. With this interface TaxiDrivers can set their status as available or unavailable, affecting their presence in the queue. This also mean that the QueueManager can add or remove a TaxiDriver from the queue.

- **add** this method is used by the QueueManager to add a taxi driver to a queue

  Input TaxiDriver that will be added to the queue

  Output the result of the operation (success or failure in case of error)

- **remove** this method is used to remove a taxi driver from a queue

  Input TaxiDriver that will be removed from the queue

  Output the result of the operation (success or failure in case of error)

2.7.8 Area

This interface is provided by the Area. With this interface the Request can retrieve Area's informations and get the first available taxi from queue in order to submit him a request.

- **getQueueHead** this method is used to retrieve the first taxi in a queue

  Input Nothing

  Output The first taxi driver from the queue corresponding to the area in use

2.7.9 Database

This interface is provided by the database and it is used by the Backend and Frontend servers. With this interfaces Backend and Frontend servers are able to interact with the database, storing and retrieving data about for example registered passengers or reservations. There is no particular function since it will depend on how the database is effectively implemented.

2.8 Architectural styles and patterns

We want suggest some some styles and patterns that we decided to adopt in our system, but that have not been explained clearly yet.

- The first, and probably the most relevant, is the four tier architecture. Our system is designed to support a JEE architecture with a thin client on the user device, which is used only as a interface, and a, relatively, fat server that processes all the operations.

- An MVC approach has been followed. The user interfaces in the User-Client (view) are used only for data exploration and command submission by the user. The backend operates and processes (controller) the requests coming from the user interfaces and interacts with the persistent and dynamic (but global, like areas and queues) data belonging to the model.

- JMS is a good way to solve the "messaging problem". JMS is a services provided by JEE in its architecture. Its goal is to manage the point-to-point message delivery in an asynchronous way, simply specifying who has to receive a notification. A similar approach can be reproduced also without the JMS constraint.

- There are two patterns that could be useful during implementation:

  Factory Method useful for components that must be continuously created, maybe also by different components or as a result of different commands, like Users (Guest, ecc.), Requests and Reservations. This allows a stronger control on the instance creation.

  Singleton useful for components which needs only one instance in the whole system.

3 Algorithm Design

In this section are presented some algorithms that we consider fundamental to understand how our system should work. We will not provide the code of the most common functionalities, but we have selected two of the most significant for our project.

3.1 Request management

This algorithm is focused on the request managment.
In the first procedure ManageRequest, first it checks if the request by that specific client is valid, meaning that the client hasn't done any request in the last 30 minutes. If the request is not valid then it is refused with an error notification sent to the client, otherwise the request is instantiated and stored in the database. From the request's info it retrieves an area and then the first taxi available, to which the request is sent.
In the second procedure SubmitRequest there's a timer indicating the window of time that the taxi driver has to answer the request and the notification that is showed to the taxi driver.
In the third procedure RequestReply it is checked the answer given by the taxi driver. In case of negative answer, the number of declined answers by that taxi driver in incremented by 1 and when it reaches the third consecutive decline the taxi is set unavailable, otherwise it is added in queue again.
In the last procedure ManageAnswer it is checked again the answer. In case of negative answer the operations to retrieve the first taxi in the queue are done again, otherwise the request confirmation is sent to the client which contains the estimated arrival time and the taxi ID.

3.1.1 Pseudocode

```
1:  procedure MakeRequest(user)
2:      if !validRequest(user) then
3:          sendMessage(SPAM_REQUEST_ERROR, user)
4:      else
5:          rqst ← new Request(user)
6:          storeRequest(rqst)
7:          area ← retrieveArea(origin)
8:          taxi ← area.queue.remove(0)
9:          taxi.submitRequest(rqst)
10:     end if
11: end procedure
```

```
1: procedure SubmitRequest(request)
2:     timer ← newTimer(60)
3:     timer.start()
4:     sendMessage(rqst, taxi)
5: end procedure
```

```
1: procedure RequestResponse(answer, request)
2:     request.manageAnswer(answer)
3:     if answer ≡ REFUSE then
4:         taxi.declined + +
5:         if taxi.declined ≡ 3 then
6:             taxi.setUnavailable()
7:         else
8:             queueManager.add(taxi)
9:         end if
10:    else
11:        taxi.setUnavailable()
12:    end if
13: end procedure
```

```
1: procedure ManageAnswer(answer)
2:     if answer ≡ REFUSE then
3:         area ← retrieveArea(rqst.origin)
4:         taxi ← area.queue.remove(0)
5:         taxi.submitRequest(rqst)
6:     else
7:         user.sendConfirmation(estimatedTime(), taxi)
8:     end if
9: end procedure
```

3.2 Queue balancing

This algorithm shows the mechanism that every two minutes balances the area queues. The idea is that a taxi may move from its position at any time, for any reason, and, perhaps, pass through other areas than the one he belonged to when he was first inserted into the system. So a simple job running this algorithm can keep the system up to date and avoid inconsistencies (like sending a notification to a taxi out of the interested area).

3.2.1 Pseudocode

```
1: for all TaxiDrivers available as TaxiDriver do
2:     actualArea ← TaxiDriver.Area
3:     oldArea ← TaxiDriver.Queue.Area
4:     if actualArea ≠ oldArea then
5:         oldArea.extract(TaxiDriver)
6:         actualArea.insert(TaxiDriver)
7:     end if
8: end for
9: criticalAreas ← List
10: for all Areas as Area do
11:     if Area.Queue.size ≤ Area.minCriticalSize then
12:         criticalAreas.add(Area)
13:     end if
14: end for
15: surplusTaxis ← List
16: for all Queues as Queue do
17:     for all TaxiDrivers with index ≥ Queue.Area.maxCriticalSize
    do
18:         surplusTaxis.add(TaxiDriver)
19:     end for
20: end for
21: notification ← "In the areas ".criticalAreas.toString()." we have a taxi
    shortage. Go there and get back to work!"
22: for all surplusTaxis as TaxiDriver do
23:     TaxiDriver.sendNotification(notification)
24: end for
```

3.2.2 Clarifications

The first section of the algorithm updates the area queues, moving taxis from the areas they belonged last time they were inserted to the one they belong now, only if they are different (we assume that *TaxiDriver.Area* retrieves

the actual position of the taxi and returns the corresponding area). After the update, the system tries to balance his queues where the values are out of some critical values. It is simple: if the length of an area queue is under a *minCricitalSize*, that area needs to be "filled" with taxis. If the length of an area queue is above a *maxCricitalSize*, that area has too many taxis without a reason an the last taxis in those queues can be moved elsewhere. So the system notifies these *surplusTaxis*, advising them to move where they are really needed.

4 User Interface Design

4.1 User Experience

4.1.1 Traditional User

This is the UX from the point of view of a customer. From the homepage he can perform a request as a guest, adding some basic personal information and giving the consensus of using his position. He can also register to the service, compiling a registration form, or login in if already registered. Both choices lead to a new page in which the now registered passenger can perform requests without adding any extra information or reservations, filling the corresponding form.

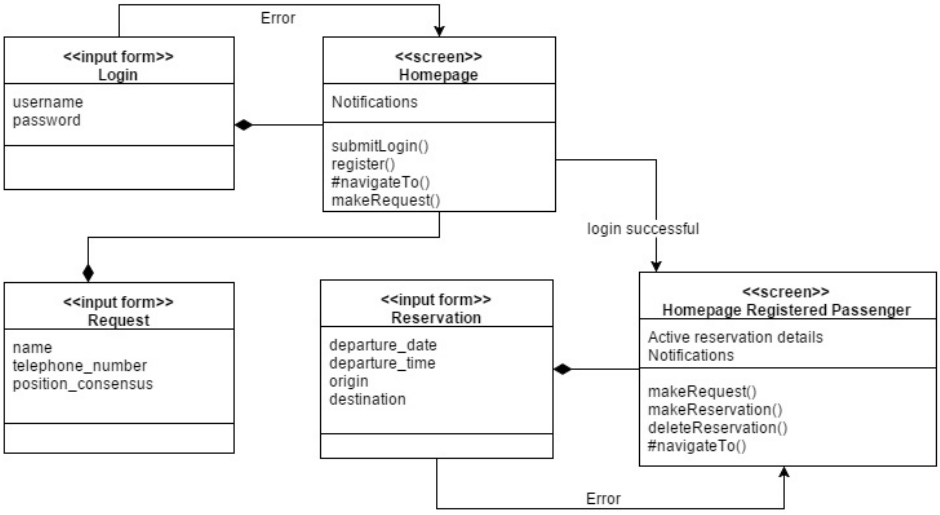

Figure 16: User Experience - User

4.1.2 Taxi Driver

This is the UX from the point of view of a taxi driver. From the homepage he still is considered a guest so he has access to the same functionalities that we have explained for the UX above. Once loggend in he has a completely different interface with respect to the customer. He can set his status as available or unavailable, see if there is a pending request and in case accept or refuse it.

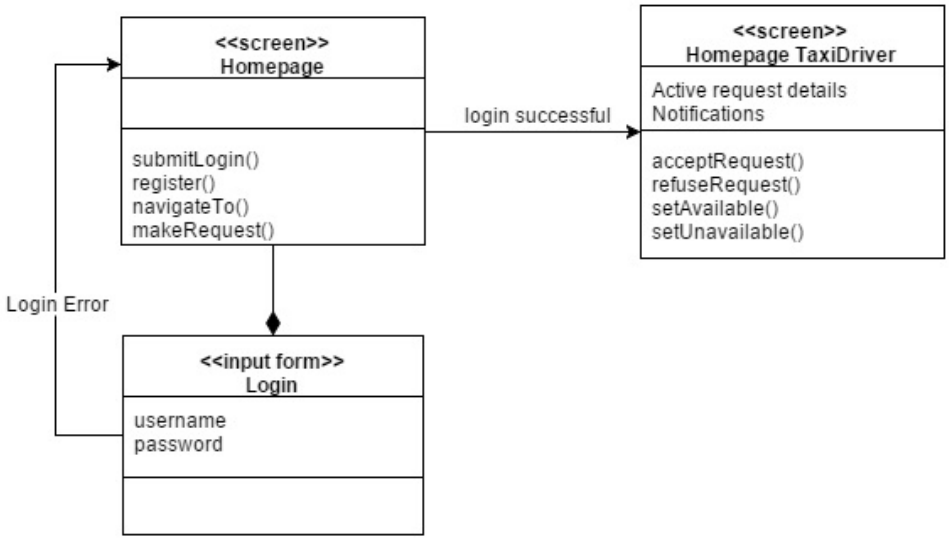

Figure 17: User Experience - Taxi Driver

4.2 Mockups

Wr are not putting any mockup in this section since we already have them in the RASD, in particular in a subsection of the External Interface Requirements section, User Interfaces.

5 Requirements Traceability

In this section we will go over how what is written in the Specific Requirements section in the RASD is connected to some design element defined in this document.

5.1 Functional Requirements

The functional requirements that are present in the RASD are connected first of all to our component view, in which it is clear where each requirement is satisfied and by which component (make reference to subsection 2.2 , and also better explained in section 2.6 . They are also present in the sequence diagrams showing even more in details some functionality.

5.2 API

The API of GoogleMaps are used by the components of the Backend system, in particular by the Area. The main application is retrieving the taxi position and then assign that taxi to the correct area, but also calculating the estimated time of arrival of a taxi to the origin position of a request.

6 References

In order to write this Design Document we use as reference:

- *IEEE standard on Design Descriptions*
- *IEEE 1417*
- *Other Design Documents*

7 Other Info

7.1 Hours of work

For each component follows an approximate indication of how much time was spent on the realization of this document

Component	Hours
Bernardis Cesare	32
Dagrada Mattia	30

7.2 Tools

We used various tools to develop this document:

- LaTeX 2_ε and TeXMaker editor 2.10.4
- Astah Professional 7.0.0
- Draw.IO

7.3 Version 2.0

We changed few things in the interfaces section since some functionalities were assigned to the wrong interface.

7.4 Version 2.1

Some mistakes in the descriptions of the subsystems have been fixed.