Politecnico di Milano
Scuola di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Computer Science and Engineering
A.Y. 2015-2016

Software Engineering 2 Project
"myTaxiService"
Design **D**ocument

December 4, 2015

Principal Adviser: Prof. **Di Nitto**

Authors:
Davide Fisicaro 854043
Gianmarco Giummarra 852667
Salvatore Ferrigno 850130

Contents

1 Introduction

1.1 Purpose

The Design Document is a document which will help the software development by providing a detailed functional description of the system to be developed. It will include narrative and graphical documentation of the design for the project, including Use Case models, Sequence diagrams and other supporting requirement information.

1.2 Definitions, Acronyms, Abbreviations

1.3 Reference Documents

1.4 Document Structure

2 Architectural Design

2.1 Overview

The myTaxiService system will be developed on the J2EE platform, for many reasons:

- to manage the big number of possible users;
- to help developers building it as a distributed system;
- to allow the developers to concentrate on business logic;
- to satisfy the portability requirement;

Furthermore it must be said that the system will be built in a multi-tier and multi-layer architecture that will be specified below.

2.2 High level components and their interaction

From a high level point of view, the system will be designed according to the traditional multi-tier architecture used in enterprise applications. In more detail, the architecture will consist of four logical levels: the Client tier, the Web tier, the Business tier and the Data tier. The Client tier is allocated to the terminal the user logs in from, the Web tier and the Business tier stays on the Application Server, and the Data tier stays on the DBMS: this decision is made in order to guarantee properties such as scalability and fault tolerance. The Business tier are expected to interact with a mail server, using the SMTP protocol to send users messages, for example the confirmation code of the recording or the confirmation of a reservation.

Here follows a detailed description of each logical level:

- **Client tier:** accessible via the web or mobile application. This tier interacts with the Web tier through an HTTP connection, requesting HTML pages and eventually sending data provided by the user. Furtermore the browser interprets the JavaScript code, in order to provide an improvement of performances of the service (e.g. validation forms or asynchronous requests to the server).
- **Web tier:** this level receives from the Client tier HTTP requests and answers with HTML pages, generated basing on the data received, and eventually forwards requests to the Business tier. The Web tier will be implemented with JEE technology and will work with a compatible application server.
- **Business tier:** this tier can directly communicate with the database and encapsulates the business logic. The model is based on a MVC pattern, and the persistent application data are represented by Entity Beans objects. The interaction with the database and the application logic are made with EJB components. To access the database this tier uses the JPA specification, which also features ORM, abstracting the relational model implemented by the database in a model of data objects, in order to allow the interaction with the application.
- **Data tier:** it consists of a DBMS in order to guarantee data persistency. The Business tier communicates with the database using the standard technologies oj java (JDBC).

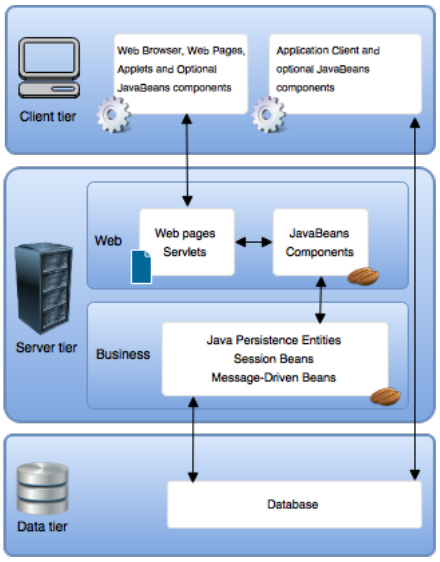

Figure 2.1: High-level architecture

2.3 Component view

In order to better understand the tasks we will face while implementing functionalities we decided to adopt a top-down approach. This means that we are supposed to decompose our system into several sub-systems. This will let us to separate, at least from a logical point of view, different set of functionalities. Later, once defined the sub-systems, we will be able to adopt a bottom-up approach in order to create more modular and reusable components. Here there are the sub-systems we decided to split our system in, with, expanded, the most relevant ones:

2.3.1 Components

Server side

- *Authentication manager*: this is an important sub-system responsible for the security of the customer registration and login system. It handles activities such as creating a new customer account, log into the system or deleting an existing account;
- *Customer profile manager*: this is the component responsible for the customer profile management;
- *Immediate call manager*: this component is the one which manages the immediate taxi calls from the customer point of view;
- *Reservation call manager*: this component is the one which manages the reservation calls from the customer point of view;
- *Taxi driver profile manager*: this is the component responsible for the taxi driver profile management;
- *Incoming call manager*: this component is the component the taxi drivers interact with in order to accept or decline the customer requests;
- *Assistance call manager*: this component is the component the taxi drivers interact with in order to accept or decline the customer requests;
- *Data component*: it is the component where the data is managed and viewed;
- *Payment manager*: it is the component that manages the payment information and calculates the fees for the rides;
- *Notification manager*: it is the component that manages the notifications to/from the users;

Client side

- *Guest component*: this is the component where the guest user is implemented, with the allowed privileges; it interfaces with the limited views which allow the registration;
- *Customer component*: it is one of the key component of the system and it's where the customer is implemented. From this component, through the proper interfaces that we will discuss below, the user gets access to the functionalities and the key features of the service;
- *Taxi driver component*: it is another key component of the system, it implements the taxi driver. It's the dual of the customer component;

2.3.2 Interfaces

In this subsection we define the interfaces the components will interact with, in order to follow the adopted top-down approach.

Server side:

- *Authentication manager*:
  1. *authenticationManagement*: this interface provides the methods that will allow the non-logged users to login to the system; it manages situation like guiding the recognized user to the relevant home page or sending email confirmation in case of registration of a new user; it will guide the recognized user to the relevant home page;
  2. *registrationManager*: this interface provides the methods which contribute to the registration of a new user and the consequent email verification request;
- *Customer profile manager*
  1. *customerProfileManagement*: this interface provides methods such as *updateProfileInfo()* and *updatePaymentInfo()*;
- *Immediate call manager*
  1. *makeImmediateCall*: this interface allows the customers to interact with the system in order to make requests for immediate taxi calls;
- *Reservation call manager*
  1. *makeReservation*: this interface allows the customers to interact with the system in order to make requests for reservations;
- *Taxi driver profile manager*
  1. *taxiDriverProfileManagement*: this interface provides methods such as *updateProfileInfo()* and *seePersonalUserRating()*;
- *Incoming call manager*

1. *incomingCallManagement*: this interface provides methods such as *acceptCall()* and *declineCall()*;

- *Taxi driver assistance call component*

  1. *assistanceCallManagement*: this interface provides methods such as *accidentAssistanceCall()*, *systemFailureAssistanceCall()* or *genericAssistanceCall()*;

- *Data manager*:

  1. *dataHandler*: this interface provides the data management methods in order to store and retrieve data to and from the server;

Client side:

- *Guest component*: the guest component communicates with the authentication and user management component through the *registrationManager* and *authenticationManagement* interfaces, explained above.

- *Customer component*: the customer component communicates with the corresponding server side components through the *customerProfileManagement*, *makeImmediateCall* and *makeReservation* interfaces, explained above.

- *Taxi driver component*: the taxi driver component communicates with the corresponding server side components through the *taxiDriverProfileManagement*, *incomingCallManagement* and *assistanceCallManagement* interfaces, explained above.

Here is the diagram showing the interaction between the two sides of the component view, including the interfaces.

Figure 2.2: Component view of the system, with interfaces.

2.4 Deployment view

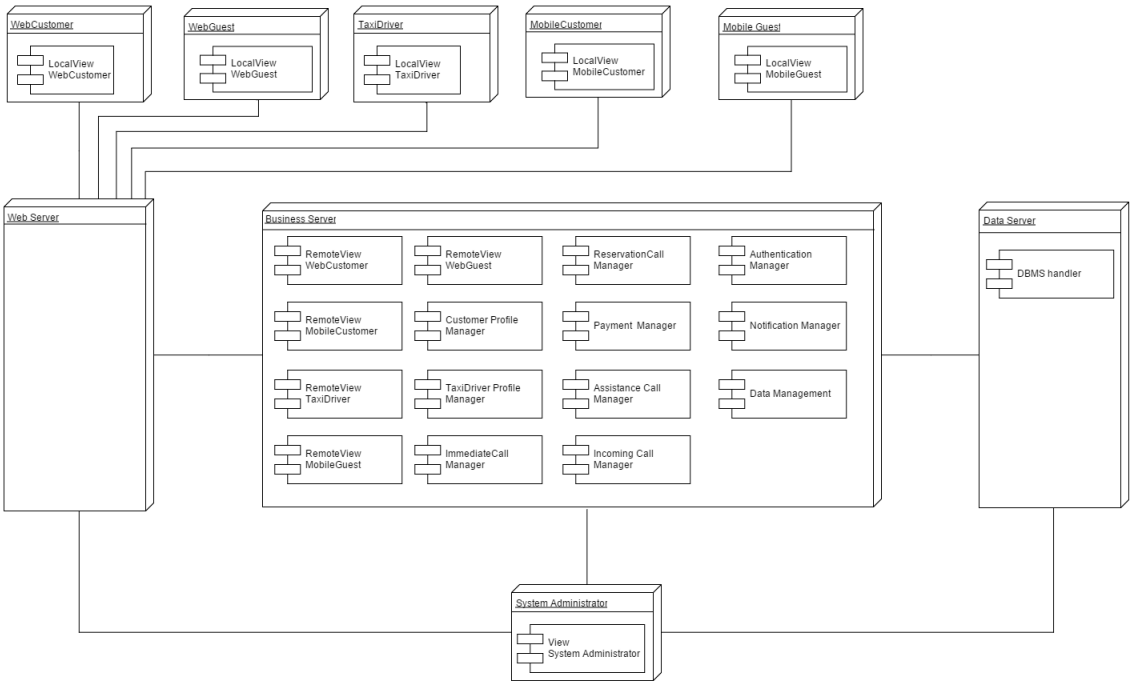

Figure 2.3: Deployment view diagram

In this section in shown how the software components are distributed with respect to the hardware resources available in the system. This diagram reports the main types of clients:

- Web customer: this is the customer that access the service using a web browser. It communicates with the Web server that receives its requests and provides HTML pages in order to start the user experience, because it doesn't have an integrated interface. Through the Web server they can interact with the Business server;
- Taxi driver: this client interacts directly with the Business server, bypassing the Web server. This decision has been made basing on the difference between a web application and a mobile application: in spite of the former, the latter has its interface included in the application;
- Mobile customer: the behaviour is the same of the taxi driver; differences between this and the previous client will be further explained in detail;

In every node representing the clients is contained a specific module, called local view, representing the accessible functionalities of the system. Every local view

corresponds to a remote view on the Business server. The Business server contains the application logic and the modules that handle the main functionalities of the system. Every module is accessible only by the allowed users, due to the different views that they have. All the data needed for the operations handled by the modules are retrieved from the database in the Data server. The various interactions with the database are monitored by the DBMS in order to guarantee the ACID properties of the database. Finally there is a system administrator, that can access to every server and most of the modules provided, basing on its particular view: we decided to deny a full access to every administrator in order to preserve the clients' privacy and deny access to sensible data.

2.5 Runtime view

In this section are explained through Sequence diagrams some of the most relevant interactions between components and actors of the system. Like in the RASD document, some assumptions are made in order to simplify the representation and clarify some aspects; by the way, some of the most important features, if considered implicit in a Sequence diagram, are explicitly represented in another one.

2.5.1 Registration Sequence diagram

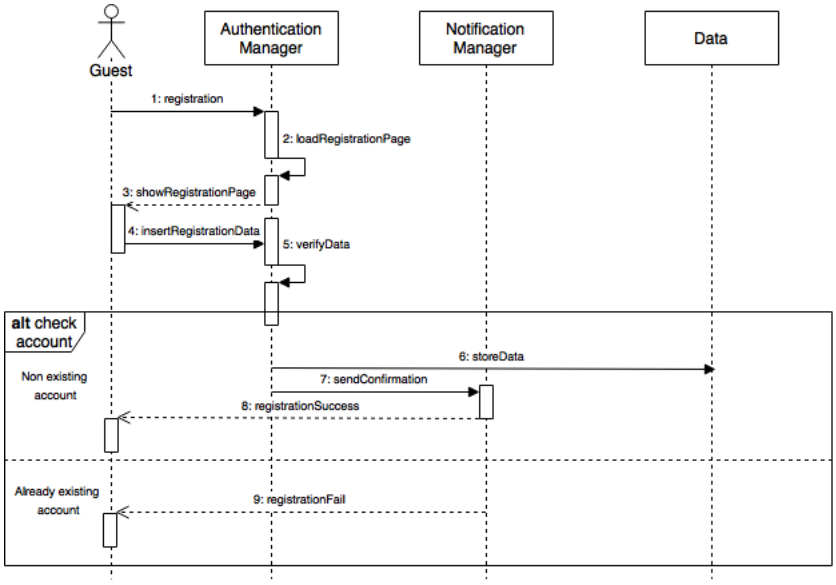

Figure 2.4: Registration Sequence diagram

2.5.2 Private immediate call Sequence diagram

From a detailed structural analysis of the system, we updated the previous sequence diagram shown in the RASD.

The changes are the following:

- added the Data, Notification Manager, Payment Manager systems;
- added the data storage operation when a ride is assigned;
- explicitly shown the interaction of different systems through the Notification Manager, when necessary;

For simplicity we assumed that, when the system is searching for an available taxi cab, there is at least one in the specific zone of the call.

Another assumption is that the Payment Manager, when verifies the validity of payment information, interacts implicitly with the Data system in order to retrieve the information to check.

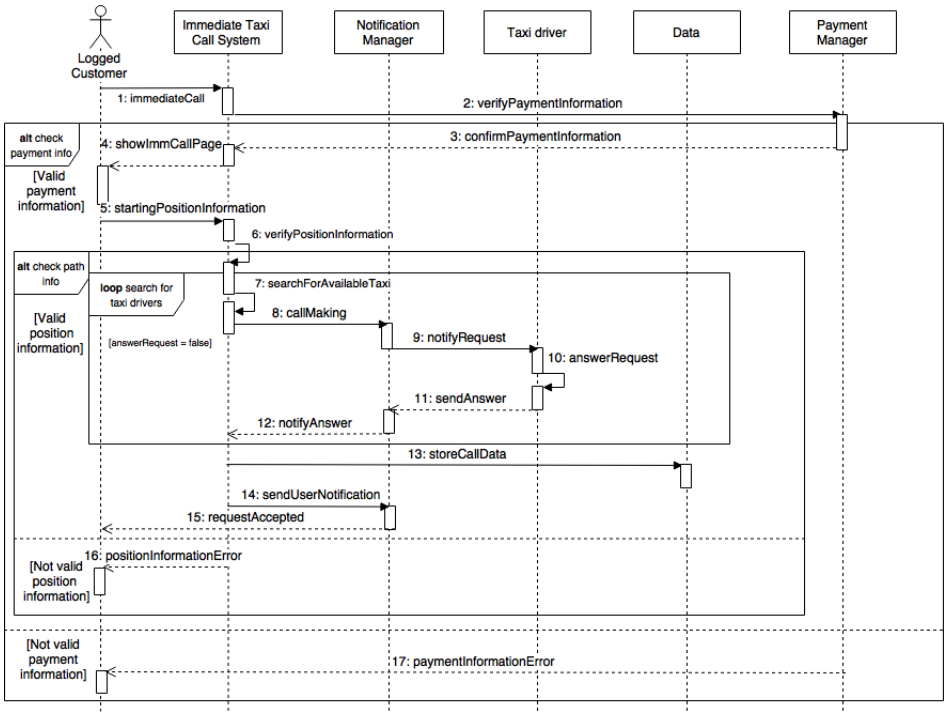

Figure 2.5: private immediate call Sequence diagram

2.5.3 Shared reservation Sequence diagram

For this sequence diagram, the changes with respect to the previous version, are basically the same of the sequence diagram for the Immediate call (2.5.2).

Furthermore, the payment information, after the check by the Payment Manager system, are assumed to be valid.

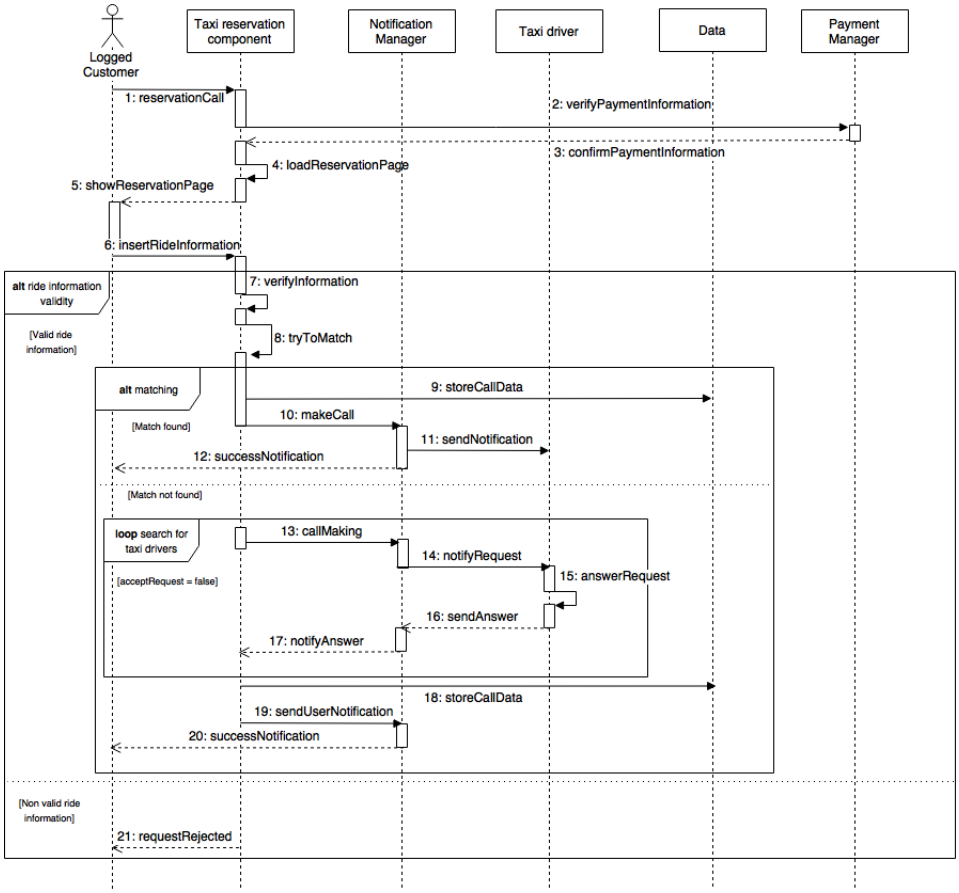

Figure 2.6: shared reservation Sequence diagram

2.6 Component interfaces

2.7 Selected architectural styles and patterns

(Please explain which styles/patterns you used, why, and how)

2.8 Other design decisions

3 Algorithm design

This section contains some algorithms that describes some of the most relevant features of the system. All of these algorithms are written in a java-like pseudocode: their aim is to give an idea of how the code should be written, independently from the chosen programming language.

Here follows five algorithms implementing the most important functionalities: note that in many cases an algorithm is used by others also described here. Some basilar methods useful for the code understanding are also defined, while others are not defined and they refer to the interaction with gps and other system components. We don't know whether they exist or not, or exists with different names, but they are surely easily implementable using the appropriate APIs (for example for the gps).

3.1 Taxi queue handler

This function handles the taxi queue for a certain zone, searching for new taxis and deleting the ones that have gone away.

```
/*
This function handles the taxi queue
for a certain zone.It uses some other
methods defined below for the basic
operations such as refreshing, adding
and removing an item from the queue.
Some methods are not defined and they
refer to the interaction with gps. We
don't know weather they exists or
not, or exists with different names,
but they are surely easily implemented
using the gps' API.
*/
List<int> queue;
void TAXI_QUEUE_HANDLER(){
queue = empty_set;
while service is active do{
List<signal> taxiSignals = searchSignals(currentZone);
for(i=0; i<taxiSignals.length; i++){
id = getTaxiDriverId(taxiSignals[i]);
if(id doesn't belongs to queue){
queue = ADD_TO_QUEUE(id);
}
}
queue = REFRESH_QUEUE();
/*
```

after inserting new elements, the system refreshes the queue in order to eliminate taxi drivers that moved in another zone.
*/
}
}
List<int> REFRESH_QUEUE(){
Position currentPosition;
for(i=0;i<queue.length;i++){
currentPosition=getTaxiDriverPosition(queue[i]);
if(currentPosition doesn't belongs to currentZone){
queue = REMOVE_FROM_QUEUE(queue[i]);
i–;
/* in this way the algorithm will recheck the same value of i at the next iteration, but with the correct next value, taking into account that the list we are reading changes at every remove.
*/
}
}
return queue;
}
List<int> ADD_TO_QUEUE(int taxiDriverId){ queue.add(taxiDriverId);
return queue;
}
List<int> REMOVE_FROM_QUEUE(int taxiDriverId){ queue.remove(taxiDriverId);
return queue;
}
List<int> GET_QUEUE(){
queue = REFRESH_QUEUE;
return queue;
}

3.2 Shared ride search

Private Ride Search: the algorithm retrieves the zone of the starting point of the ride, then according to that queue forwards the request to the first taxi driver in the line, empting the queue with a FIFO policy. If the request is accepted the taxi is removed from the queue and takes in charge the call. Otherwise he is removed and re-added to the same queue, in order to shift him in the bottom position.

void SEARCH_SHARED_RIDE(Position startingPoint, Position destinationPoint){
Ride selectedRide = RIDE_MATCHING;
if(ride == null){
create a new ride with the specified starting and destination points;

}
else{
add the customer to the selected ride;
}
}
Ride RIDE_MATCHING(Position startingPoint, Position destinationPoint){
List<Position> path <- calculate the ideal path for the ride, knowing the starting and destination points as parameters;
List<Position> currentPath = new List<Position> for(Ride r: all the active shared ride){ currentPath = r.getPath();
if(path is totally contained in currentPath && r.passengers.length()<4)
return r;
if(currentPath.contains(path.startingPoint) && path.contains(currentPath.destinationPoint) && r.passengers.length()<4)
return r;
}
return null;
}

3.3 Private ride reservation call

Private Reservation Call: the algorithm is used to accomplish a private reserved ride, so it checks that the hour is at least two hours later and then, 10 minutes before the selected time, searches for a taxi driver in the zone. If there is no available driver checks the nearby zones.

/*
The algorithm retrieves the zone of the starting point of the ride, then according to that queue forwards the reuest to the first taxi driver in the line, empting the queue
with a FIFO policy. If the request is accepted the taxi is removed from the queue and takes in charge the call. Otherwise he is removed and readded to the same queue, in order to shift him in the bottom position.
*/
Ride SEARCH_PRIVATE_RIDE(Call call){
Zone startingZone=call.getQueueStartingZone(); List<int> queue = startingZone.GET_QUEUE(); for(i=0, j=0; i<queue.length() && j<queu.length(); i++, j++){
forward request to taxi driver with id queue[i];
wait for taxi driver response;

if(taxi driver accepts the call){
ride<-create a new ride; REMOVE_FROM_QUEUE(queue[i]);
return ride;
}

else{
REMOVE_FROM_QUEUE(queue[i]); ADD_TO_QUEUE(queue[i]);
i–;
/*
In this way at the next iteration i'll chech again the same index but with a different associated object, due to the shift. j is not decremented, in order to avoid a loop if every taxi driver rejects the call and the queue restarts from the first rejecting taxi driver: in this way we forward every request only one time to the same taxi driver.
*/
}
}
return null;
}

4 User interface design

The user interface design, already shown in the RASD, are presented in this document with the addition of the Web Browser UI.

4.1 Customer UI

The following mockups represent the customer UI, regarding both the Web Browser and the Mobile Application interfaces.

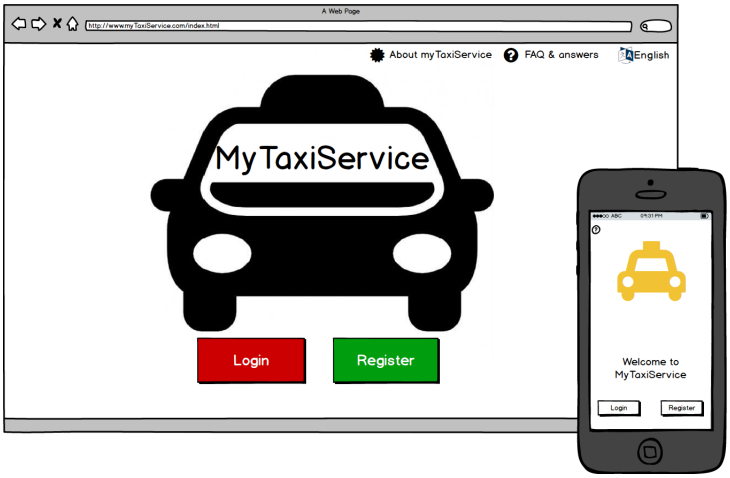

Figure 4.1: home user interface for the customer

Figure 4.2: login user interface for the customer

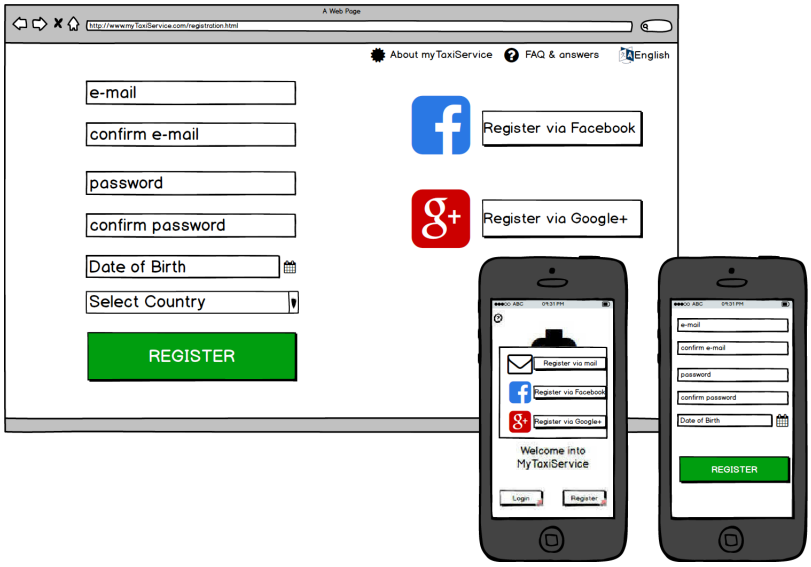

Figure 4.3: registration user interface for the customer

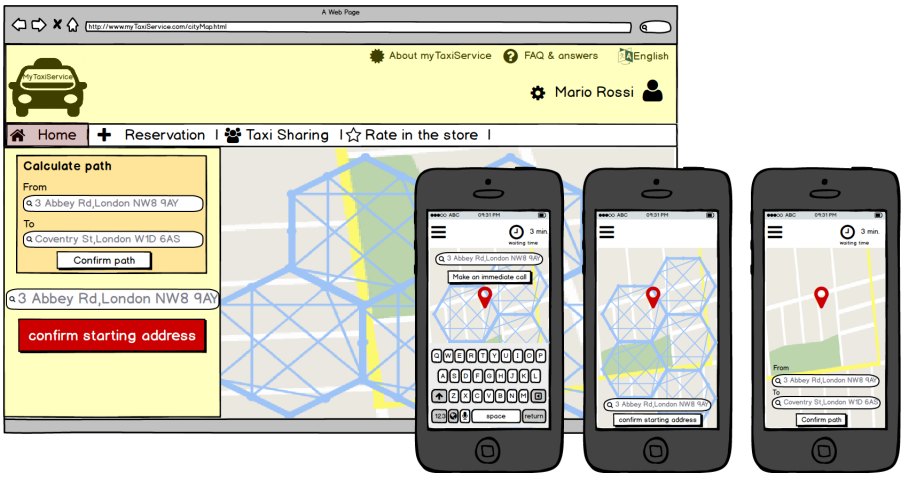

Figure 4.4: map user interface for the customer

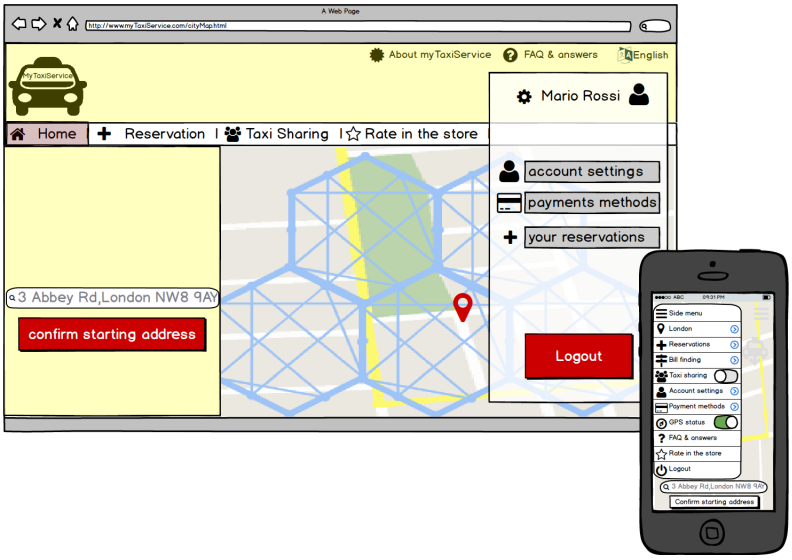

Figure 4.5: setting user interface for the customer

Figure 4.6: taxi confirmation user interface for the customer

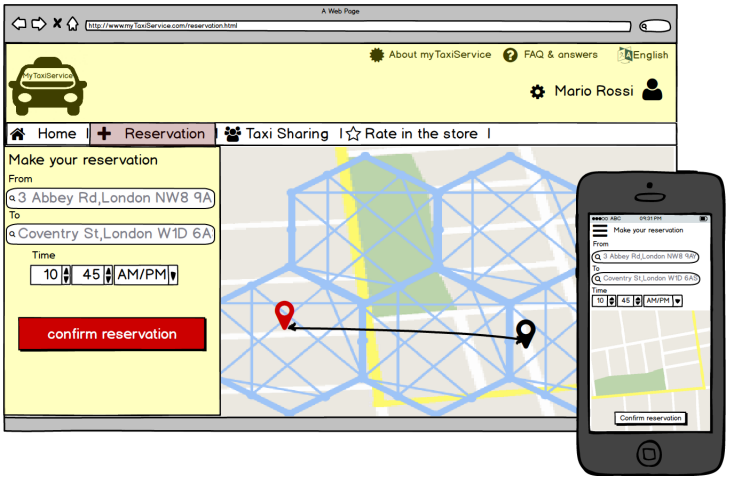

Figure 4.7: taxi reservation user interface for the customer

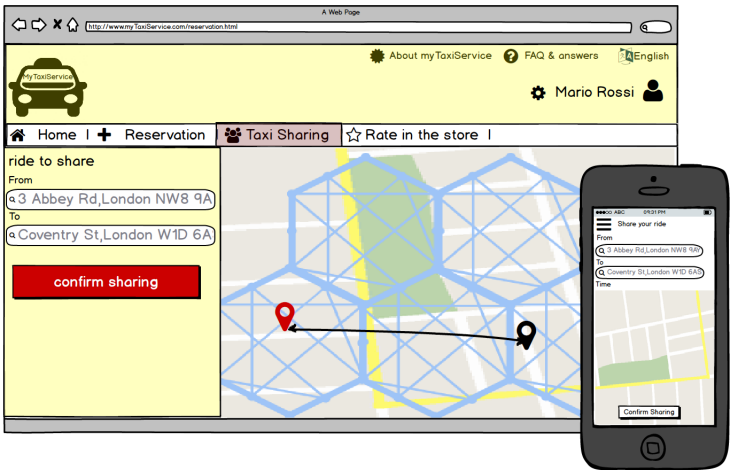

Figure 4.8: taxi sharing user interface for the customer

4.2 Taxi driver UI

The following mockups represent the taxi driver UI.

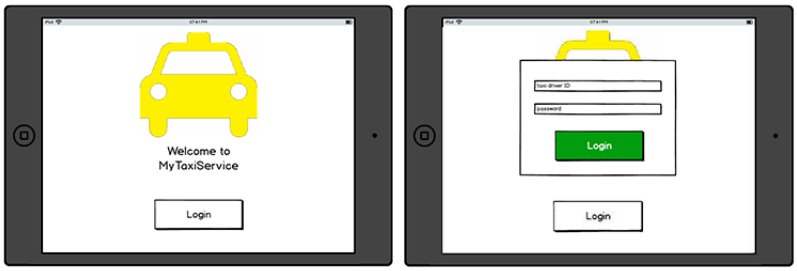

Figure 4.9: home and login user interface for the taxi driver

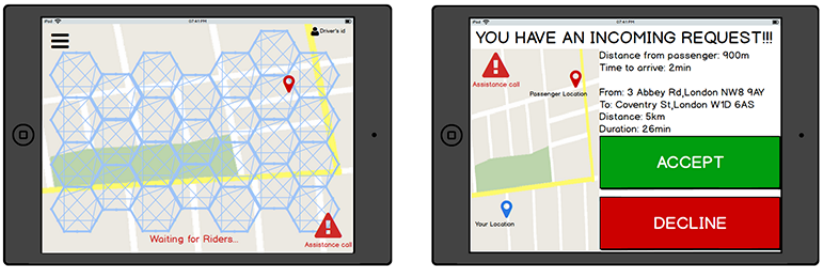

*Figure 4.*10: incoming request user interface for the taxi driver

*Figure 4.*11: riding and assistance call user interface for the taxi driver

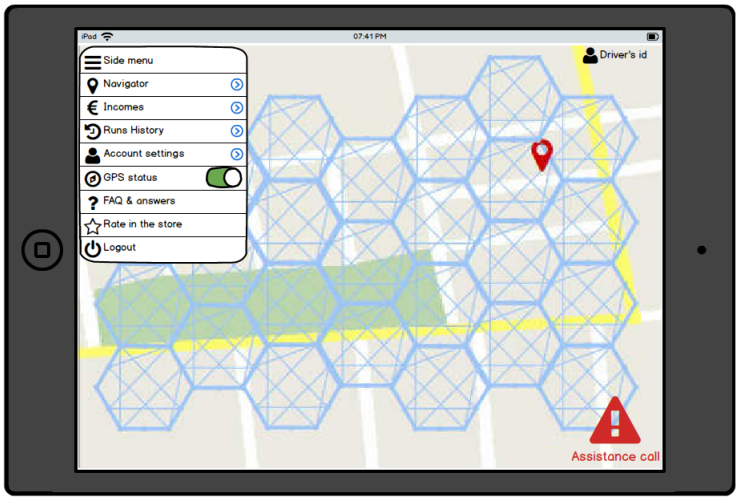

Figure 4.12: slide menu user interface for the taxi driver

5 Requirements traceability

(Explain how the requirements you have defined in the RASD map into the design elements that you have defined in this document)

6 References

7 Glossary

- **DBMS:** database management system, a software that control the creation, maintenance, and use of a database. (e.g. MySQL).
- **HTML:** HyperText Markup Language, a markup language for building web pages.
- **HTTP:** Hypertext Transfer Protocol, an application protocol used by web browsers.
- **SMTP:** Simple Message Transfer Protocol, an internet standard for e-mail transmission.
- **MVC:** Model-View-Control, a design pattern.
- **JDBC:** Java Database Connectivity, a Java API that allows the application to communicate with a database.open source database manager system, a software capable of managing data.

- **JPA:** Java Persistent API, a programming interface specification that describes the management of relational data.
- **ORM:** Object Relational Mapping, a technique of access do databases from object oriented languages.
- **EJB:** Enterprise JavaBeans, a managed server-side component architecture for modular construction of enterprise applications.
- **Entity bean:** a distributed object that have persistent state.