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

- **DBMS:** database management system, a software that control the creation, maintenance, and use of a database. (e.g. MySQL).
- **EJB:** Enterprise JavaBeans, a managed server-side component architecture for modular construction of enterprise applications.
- **Entity bean:** a distributed object that have persistent state.
- **HTML:** HyperText Markup Language, a markup language for building web pages.
- **HTTP:** Hypertext Transfer Protocol, an application protocol used by web browsers.
- **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.
- **MVC:** Model-View-Control, a design pattern.
- **ORM:** Object Relational Mapping, a technique of access do databases from object oriented languages.
- **SMTP:** Simple Message Transfer Protocol, an internet standard for e-mail transmission.

1.3 Reference Documents

- RASD document.

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).

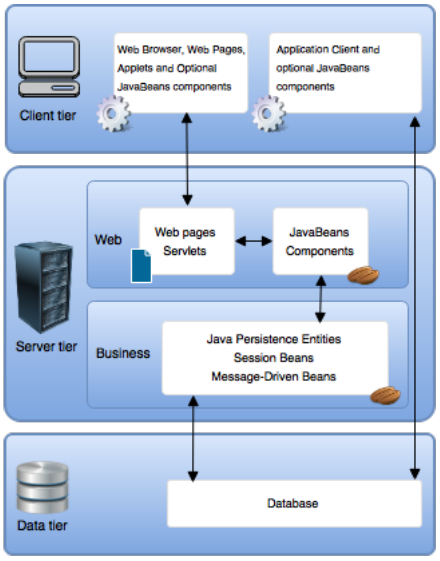


Figure 2.1: High-level architecture

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. Differently from the definition given in the RASD, the guest is a generic user, so it can either not registered or already registered but not logged;
- *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;

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

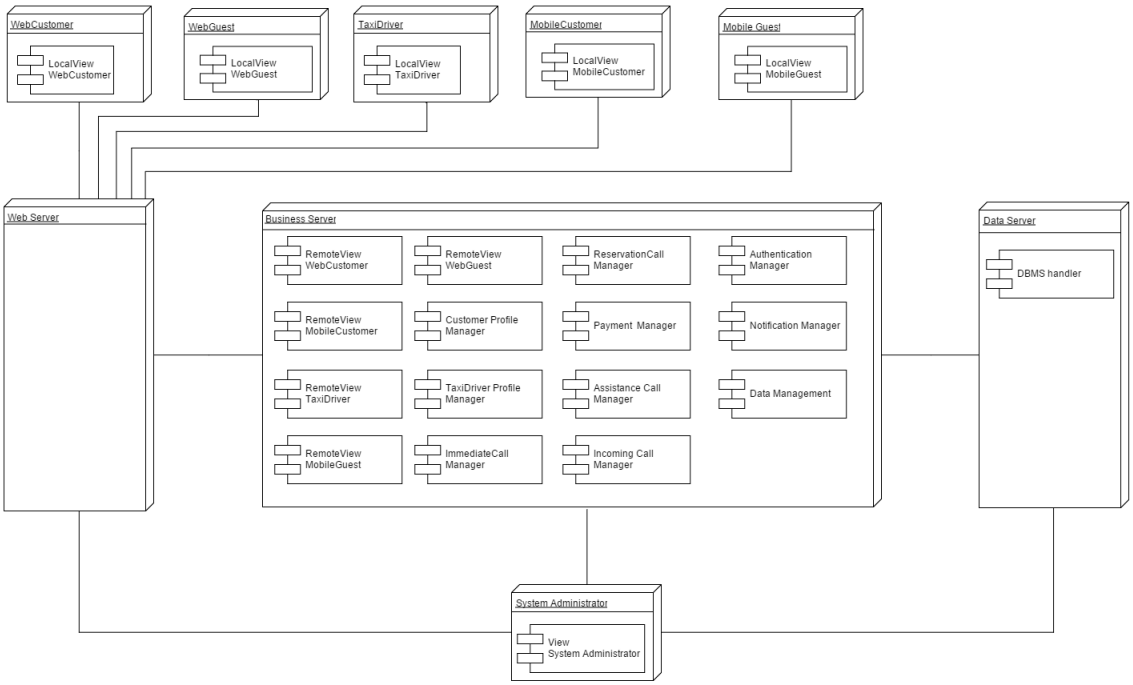


Figure 2.3: Deployment view diagram

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

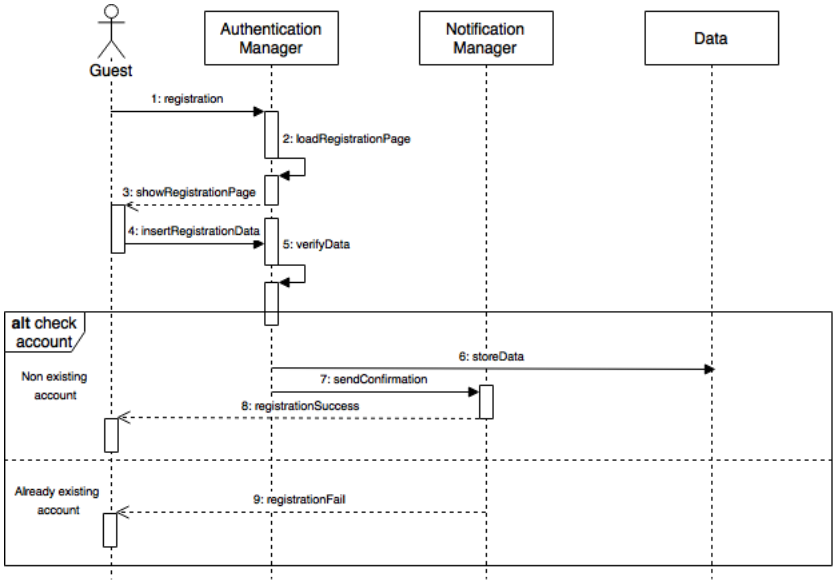


Figure 2.4: Registration Sequence diagram

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.

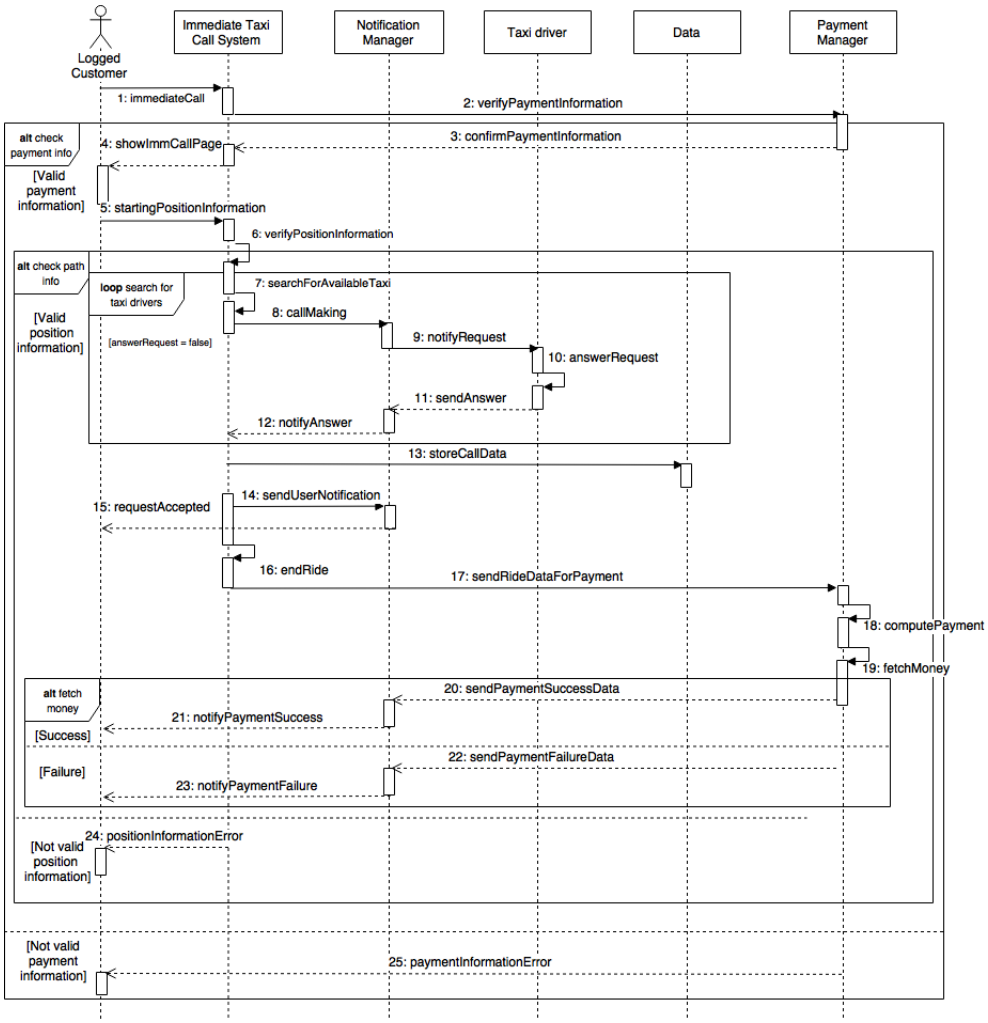


Figure 2.5: private immediate call Sequence diagram

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.

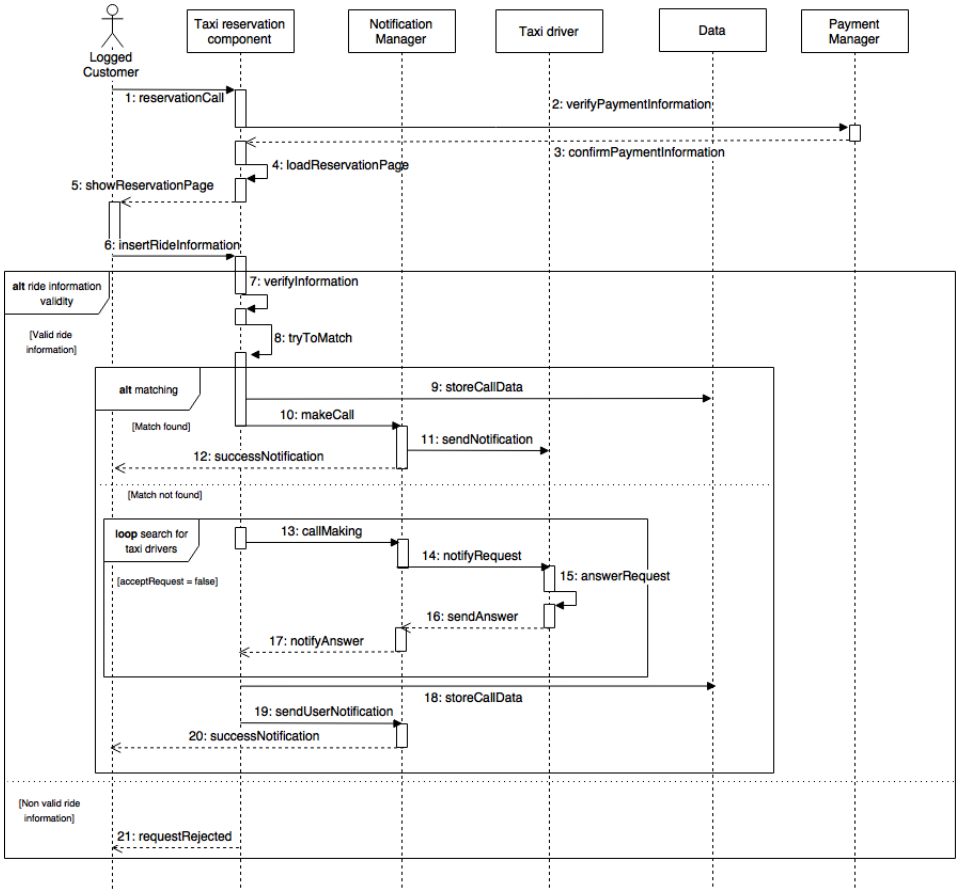


Figure 2.6: shared reservation Sequence diagram

2.6 Component interfaces

This section shows what are the interfaces that the system provides to the world, defining the functionalities they offer and the needed inputs. The high level description of the intercfaces is in section 2.3.2, so in this one are shown only the provided methods and their function.

2.6.1 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.

The most important methods are:

- void login (string id, string password);
- void logout();

2.6.2 registrationManagement

This interface provides the methods which contribute to the registration of a new user and the consequent email verification request.

The most important methods are:

- void submitData (string name, string surname, Date dateOfBirth, string email, string password, string confirmPassword);
- bool confirmIdentity (string email);

2.6.3 customerProfileManagement

This interface provides methods that allow the retrieve and modification of personal data.

The most important methods are:

- void changeEmail (string oldEmail, string newEmail, pasword);
- void changePassword (string email, string oldPassword, string newPassword);
- string retrievePassword (string email);
- void changePaymentMethods();
- List<Ride> showRideHistory ()
- void enableSharingMode ()

2.6.4 taxiDriverProfileManagement

This interface provides methods that allow the taxi driver to see or modify his personal data (with the relevant privileges).

The most important methods are:

- void changePassword (string email, string oldPassword, string newPassword);
- string retrievePassword (string email);
- List<Ride> showRideHistory ();
- int showRating ();

2.6.5 immediateCallManagement

This interface is used both for the private rides and the shared rides. The mobile and web application chose locally which methods to use, if for private or for shared, and invoke them on the server using thier interfaces.

The main method is:

- void sendImmediateRequest ();

2.6.6 reservationCallManagement

This interface is used both for the private rides and the shared rides. The mobile and web application chose locally which methods to use, if for private or for shared, and invoke them on the server using thier interfaces.

The main method is:

- void sendReservationRequest();

2.6.7 incomingCallManagement

This interface provides methods for the taxi driver that are used to accept or decline an incoming call.

The two main methods are:

- void acceptCall();
- void declineCall();

2.6.8 taxiDriverAssistanceCallManagement

This interface provides methods that allows the taxi driver to signal a problem during his service.

The most important methods are:

- void signalAccident(string comment);
- void signalSystemFailure(string comment);
- signalGenericFault(string comment);

2.6.9 dataHandler

This interface provides the data management methods in order to store and retrieve data to and from the server. The following methods contains SQL queries that allow the communication with the database.

- void storeData();
- string retrieveData();
- void deleteData();
- void modifyData();

2.6.10 notificationHandler

This interface is used by the components that have to communicate each other by notifications.

Its role is to be an mediator between them.

The main method is:

- void notify();

2.6.11 paymentManagement

This interface contains a single method that makes requests to the banking service the system is associated with.

The main method is:

- bool checkPaymentMethodValidity();

2.7 Selected architectural styles and patterns

The architecture of the system has been previously described at higher level: the logical schema of tier provided reflects the hardware architecture. Every tier is hosted on a different machine, in particular: the Client tier is on the pc or smartphone the user access the system from, the Web tier is hosted on a separate server used only for this purpose, while another server hosts the Business tier. Finally a last server contains the Data tier. This division between all levels has been made in order to allow update and maintenance on a particular tier without affecting the others, and to prevent the whole system to interrupt in case of fault of a component.

The design pattern used for this system is the Model-View-Controller (MVC). This choice has been made mainly for the flexibility that this pattern provides: the model, the view and the controller are separated into different components that can be modified with a minimum impact with respect to the others, helping to achieve a better scalability and offering the possibility of future changes to the code or the adding of new functionalities to the system.

The Model contains the representation of data and the application logic, so all the functions that manipulates data.

The View makes the model suitable for the user interaction, providing a GUI: every user will access the system and all its functionalities through this component. Furthermore provides the possibility to have different representations (or views) for the same data, depending on the user's preference or on the user's permissions.

The Controller is responsible for the communication between the View and the Model: in particular responds to user's actions and invokes changes on the model.

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();
/*
the refreshing of the queue happen whenever a
taxi driver  changes his status, for whatever reason,
such as when the unavailability status is set,
or a request get declined.
*/

}
}

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 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.

```
/*
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<queue.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]);
                KEEP_TRACK_OF_THE_REFUSING_DRIVER();
                /*
                This operation in order to not forward
                requests to the same taxi driver twice.
                */
        }
}
return null;
}
```

3.3 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.

```
Ride PRIVATE_RESERVATION(ReservationCall call){
Time reservationTime = call.getRideStartingTime();

if(currentTime()-reservationTime < 2hours){
        return null;
        /*
        a reservation must be at least for 2 hours after the call,
        otherwise will be rejected
        */
}
```

```
else{
        wait until 10 minutes before reservationTime;
        Ride ride = SEARCH_PRIVATE_RIDE(call);
                while(ride == null && (reservationTime-currentTime()) < 5 minute
                        ride = search for a taxi driver in the adjacent zones;
                }
                notify the customer for the current unavailability of taxi drive
                retry in 5 minutes;
}
return ride;
}
```

3.4 Shared ride search

The algorithm is used to search for a shared ride, paying attention to the matching of the paths. If no shared ride can be matched with the needed path the system creates a new shared ride, and the customer will be the first passenger.

```
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;
}
```

3.5 Fee amount for shared rides

This algorithm is used to calculate the cost per passenger for every customer ending a shared ride, dividing the fee of the shared parts of the track with the other customers that are sharing the same track. At the end of the shared ride, the sum of all the costs of every customer that joined must be equal to the cost that results for the same total ride made by a single customer.

```
public class Ride{
        List<Slot> ride_slots;
        //other code
/*
This method calculates the fee related to his ride.
In order to do this, a specific listener activates
this method when it        observes that a customer
ends his shared ride.
*/
        public float feeAmountCalc(Customer passenger){
                float passenger_fee = 0;
                foreach(Slot s: ride_slots){
                if(passenger isIn s.passengerList){

passenger_fee = passenger_fee +
 (s.final_time - s.initial_time)*(fee_per_minute)/(s.passengerList.length);
                }
        }

        return passenger_fee;
        }
}

/*
This class represents a time slot; each ride is
divided   in many slots.   Every time that the
number of passengers of the ride   changes (a
passenger gets in or gets out of the cab),
a new slot is created. Of course the initial
time of the   first slot corresponds to the
starting time of the ride,    and the final
time of the slot n corresponds to the
initial time of the slot n+1. The current time that
fills the relative variables
in each time slot is taken from the system time.
*/
public class Slot{
        private List<Customer> passengerList;
        private time initial_time, final_time;
```

```
        //other code
}
```

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.

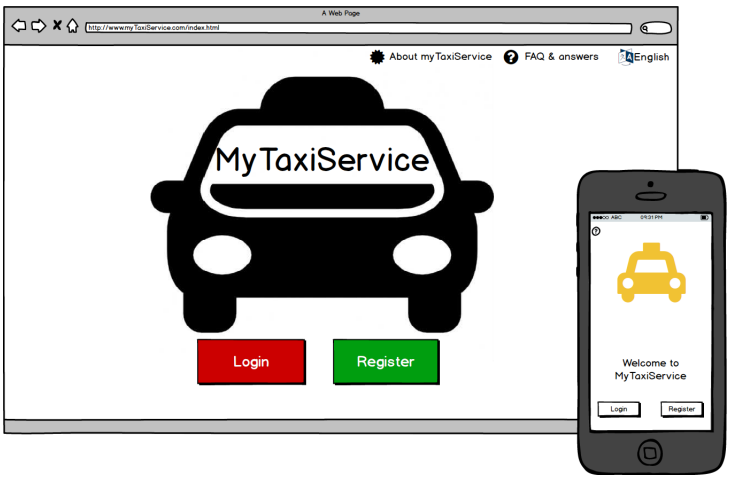


Figure 4.1: home user interface for the customer

Figure 4.2: login user interface for the customer

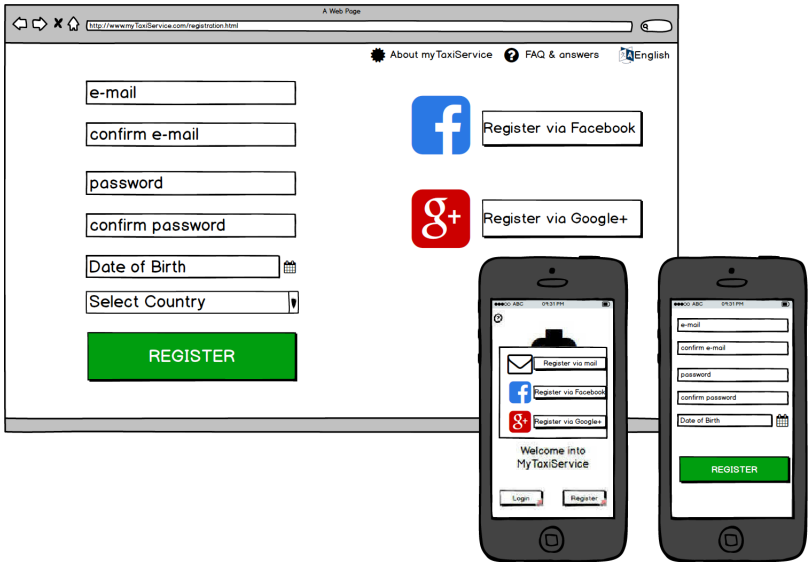


Figure 4.3: registration user interface for the customer

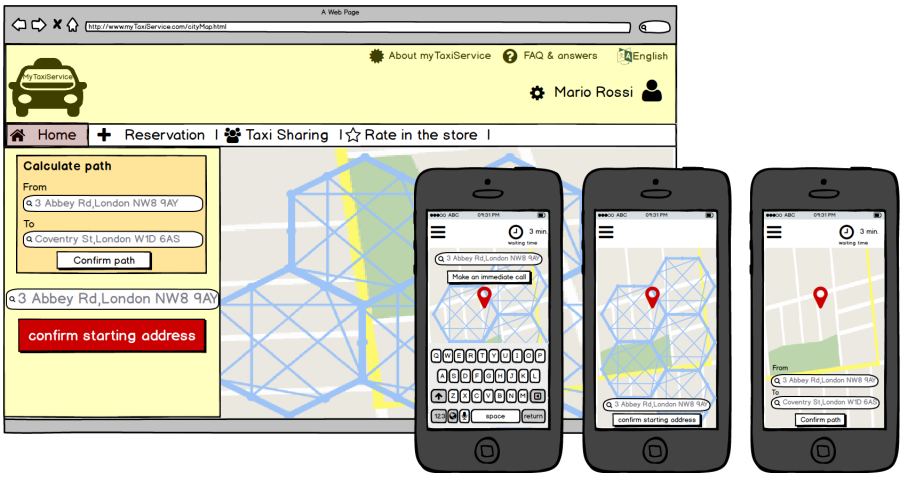


Figure 4.4: map user interface for the customer

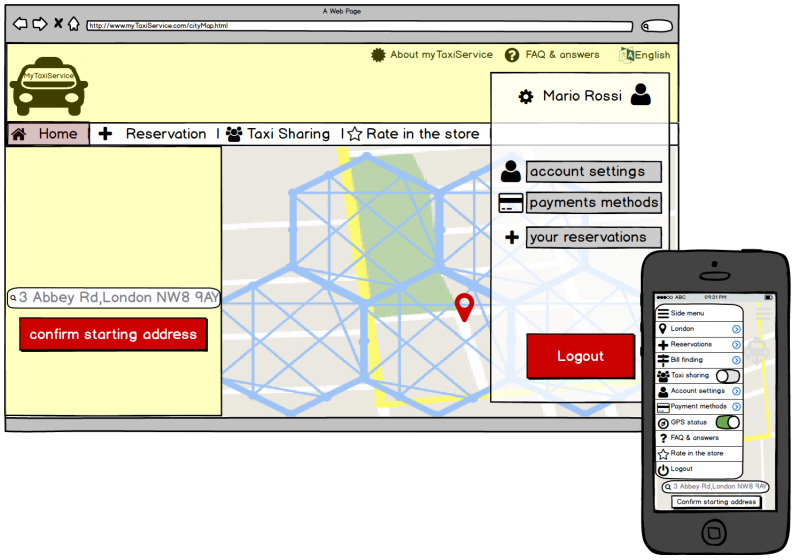


Figure 4.5: setting user interface for the customer

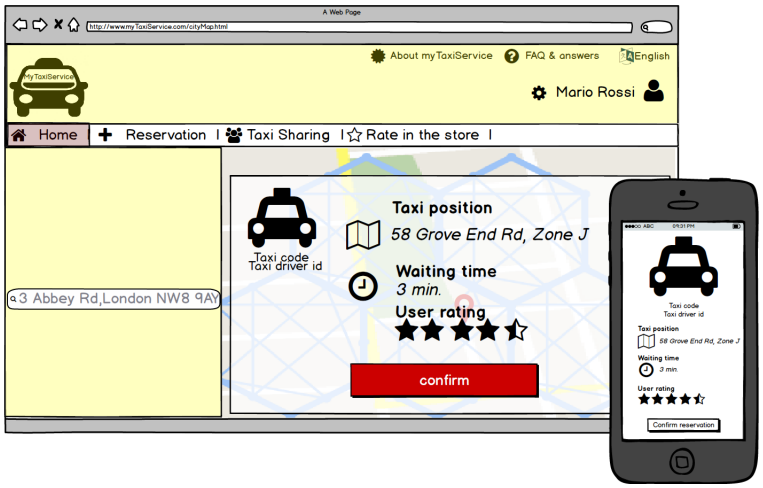


Figure 4.6: taxi confirmation user interface for the customer

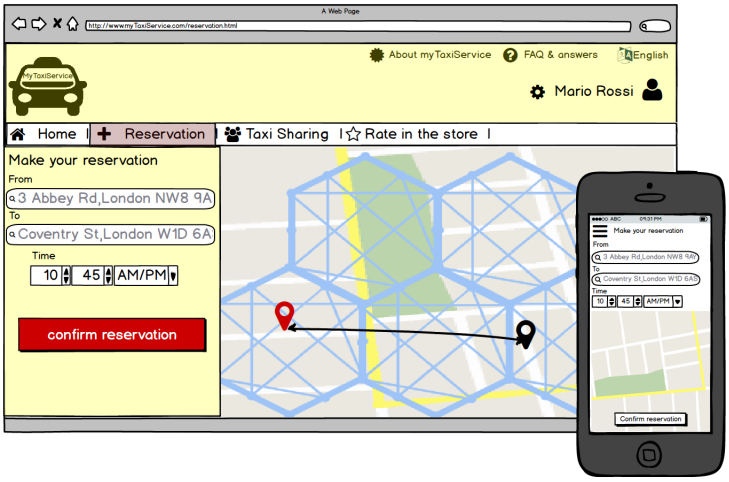


Figure 4.7: taxi reservation user interface for the customer

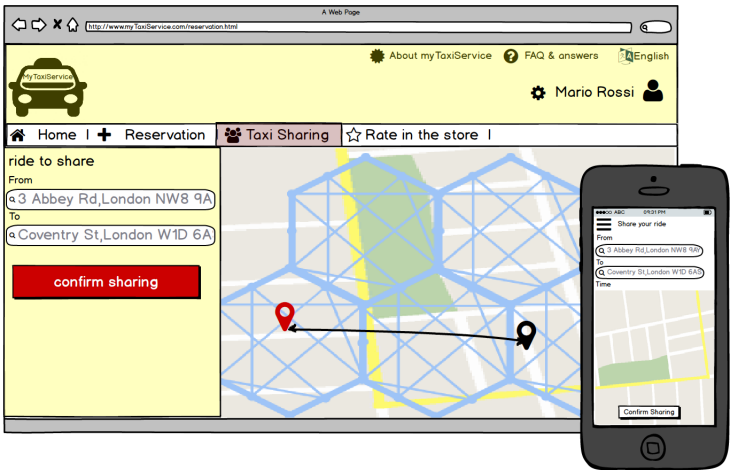


Figure 4.8: taxi sharing user interface for the customer

4.2 Taxi driver UI

The following mockups represent the taxi driver UI.

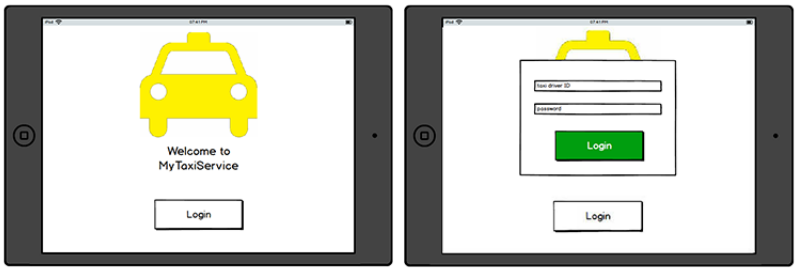


Figure 4.9: home and login user interface for the taxi driver

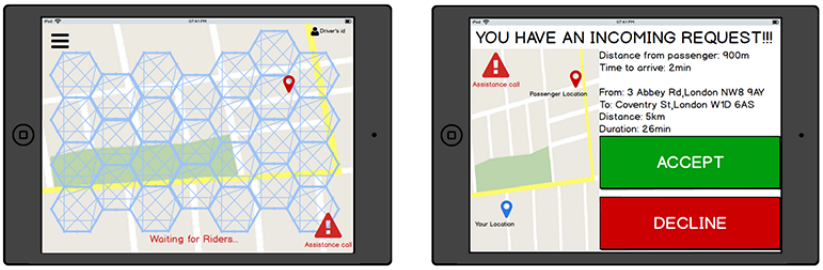


*Figure 4.*10: incoming request user interface for the taxi driver

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

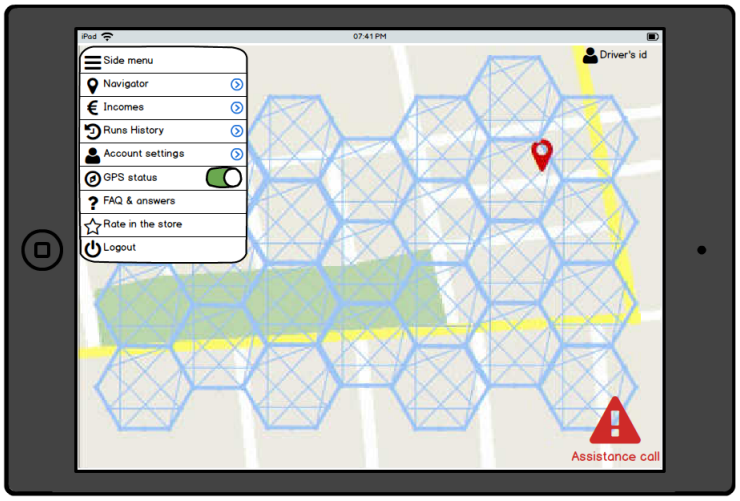


Figure 4.12: slide menu user interface for the taxi driver

5 Requirements traceability

In the RASD document have been defined all the requirements (both functional and non-functional) of the system: in this section is explained how they maps into the design elements defined in this document.

Here follows the list of requirements, together with how they maps and which components interacts in order to allow their satisfiability:

- The system has to guarantee the registration to all the new major users who want to create a new account: there are no restrictions that could not allow the registration process, such as surveys or something similar;
- In order to complete the registration process, the system sends a confirmation link to the new user via email: in the section 2.5.1, figure 2.4, the sequence diagram shows that after the registration process, if successful, the system sends a confirmation email to the customer (operation 7). Used components: registrationManager, notificationHandlerPool;
- The system has to allow the login to an already registered customer when he types the correct email and password in the login form: in the RASD document, section 5.3, figure 5.6, the sequence diagram shows that the login is successful if and only if there is an existing account associated with the username and password provided (operation 6). Used components: authenticationManager;
- The system has to allow the login to a taxi driver when he types the correct identification code and password in the login form: see the requirement below, the process is the same;

- When a customer provides his payment information, the system verifies the validity of the submitted data and in case of success unlocks all the allowed functionalities: in every process that involves a payment, the system checks the validity of the payment method as a preliminary step, before doing any other operation (see section 2.5.2, figure 2.5, operation 2, and section 2.5.3, figure 2.6, operation 2). Used components: paymentInterfacePool;

- The system notifies the confirm of the call with an email and has to assign the first taxi in the queue of the zone where the call comes from, if it's not empty: when a user makes a call providing the correct information needed, and after the acceptance by a taxi driver, the notification manager notifies the user with a mail containing the confirmation of the call (see section 2.5.2, figure 2.5, operations 14-15, and section 2.5.3, figure 2.6, operations 19-20). Used components: notificationHandlerPool, manageCall;

- When a taxi is assigned to a ride but rejects it, the system moves the selected taxi to the last position of the starting zone's queue: in section 3.2 the algorithm removes the taxi from its queue when he rejects a call and then add him to the same queue, placing him in the last position;

- If the queue of the starting zone is empty, the system has to notify it to the calling customer and aborts the current operation: in sections 3.2, 3.3 and 3.4, the algorithm returns a null ride in case of failure of the search. Used components: notificationHandlerPool;

- In a shared ride, the system asks the user the destination too: in the RASD document, section 5.2, figure 5.4, the class diagram shows that the class call contains the destination point of the ride. Furthermore, in this document, section 3.4, the algorithm used for the search of a shared ride requires the destination point as an input for the function;

- The system matches the route with the other shared calls (reservation or immediate calls), compatible both for the time and the path: in section 3.4 the algorithm defines the search for a shared ride, and to achieve this compares the tracks in order to find a compatible ride with the time and the path;

- If there isn't any compatible shared ride to match with, the system normally assigns the first available taxi: in section 3.4 the algorithm defines the search for a shared ride, and if no shared ride is compatible, creates a new ride with the provided parameters;

- If there isn't any available taxi in the zone where the request comes from, the system has to notify it to the calling customer and aborts the current operation: in every sequence diagram, section 2.5, the system notifies the user of the unavailability of taxi drivers;

- In a reservation call, the system notifies the confirm of the reservation with an email and allocates a taxi to the request 10 minutes before the meeting time with the user, notifying the first available taxi of the queue of the starting address' zone: in section 3.3, the described algorithm waits from the time the call has been made until 10 minutes before the desired reservation time, and then uses the search algorithm for a standard ride in order to find a taxi driver for the user. Used components: makeReservation, notificationHandlerPool;

- If the queue of the zone corresponding to the starting position is empty, the system has to forward the request to the first available taxi of the closest zone: in section 3.3, the algorithm, if it's not able to find a taxi in the starting zone of the ride, searches for an available taxi driver in the nearby zones.

6 Persistent data management

The structure of the database containing persistent data has been designed basing on a relational database, implemented on the MySQL server used for the Data tier which interfaces with the Business tier.

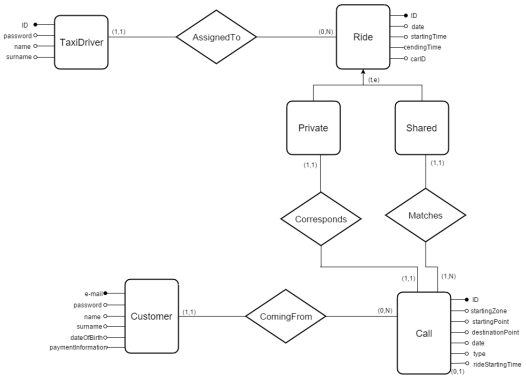
Entities and relations that we have identified are:

- Taxi Driver: this entity represents a taxi driver hired by the company. Each one is identified by a unique ID; the other attributes are the profile information (name, surname) and those needed for authentication (the value of a hash function of the password). The association "AssignedTo" means that every taxi driver can be associated to zero or more rides.

- Ride: this entity represents a ride that has been correctly performed (correctly means that it's already ended without casualties or any kind of complication). Each one is identified by a unique ID; the other attributes are the temporal coordinates (when and at what time the ride has been performed, so date, startingTime and endingTime) and the car used for the ride (carID). This entity generalises Private and Shared, which are the two type of ride that can be performed: the ride type is exclusive, so a ride must be or shared or private, but not both. Every private ride is associated with one and only one call through the relation "Corresponds", while a shared ride can be related to more than one call to the relation "Matches". A generic ride (private or shared) corresponds to one and only one taxi driver, through the relation "AssignedTo".

- Customer: this entity represents a registered user of the system, identified by a unique couple of e-mail and password. The other attributes are the personal information of the user (name, surname, dateOfBirth) and the payment information(pymentInformation). The relation "comingFrom" states that a customer can make an arbitrary number of calls, zero included.

- Call: this entity represents a user's call for a taxi. Each call is represented by a unique ID; the other attributes are the temporal coordinates of the call (date), the special coordinates (startingZone, startingPoint and destinationPoint). There is also an optional attribute, rideStartingTime, that is used for reservation calls, representing the time at which the user wants to have a taxi (different from the time in which the call has been made). The "ComingFrom", "Corresponds" and "Matches" relation means that every call is associated, respectively, with one and only one customer, private ride and shared ride.



RIDE (Id, date startingTime, endingTime, carId, type, taxiDriver)

TAXIDRIVER (Id, password, name, surname)

CUSTOMER (e-mail, password, name, surname, dateOfBirth, paymentInformation)

CALL (id, startingZone, startingPoint, destinationPoint, date, rideStartingTime*, type, customer)

PRIVATERIDE (call, ride)

SHAREDRIDE (call, ride)

6.1 Conceptual design

The ER diagram below is the result of the analysis of the Class Diagram presented in the RASD document, combined with the data of interest that we want to store persistently. The result of this analysis is the following schema representing the entities and relations of interest.

7 Changes in the RASD

During the writing of the Design Document, we noticed some aspects to change in the RASD.

- Class diagram updated: some attributes added to the "Ride" and from the "Taxi driver" classes, and an attribute removed from "Shared ride" class;

- Sequence diagram updated: see section 2.5 of this document for details;
- Assumptions: see the RASD_updated file in the 2.2 section;
- Functional requirements: see the RASD_updated file in the 3.1 section;

8 Used tools

- LYX to generate the pdf document;
- draw.io to generate the Sequence and the Use Case diagrams;
- Balsamiq Mockups 3 to generate the mockups of the mobile app and the web page.