Politecnico di Milano

Department of Electronics, Information and Bioengineering

Master Degree course in Computer Science Engineering

Design Document (DD)

myTaxiService

Instructor: Prof. Elisabetta Di Nitto

Authors:	*Code:*
Luca Luciano Costanzo	789038
Simone Disabato	852863

December 4, 2015

Contents

List of Figures

Chapter 1

Introduction

This chapter provides a short description about the purposes and the scope of this document. After that, a glossary is given to help the readers to understand the meaning of each word or acronym used in this document. At last, the main structure of the document is shown.

1.1 Purpose

The design document of myTaxiService aims to describe all the aspects concerning the architecture of the system. The introduced tiers or levels into that architecture are described and studied more in details, explaining the reasons for the single choices and the interactions between them.

After that, the key algorithms of this system are shown in pseudo-code to suggest and describe the real implementation of the code. Finally, the last purpose is to give the readers the idea of final applications (both MA and WS) using mockups.

1.2 Scope

Users, once registered, are able to ask for an immediate ride or to book one of them.

The system provides the user with a complete map of the city and its suburbs within the taxi service is available. The current position of the user is obtained by localization services of the user's smartphone if it's possible, otherwise the user notifies his position directly on the map with a marker or by a searching box. The destination is also chosen either graphically or by a research. The user can view the suggested path and then he must confirm the request.

When a user asks for a ride, the system checks the availability of a taxi driver near the current position, by splitting the city in several areas and using a FIFO (First In First Out) policy to manage the assignment of the ride's driver. The selected driver can accept or decline the ride. In the former case the system informs the user about waiting time, estimated travelling time, prices and cab car-code.

The system gives also the possibility to book a ride with at least two hours in advance. As the user does when he asks for a ride, he selects the desired starting venue and the destination. Afterwards, the system gives a calendar where the customer can choose the date (at most 30 days in advance) and the starting hour. Ten minutes before the meeting time the system starts all the operations described before in order to assign a taxi-driver.

A reservation from the app or the website can be undone until the system confirmation of the availability of a taxi, while a booking can be cancelled at most fifteen minutes before the meeting hour.

After those deadlines the ride is considered bought by the customers and an eventual absence on the established venue forbids other possibilities to book or to take a ride.

1.3 Definitions, Acronyms, Abbreviations

MA
: acronym for the mobile application.

Mockup
: a simple graphical representation.

WS
: acronym for the website of the system.

1.4 Document Structure

The chapter 2, called Architectural Design, describes all architectural choices. First, the high hierarchy of that architecture is shown and the interactions between its components are explain. Then, for each defined level or tier a standalone paragraph is dedicate to present all its characteristics.

The chapter 3, called Algorithm Design, points out the key algorithms of this system. In particular, these algorithms are the ones which manage the cabs' queues and the city areas and the ones which manage the special case that happen when no taxi are available in the desired area.

The chapter 4, called User Interface Design, describes all the graphical interfaces by using mockups.

Finally, the chapter 5 is dedicated to point out the links between requirements presented in the RASD document and the decisions taken and shown in this document.

Chapter 2

Architectural Design

In this chapter the complete architecture of myTaxiService is shown with various levels of description. In the section 2.2 there is a global view and the interactions between all the components are described.
The data tier is illustrated in section 2.3 with all related policies and entities. Then the other tiers are characterized using different diagrams.
In the section 2.4 the deployment of each components is illustrated (for instance the data component is sit in a different place with respect to the other component? It is replaced twice or more? And similar question will have an answer).
In section 2.5 the view level is defined. The interactions between all kinds of user and the system are described using UX diagrams and sequence diagrams that display the order in which each screen is visualized. Besides, the mockups of these screens are shown in chapter 4.
A standalone paragraph, the section 2.6, is dedicated to list all interfaces, both internal (between two components) and external.
Finally, in the section 2.7 the design patterns used to develop myTaxiService are described first in general case. After that, all the changes needed to adapt this design patterns to our system are characterized.

2.1 Distinctions between various kind of Users and Clients

The "visible architecture" of myTaxiService is very varied. The term visible is referred to various user interfaces, so what the users can see when they are using the system.
On the other hand, we have said that myTaxiService is varied because it has two principal version (MA and WS) and for both of them there are a few levels of specialization, according to the kind of the user. All of them are explained in this paragraph.

The WS version is shown into a browser, so there is no client application that can be used. Hence, all the pages are loaded into the server and then they are sent to the client browser.
Instead, the MA is a client application and it has different ways to communicate with the server. All the aspects concerning these differences are explained better during the descriptions of the architecture's components that handle the clients.

The cab company is the special user who administrates the service. Obviously, it has a command center at its headquarters where it can control both the system and the service situation. Hence its special functions are not implemented neither in the MA nor in the WS, but they can directly access the server using private keys and reserved terminals.
A customers can use both the MA and the WS to enjoy his functionalities. No particular cases or restriction are reserved to them.
When a driver logs into the service, we suppose that it is working, so his special functionalities are developed and implemented only for the MA. In fact no driver carries a computer with an internet connection on his taxi and uses it. On the other hand, since the driver is also an user, it can use the WS, but here he has only the user functionalities due to the reason shown above.

2.2 High level components and their interaction

The main structure of myTaxiService can be described as a Service-Oriented architecture (see section 2.7 to have a detailed description of this design pattern). In addition to this idea the Model-View-Controller paradigm is applied in order to develop a good-programmed, reusable and easy-maintainable system.

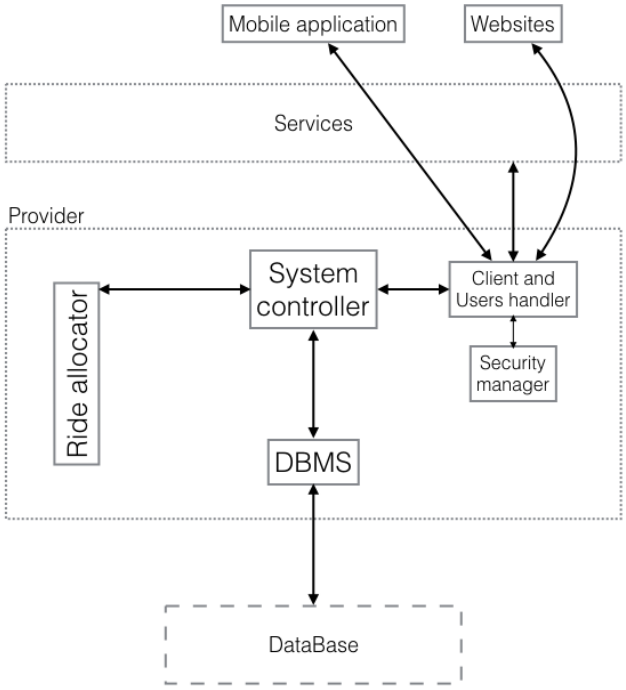

Figure 2.1: High level architecture

In the figure 2.1 is shown the architecture. At the bottom of the figure there is the Data tier which contains only the database (a component that stores all data about the customers, the rides and the system).

The central part of the picture contains the provider, the big component that gives the service. At this level of description we can look in the provider. This component is split into five parts. The DataBase Management System (DBMS) manages all communications with the Data tier. The Ride Allocator has only

one role, to assign and to handle the ride, both future and zerotime. The system controller is the "heart"of the provider: it involves the ride allocator when is needed, it gives all the services of the system, it provides the DBMS with the data to be stored (or asks him some data to be found) and communicates with the Client and Users Handler. The Client and User Handler, as its name said, handles the communications with the clients and administrates the Users, so which services can be shown and selected by them.

The third tier is a presentation level. In the Service-Oriented pattern this level represents the available services. Here, the clients can see the services and select the desired one.
Finally, at the top of the image, there is a representation of the two kind of clients, the WS and the MA.

All the tiers are also distinguished into the three parts of MVC-paradigm. The Data tier is the model[1]. The provider is also the controller: even if the controller can be seen as the component that manages the interaction between the view and the model, in this application we consider other "handler"as part of controller because they encapsulate rules and action codes. The view is the union between the presentation tier (here the services are only shown to client, the "decisions"are taken by the provider) and the clients.

Up to now, we have introduced all the main components or tier of our system, without focusing on the communications between them, so we'll dedicate a little part of this paragraph on this argument.
All the communications between the model and the provider are managed by the DBMS that has store policies and finds the required data. The system controller is the central node in communication because it is the only way to all the other components of the provider. The Client and User Handler is the link between the provider and the clients. It checks the type of the current user and, as

[1] see the Alloy section or the UML class diagram into the RASD to see it. However, in section 2.3.1 it is studied again.

consequence, the available services that he can uses. When a user asks a service, he binds the request asking for the related function to the System Controller. Finally, it checks the type of client, because they have two ways of communication. When a user is connected with the WS, the Client and User Handler load the pages into itself and then, using the HTTPS protocol sent them to the client. Besides, the communications with the MA is implemented using sockets. The implementation of these ways of communication with the clients is realized by a common interface (see the section 2.6 for further information).

2.3 Component view

2.3.1 Data tier

This paragraph shows the Logical schema of the database in figure 2.2 for the application.
The tuples names are written in italic style, the underlined attributes are the primary keys, while the bold attributes are the reference keys.

- *rides* (rideID, **passenger**, **driver**, **departure**, **destination**, departureTime, arrivalTime, creationDate, isFuture) :
  it contains both the zerotime rides and future rides, the field isFuture has value 1 when it's a future ride, 0 when zerotime ride.
- *users* (userID, email, password, name, surname, taxcode, birthday, cityOfResidence, isDriver, registrationDate, activationCode, activated) :
  it contains the general and fundamental information of all the users. The field isDriver has value 1 when the user is a driver, 0 otherwise. The field activated has value 1 if the user has completed the registration process by confirming on the link with his activation code sent to his e-mail address, 0 otherwise.
- *drivers* (driverID, **userProfile**, cabCarCode, **cabCompany**) :
  it contains all the information of the drivers. The field userProfile is the reference key to the record in the users table which contains all the other general information of the driver.

- *cabCompanies* (cabCompanyID, name) :
  it contains all the cab companies that use myTaxyService.
- *positions* (positionID, gpsLatitude, gpsLongitude, **address**, civicNumber) :
  it contains the stored positions for the rides (departure and destination) with the reference to the corresponding address in the table addresses and optionally the gps coordinates and the civic number.
- *addresses* (addressID, city, street, area) :
  it contains all the existing streets and cities covered by myTaxyService, with the reference to the area which they belong to.
- *areas* (areaID, name) :
  it contains all the areas covered by myTaxyService.
- *driversWaiting* (**area, driver**, driverAddedTime) :
  it contains all the available drivers waiting for a ride request. The couple of keys "area,driver" is unique because a driver cannot be available in more than one area at time. The field driverAddedTime contains the date and time when the driver is added to the list of available drivers for a certain area, it is useful to sort the list with the desired order (a FIFO queue in our application).
- *workShifts* (workShiftID, **driver**, weekDayNumber, startingTime, endingTime) :
  it contains all the work shifts of a driver. weekDayNumber is the number of the day of the week (between 1 and 7, 1 is Monday, 7 Sunday). startingTime and endingTime represent a continuous time range of a work shift of a driver in a certain day of the week.
- *alerts* (alertID, **user**, receptionDate, message) :
  it contains all the alerts that a user has received. "user" field is the reference to the receiver of the alert.

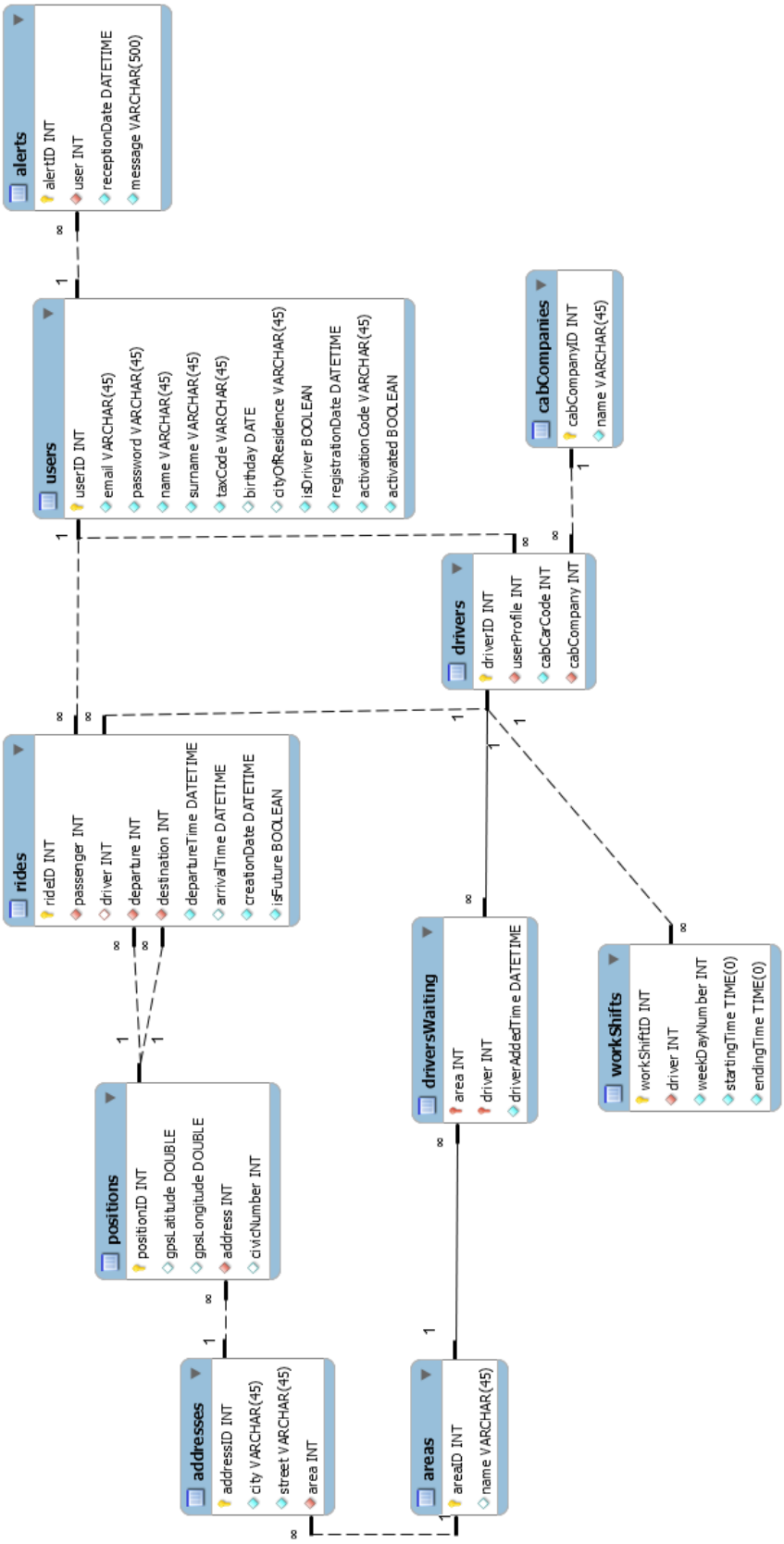

Figure 2.2: Logical schema of the database in the Data tier

2.3.2 Provider tier

The provider is the "hearth"of myTaxiService and it is split into several components, each one dedicated to a specific function. In section 2.2 a short description of the provider's components has been written. In this paragraph we'll go in deep for each one.
The **DBMS**, as its name says, manages the database and gives the system an interface to search, update or delete data. This interface can be used only by the System Controller by the use of queries already written into the application code. When the DBMS receives a query, first checks it to prevent some malevolent data (for instance SQL injection on same parameter asked to the client), then performs the required operations and gives methods to access the result.

The **Security Manager** is a small component that implements all the security procedures of the provider. It contains:

- The definition of the data encryption methods;
- The firewall methods between the Provider and the Client and Users Handler;
- The HTTPS protocol implementation;
- The algorithms to check the authenticity of the administrators.

The **Client and Users Handler** binds the clients to the provider, so it has the following functions:

- *Client interface.* The handler implements the common interfaces of the clients[2] to hide the kind of the client to the provider. In section 2.1 it is available a description of the main differences between the two client's versions.
- *Authentication.* The handler implements the methods to recognize the type of user by a code automatically generated by the System Controller at each valid session (typically it is the cookie used by the user or a similar number if he is using the MA).
- *Services showing.* With the authentication module support the handler can decide which class of services can be seen by the user.

[2]see section 2.6 to have a precise description of the interface.

- *Services interfaces* In this component the interfaces for each group of services (there is an hereditary hierarchy to realize them) are defined to allow the handler to bind the desired service by calling the System Controller (for security reason it is only possible to enqueue the request on the System Controller buffer). An other reason for their presence in this component it's the possibility to show that on the presentation tier.

The figure 2.3 shows a simple class diagram that models the Client and User Handler, as it was described above.

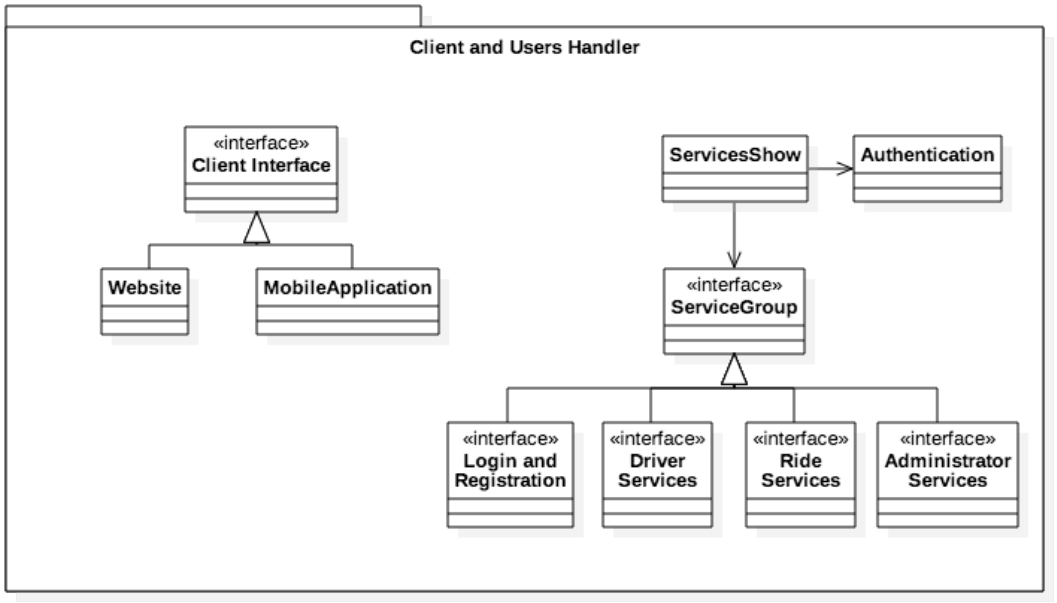

Figure 2.3: A class diagram for the Client and Users Handler.

The **Ride Allocator** is a component involved by the System Controller only when a ride needs to be assigned and when a driver starts to wait for a new ride. Its functionalities are related to the situations described a few words before. The implementation of these functionalities is hidden to System Controller, that invokes them by an interface[3].

[3]In section 2.6 it is described this interface and the external interfaces used to implement the

First, the Ride Allocator creates a representation of the city map split into areas[4]. With this representation and the external APIs it is easy to identify the area where either a position or an address is into.
Second, the Ride Allocator has to manage the queues for each areas, so it has two subcomponents dedicated to this purpose: the Queue Creator which creates and defines a queue for each area; the Queue Manager that implements the methods to handle the queues, so the management policies, the adding or the removing of a driver and so on.
Finally, the Allocator assigns a cab-man to a ride for which the starting position (the destination is only an information for the driver, but not for this component) is passed by the System Controller.
In figure 2.4 the Ride Allocator is modelled by a simple class diagram.

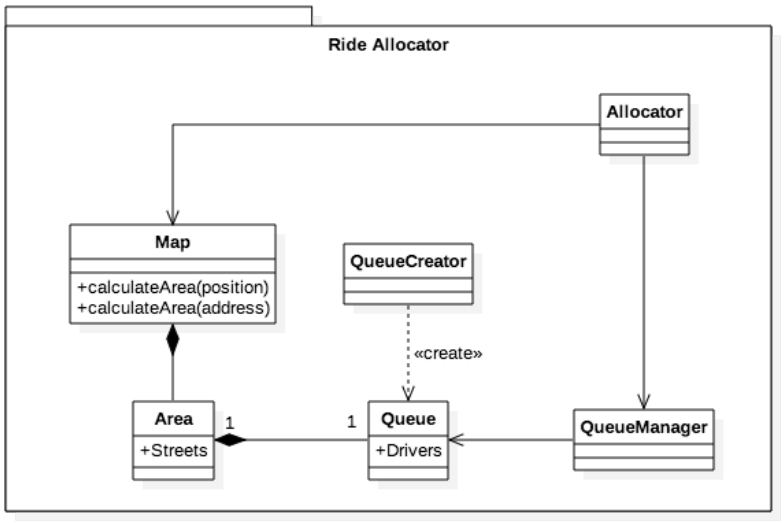

Figure 2.4: A class diagram for the Ride Allocator.

city map and the localization

[4]An important remark on area division is the following. As it is partially described into data tier model, the division into the areas is not made by a set of positions. An area is composed by a set of streets that are closest to each other. This choice can create same strange areas when a street is very long: in this case the area is composed by a set of closer streets that defines a polygonal area and by a street that is include into this polygon, but "fills out"for part of itself.

The **System Controller** is the central node of the application: all the communications pass through it. As consequence it requires a very good design to not slow down the system.
The main subcomponent is the Dispatcher, realized first as an interface with only the method to enqueue a request for another component (for instance an user request coming from the Client and Users Handler), then as a concrete implementation with all the hidden methods. The requests are also handled. The policy to manage the request is a three priority levels FIFO[5] queue, where the first level is reserved to some problems that can occur during system execution and the second one is dedicated to the administrators' operations. To clarify, near the 100% of the message have the low-level priority, but the system is able to immediately react when a fault or an external attack happen (the related messages have the high-level priority).
Second, the User Creator is able to create user by Factory Method pattern[6]. Hence, with some strategies hidden by that pattern the User Creator creates the correct type of user. This fact implies an important consequence to the system: each type of user can create or not particular cookies to allow the Client and User Handler to recognize them and to perform the available operations.
Third, the User Checker is a supplementary component to increase the security of the system and works in parallel to the Dispatcher. When a request is enqueued to the dispatcher low-level the Checker immediately checks the validity of that request and, if any, the request is removed by the queue.
Fourth, the Data Checker has the only role of checking the validity of input data inserted by an user. Thus, it checks the correctness of inserted addresses (in collaboration with the Ride Allocator) and of personal data.
Finally this component contains all the main logic of the system, so all the classes not related to some particular component (for instance all the handlers and the rides' definition and management) is defined here.
In figure 2.5 a class diagram that model this component is shown.

[5]FIFO means *First In First Out*, so the first arrived request is handled before the others. In addition there are three priority levels to allow a more important message to be handled earlier than a low-priority one arrived in advance.

[6]see section 2.7 for a formal description of this design pattern.

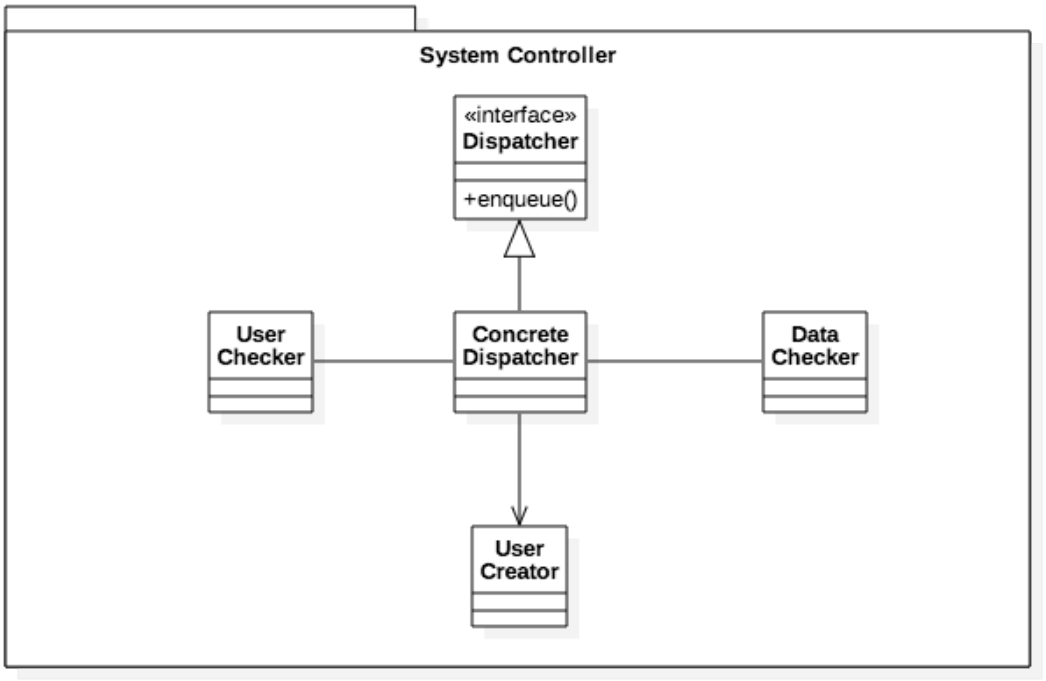

Figure 2.5: A class diagram for the System Controller.

2.3.3 Presentation tier

The available services are grouped into four main section:

- *Login and Registration.* This service is always available at the first access to myTaxiService, both for the MA and the WS. The user (it is not important which kind of user he want to become) has to log into the service or to register himself the first time. When the visitor clicks on login [7], he inserts his personal keys, then the Client and User Handler verifies the correctness of those keys, then checks the type of user to shows his services. (If the user is a driver or a cab company administrator other special credentials will be asked to increase security).

- *Ride services.* This macro section contains all the functionalities related to

[7] see chapter 4 for further information

the rides, so the characteristic ones of myTaxiService. The zerotime and future rides' reservation, the possibility to view or to modify the personal rides and the reading of user's alerts belong to this area and they are all shown together.
The Client and User Handler shows these services to all logged users (except for the administrators).

- *Driver services.* Driver services is a section reserved to the drivers logged into the MA, for the reason presented in section 2.1. The functionalities give here are the management of work shifts, the notifications of availability and the reading of reserved alerts (for instance, the request for a ride or the confirmation of a ride).

- *Administration services.* This section, for security reasons, can be accessed only with administrator credentials (they consists in a couple name and password and in the use of an electronic card) into the company's headquarters on the WS version. The Client and User Handler shows into this section the following functionalities: create or remove a driver profile, notification for all the users (usually, for general information or problems with the service) and checks some service information (for instance anomaly values of taxi queues).

2.4 Deployment view

In this section the deployment of myTaxyService is pointed out. Due to the reasons explained in subsection 2.3.2 the most important component of the system is the Dispatcher, thus this component has a half of the computational power of the system dedicated only to itself. The other components of the provider are located in the same physical location of the System Controller unit, but on various nodes, according to this guide line:

- the remaining computational power is equally distributed among four

other nodes;

- the Ride Allocator and the subcomponents of the System Controller (except for the Dispatcher) are sit in the same node;

- the DBMS and the Client and Users Handler are located on standalone nodes.

Figure Figure 2.6 shows a deployment diagram that focuses on the communications between significant nodes and on the presence of the nodes themselves.

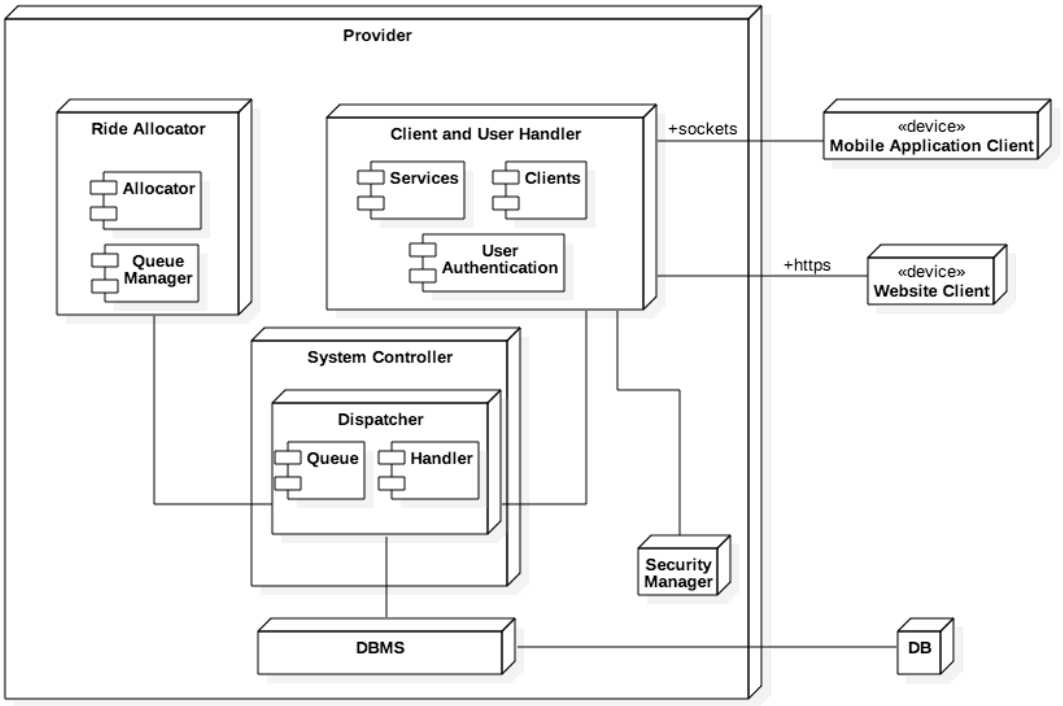

Figure 2.6: A deployment diagram for myTaxiService.

In addition to this short description, two firewalls are located to improve security: one out of the Provider (the communication links outgoing from the Client and User Handler to the Clients) and one on the communication link be-

tween the Dispatcher and the Client and User Handler.
At last, in figure the database is represented as a node: into the designed architecture the database is replaced twice to prevent loss of data and these two portions are located in different places with respect to each other and to the physical location of the provider architecture. To be precise these two position are into a circle with radius fifty kilometres and centre on the provider's location. The policies related to access the database, to store and to backup all the data (into other location) are defined by the DBMS, that communicate with the two database thanks to an encrypted connection.

2.5 Runtime view

In this paragraph we'll show the main features on the runtime view, which means the UX diagrams and the sequence diagrams.
The UX diagrams display the structure of the myTaxiService's screens (input forms, available functions and shown data) and the effects of each function (the next screens viewed). The sequence diagrams show the sequence of functions' calling and screens to benefit of a functionality.

An important note is the following: the functionality Read The Alerts[8] is not shown here. It is performed as a simple screen on the Homepage that redirect to another screen with only the list of alerts (composed by a title and a written text).

Finally, the two version of myTaxiService (WS and MA) have no differences on the visualization of the screen. In fact, the screen has the same name and the same purpose, but they are adapted to the particular version where they are displayed.
In particular, the sequence diagrams are usually referred to the WS version to define the labels of each arcs, but the same labels can easily "translated"to the MA version. For instance, in figure 2.5.2 the first edge has as label "*navigate to*

[8]see the RASD for further information.

myTaxiService websites". In the MA this sentence is equivalent to "*open myTaxiService's application*".

2.5.1 UX Diagram

Starting Page

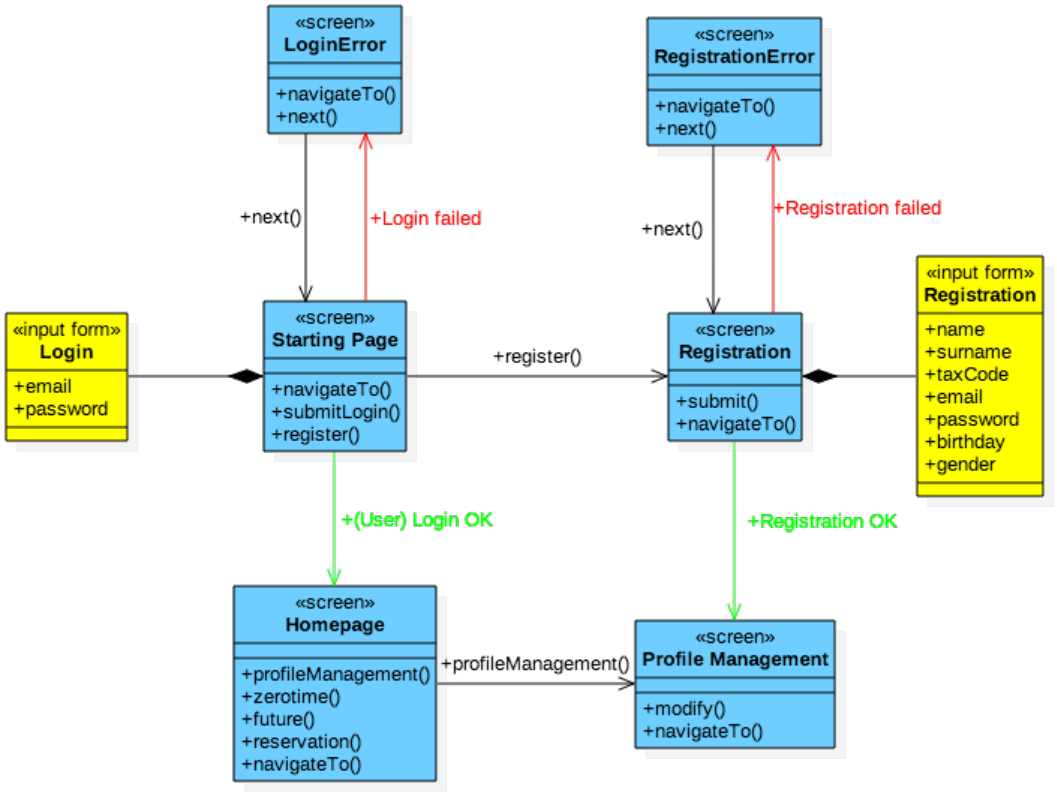

Figure 2.7: The UX diagram from the Starting Page. The main functionalities available from this screen are displayed here.

Profile Management

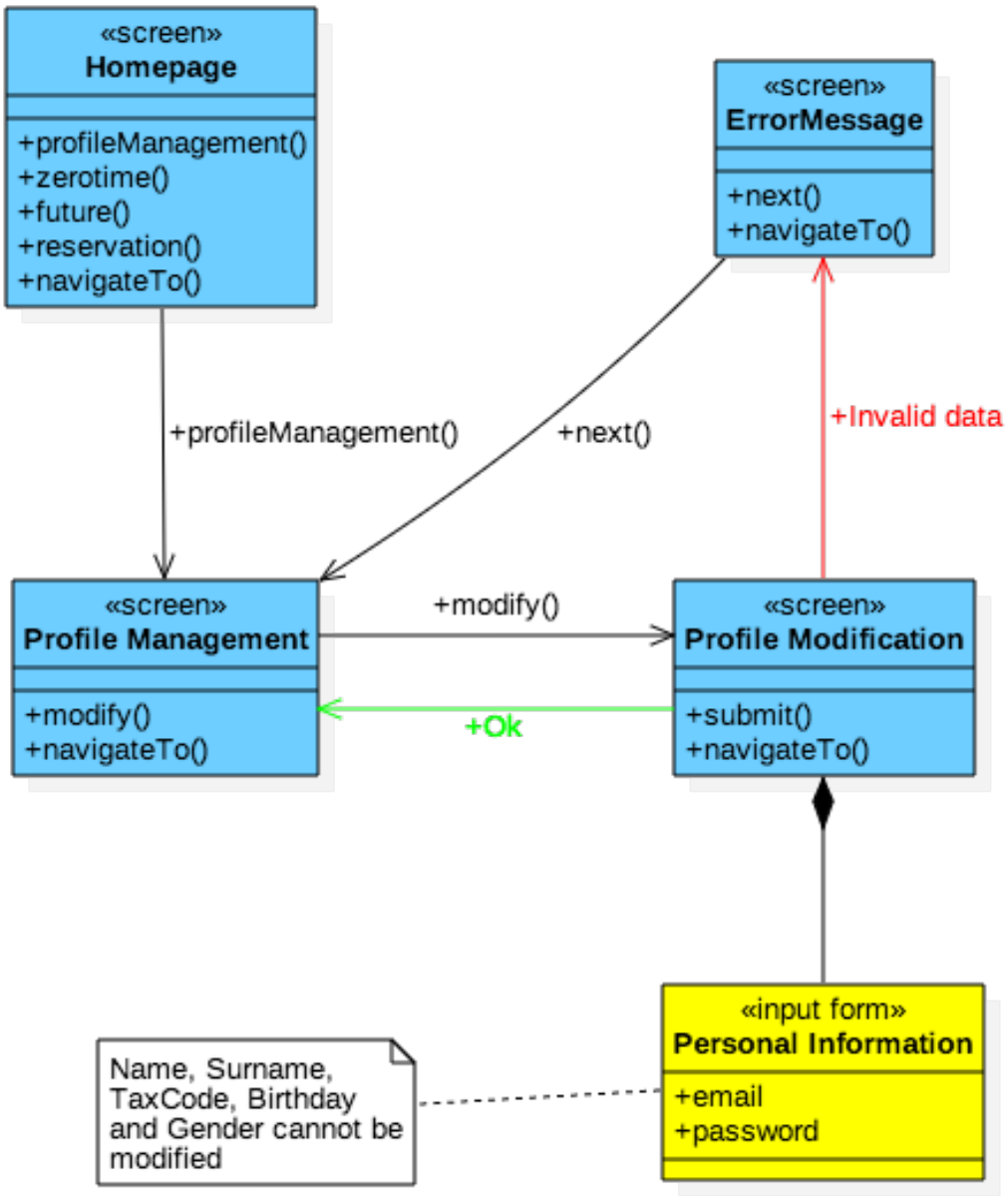

Figure 2.8: The UX diagram for the management of the personal profile. All the possible operations are displayed here.

Zerotime and Future Rides

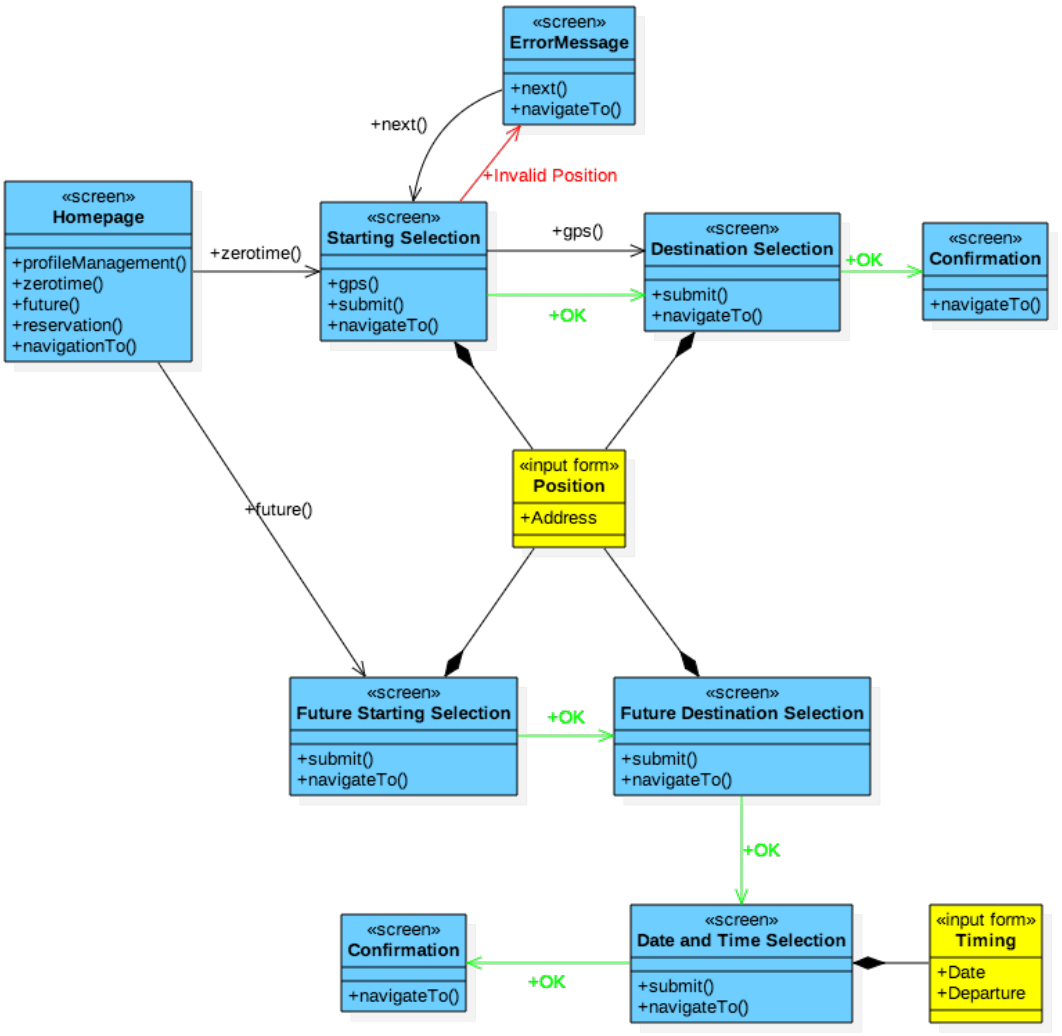

Figure 2.9: The UX diagram for both the booking of a future ride and the asking for a zerotime ride, starting from the homepage.
In order to make the diagram easy to understand, not all the error screens are inserted into the diagram: only the first one is shown.The other ones are similar and where there is an input form.

Check The Reservations

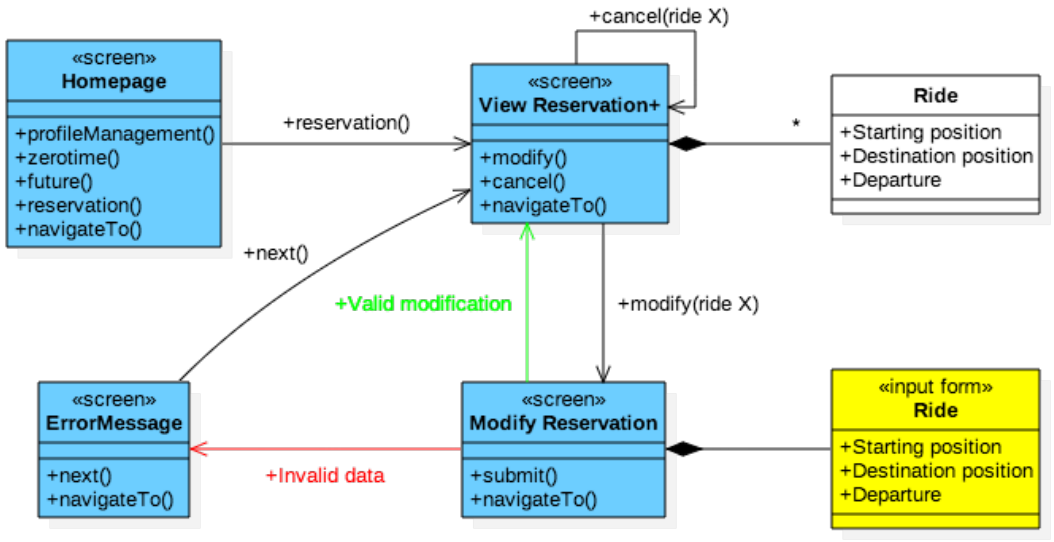

Figure 2.10: The UX diagram for the management of the personal reservations.

Driver Functionalities

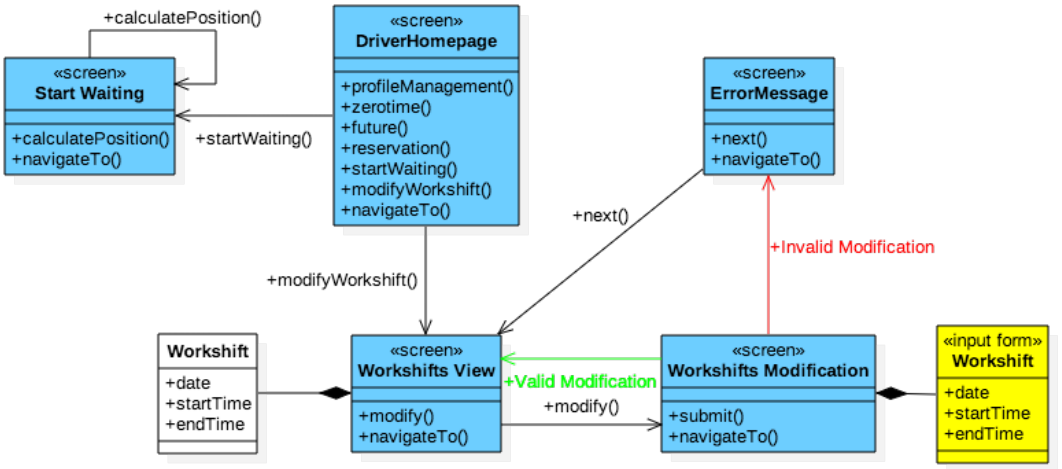

Figure 2.11: The Ux diagram for the Driver functionalities.

2.5.2 Sequence Diagram

In this section are displayed the sequence diagram of myTaxiService run-view. In each diagram where the actor is an user and not a visitor of the website there is always an edge with "Login Procedure"that can be misunderstood: the login to the websites it is required to perform the desired action, but if it is already done (for instance when the user is on his homepage after another operation) the login it is not requested again.

Registration

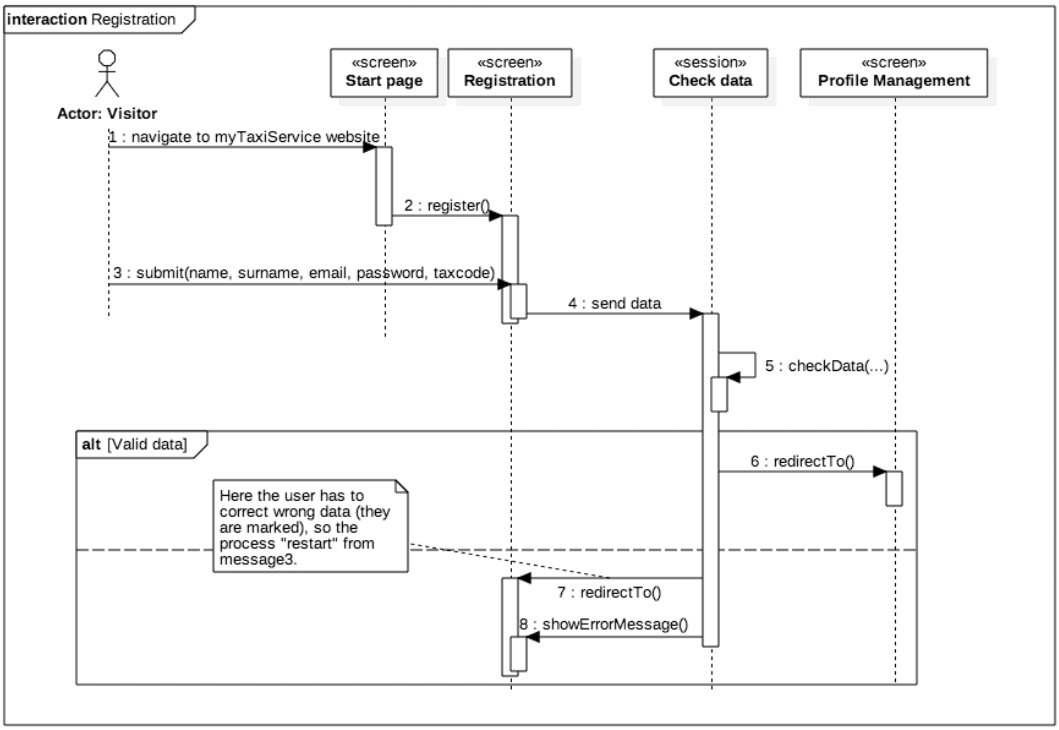

Figure 2.12: The sequence diagram for the Registration.

Login

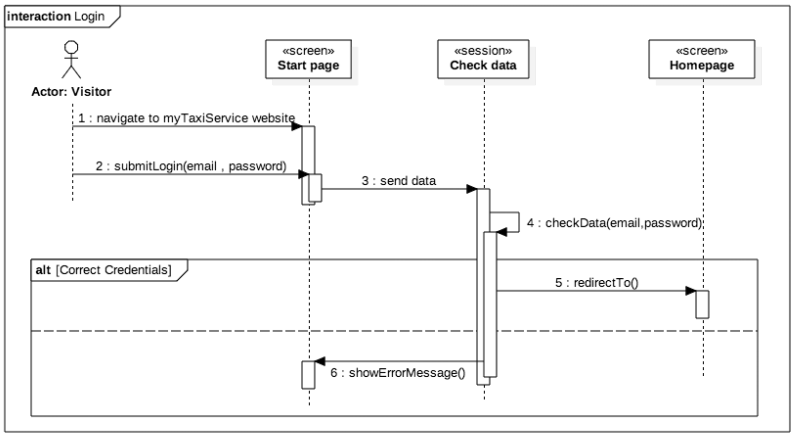

Figure 2.13: The sequence diagram for the Login.

Profile Management

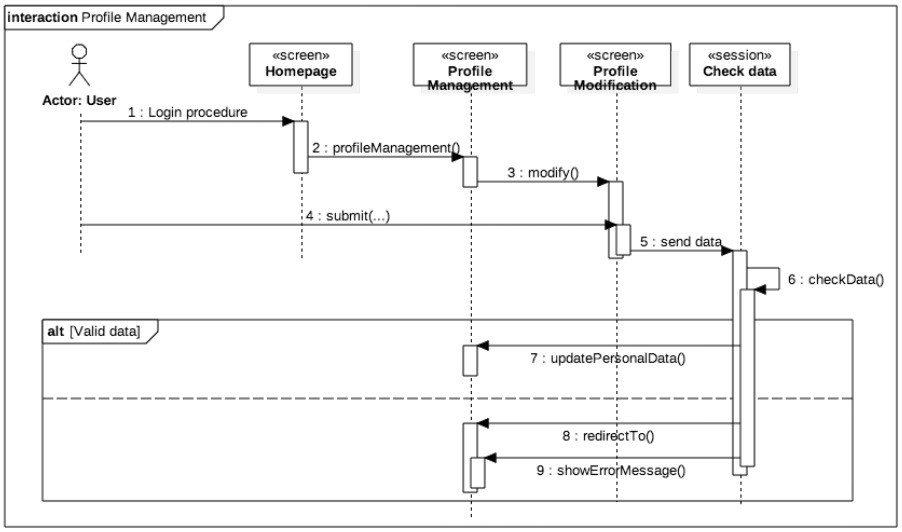

Figure 2.14: The sequence diagram for the management of the personal profile.
The only data that can be modified are the email and the password used to log into the system, while name, surname, tax code, gender and birthday are not modifiable by definition.

Check The Reservations

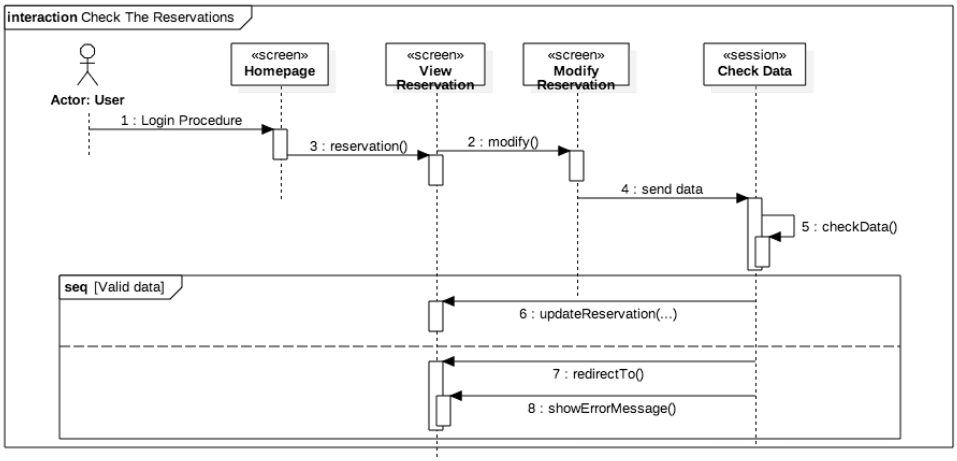

Figure 2.15: The sequence diagram for the management of personal booking.

Start Waiting Time

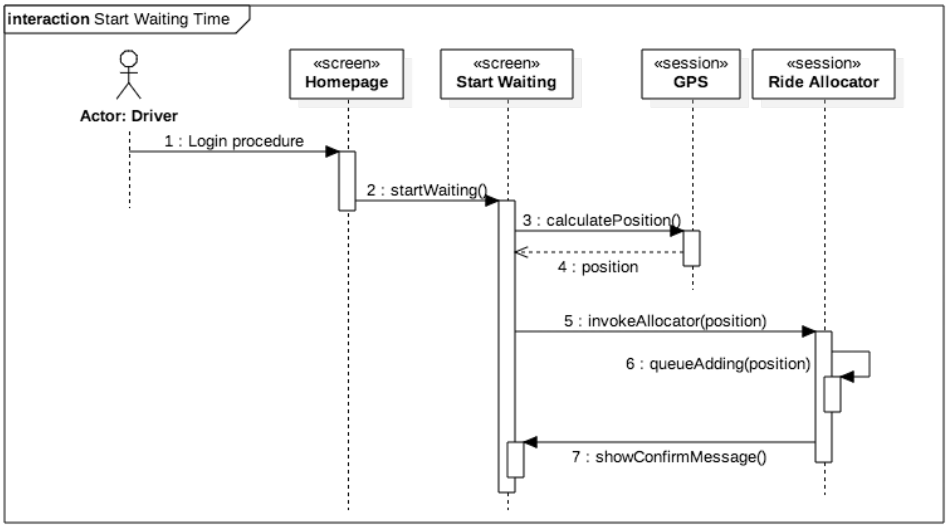

Figure 2.16: The sequence diagram for the notification of waiting time's start.

Work shifts Management

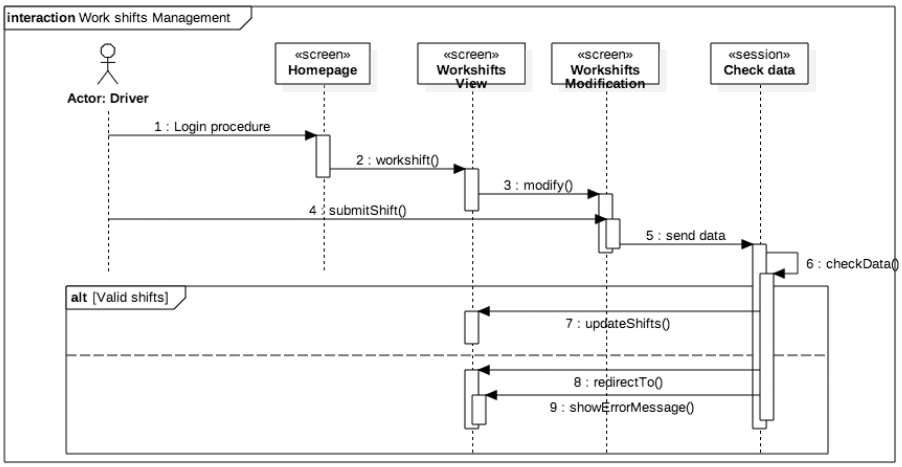

Figure 2.17: The sequence diagram for the management of the driver's work shifts.

Future Ride Booking

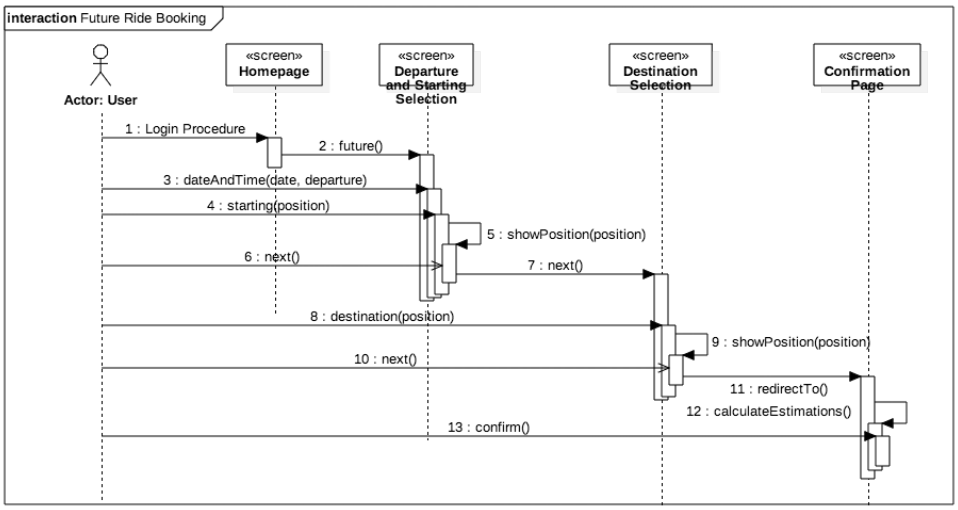

Figure 2.18: The sequence diagram for the booking of a future ride.
In order to make the diagram easy to understand, the errors on inserting either the positions or the departure are not modelled.

Zerotime Ride

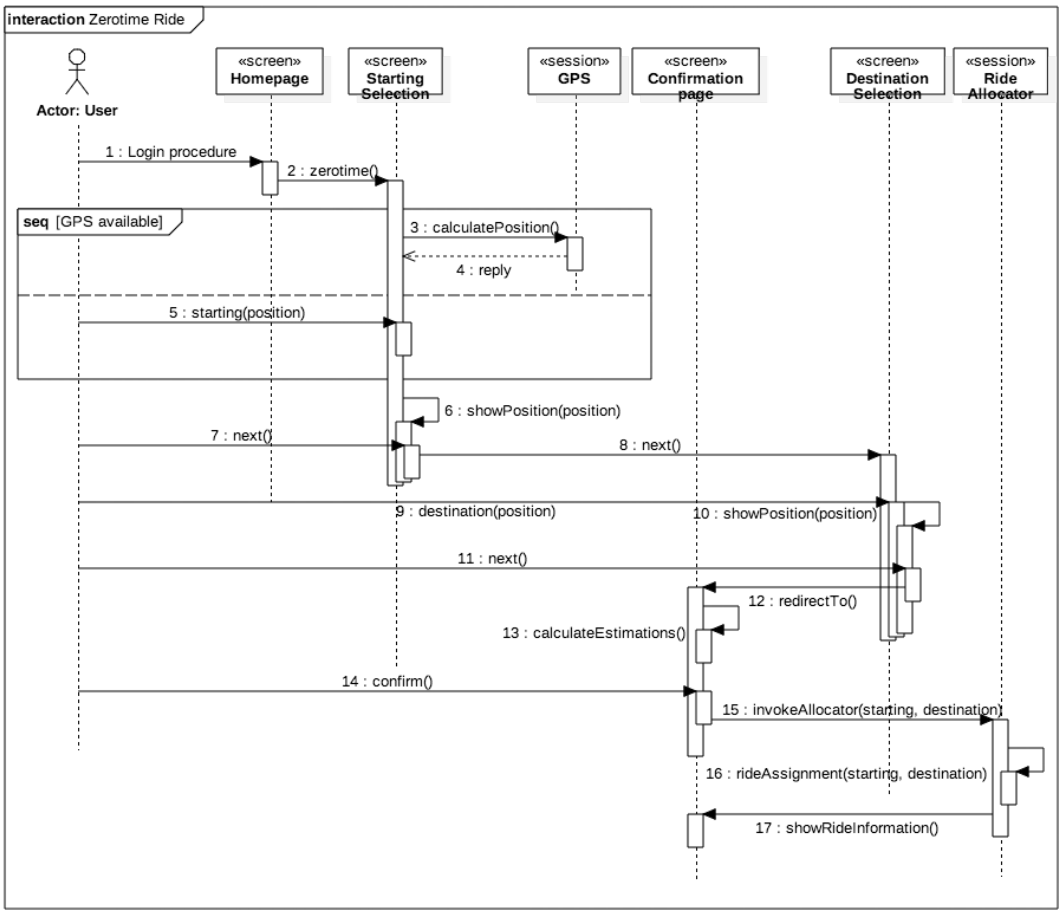

Figure 2.19: The sequence diagram for the asking for a zerotime ride.
In order to make the diagram easy to understand, the errors on inserting the positions are not modelled.

2.6 Component Interfaces

This section describes the interfaces of the components, i.e. the operations they offer to the external world.
The methods are written in the following way:

return_type method_name ([parameter1_type parameter1_name], [...])

- **GpsInterface**. This interface is implemented by the GPS system. For the GPS functionality we use also the Google Maps APIs to obtain the address corresponding to a gps position and to show the map.
  The GpsInterface offers two public methods:
  - *Position calculatePosition()*
    it calculates the current position of the method's caller and return an object of type Position.
  - *double getGpsLatitude(Position pos)*
    it returns a double precision decimal number that represents the latitude of the Position given as parameter.
  - *double getGpsLongitude(Position pos)*
    it returns a double precision decimal number that represents the longitude of the Position given as parameter

- **RideAllocatorInterface**. This interface is implemented by the Ride Allocator to manage the rides and the queues of available drivers for each area. It offers three public methods:
  - *void assignRide(Ride ride, Driver driver)*
    it assigns Ride given as first parameter to the Driver, given as second parameter.
  - *void enqueueDriver(Driver driver, Position pos)*
    it calculates the Area which the Position of the driver (given as a second parameter) belongs to, then it add the Driver (given as first parameter) to the corresponding queue of available drivers of the area.
  - *boolean isValidAddress(String city, String street)*
    it verifies the existence of the address, given as the couple of String(s) representing the city and the street, in one of the areas covered by myTaxyService. It returns the boolean value true if the address exists, false otherwise.

- **ClientAndUsersHandlerInterface**. This interface is implemented by the

Client and Users Handler. As described in the sections above the services are split into a few interfaces that are grouped by a common role or actor[9]. To simplify this section, we'll list them sequentially without respecting the division in groups.

This interface offers the public methods to manage the user functionalities:

- *String login(String email, String password)*
  this method needs two parameters: a String containing the e-mail of the user and a String containing the encrypted password with the method defined by the SecurityManagerInterface. It returns a String representing the result of the operation with a number identifier at starting and a text message after. All the possible results are:
  - *"0 : Login Successful"*
  - *"1 : User already logged in"*
  - *"2 : Login Error, the e-mail and/or the password are wrong"*
- *String register(String email, String password, String name, String surname, String taxCode, Date birthday, String cityOfResidence)*
  this method needs many String parameters: the e-mail of the user, the encrypted password with the method defined by the SecurityManagerInterface, the name, surname, personal tax code, birthday, the city of residence. It returns a String representing the result of the operation with a number identifier at starting and a text message after. All the possible results are:
  - *"0 : Registration Successful, an e-mail with activation code has been sent"*
  - *"1 : Registration Error, user already registered with that tax code"*
  - *"2 : Registration Error, e-mail address already in use"*
  - *"3 : Registration Error, data are not valid"*
- *String changeEmail(String email)*
  it changes the e-mail of the current logged user that calls the method, with the new one given as parameter. This method returns the result

[9]For each method the checking of correctness of the caller actor is entrusted to the Security Manager.

of the operation as a String containing a number identifier at starting and a text message after. All the possible results are:

  - *"0 : E-mail Address Changed, an e-mail with activation code has been sent"*
  - *"1 : E-mail Changing Error, the specified e-mail address is already in use"*

- *String changePassword(String oldPassword, String newPassword, String passwordConfirmation)*
  it changes the current password of the current logged user that calls the method, with the new one given as second parameter. The first String parameter contains the old password encrypted with the method defined by the SecurityManagerInterface, the second parameter is the new password encrypted and the third one is the confirmation of the new password (the new password repeated) encrypted. This method returns the result of the operation as a String containing a number identifier at starting and a text message after. All the possible results are:
  - *"0 : Password Changed"*
  - *"1 : Password Changing Error, the new password is not valid"*
  - *"2 : Password Changing Error, the new password and the password confirmation don't match"*
  - *"3 : Password Changing Error, the old password is wrong"*
- *void askForRide(User passenger, Position departure, Position destination)*
  it asks for a zerotime ride for the passenger given as first parameter, with two Position parameters corresponding to the departure and the destination (second and third parameters).
- *void bookFutureRide(User passenger, Position departure, Position destination, Date departureTime)*
  it books a ride for the passenger given as first parameter, with two Position parameters corresponding to the departure and the destination (second and third parameters) and one Date parameter representing the desired date (and also time) of the departure.

- *Ride[] checkReservations()*
  it shows all the Ride(s) (zerotime and future) asked by the current logged User. It returns an array of all the Ride(s).
- *void deleteRide(Ride ride)*
  it deletes the ride given as first parameter (as already said it's possible only if the ride is a future reservation)
- *void changeRideTime(Ride ride, Date departureTime)*
  it changes the departure date and time for the ride given as first parameter, with the new one given as second parameter.
- *Object[][] readAlerts()*
  it shows all the alerts to the user. The returned type is a matrix of Object(s) containing the date and the message in the columns for each alert in the rows.
- *void acceptDenyRide(Ride ride, boolean accept)*
  it accepts or deny a Ride, given as first parameter, for the current logged Driver. The first parameter is the Ride to accept or deny, the second parameter is the boolean value representing the desired choice, it is true if the driver wants to accept the ride, false otherwise.
- *void startWaitingTime(Position pos)*
  it starts the waiting time for the current logged Driver in the Position given as parameter, i.e. the driver is available and waiting for a ride request in the area associated to the provided position.
- *void addWorkShift(WorkShift ws)*
  it adds the WorkShift given as parameter (as described in the Class Diagram of RASD) characterised by a week day number, the starting time and the ending time.
- *void removeWorkShift(WorkShift ws)*
  it removes the WorkShift given as parameter (as described in the Class Diagram of RASD) characterised by a week day number, the starting time and the ending time.

- **SystemControllerInterface**. This interface is implemented by the System

Controller and offers only one public method:

  - *void enqueueRequest(Request req)*
    it adds a Request, given as first parameter, to the queue of the sub Component Dispatcher.

- **SecurityManagerInterface**. This interface is implemented by the Security Manager and offers these public methods:
  - *String encryptPassword(String password)*
    it encrypts a password given as a String parameter, with the hashing method estabilished by the security rules of the system, and returns the result.
  - *boolean checkValidityOfRequest(Request req, User user)*
    it checks the validity of a Request (first parameter) from a User (second parameter). It returns the boolean value true, if the user can submit the request, false otherwise.

- **DBMSInterface**. This interface is implemented by the DBMS and offers general public methods to access the data of the database:
  - *Object getValue(String key, String from, String whereCondition)* gets the value corresponding to a specific key of a record in the database. The first parameter is the name of the key of which we want to know the corresponding value, for a specific record. The second parameter is the name of the table to retrieve data from or a join of tables. The third parameter is a part of SQL query containing the conditions to search the desired record.
    The method returns the Object representing the value corresponding to the specific key of the desired record. If there are more than one records according to the whereCondition (third parameter) only the first record is taken.
  - *Object[] getValue(String[] keys, String from, String whereCondition)* this is an overload method similar to the one above, but it returns an array of Object(s) corresponding to the values of the keys in the array

of String(s) given as first parameter. The meaning of the other parameters is the same of the method above. Again if there are more than one records according to the whereCondition (third parameter) only the first record is taken.

- *Object[][] getAllValues(String[] keys, String from, String whereCondition)* this method is similar to the one above, but it returns all the values corresponding to the specific keys of all the records that respect the whereCondition (third parameter), not only the first record as it was above instead. The meaning of the parameters is the same of the method above. This method returns a matrix of Object(s) which contains all the records as rows and all the values corresponding to the keys as columns, in the same order of the array of keys given as second parameter.
- *void setValue(String key, String value, String from, String whereCondition)* sets the value of a specific key of any record in the database that respect the whereCondition parameter. The first parameter is the name of the key of which we want to set the value, given as second parameter, for a specific record. The third parameter is the name of the table or a join of tables, where the desired key is. The fourth parameter is a part of SQL query containing the conditions to search the desired record.
- *void setValue(String[] key, String[] value, String from, String whereCondition)* this is an overload method similar to the one above, but it sets the new values for many keys at the same time. The first parameter is the array of all the key names to modify, the second parameter is the array of all the new values for the keys (considered in the same order of the corresponding keys). The other parameters have the same meaning as before.

2.7 Selected architectural styles and patterns

For myTaxyService we have chosen to follow two fundamental architectural patterns combined, Model-View-Controller (also called MVC) and Service-Oriented Architecture (also called SOA).
The pattern MVC, as its name says, consists in the clear division between three main interconnected software parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. The three parts are:

- *Model.* It represents the underlying, logical structure of data in a software application and the high-level class associated with it. The Model does not contain any information about the user interface.
  It communicates with the Controller, which can read or modify the data contained. The only interaction with the View is the notification of the data changing in some applications of MVC pattern.
- *View.* It is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth). Usually the View is generalized and made extensible with the use of an interface with public methods, implemented by different view implementations for various user interfaces.
  The View doesn't contain any information of the real data contained in the Model, it knows only the information that receives from the Controller and sends to user input to it.
- *Controller.* It represents the bridge between the Model and the View, and is used to communicate between their classes. It receives the user inputs from the View and as consequence retrieves or modifies the data contained in the Model. It contains information of the Model and of the View and knows how to interact with them.

A Service-Oriented Architecture is composed of three parts:

- *Service requestor.* It requests the execution of a service. It asks for a service contained in the Service registry and once that is found, the Service requestor is bound to the correct service offered by the Service provider.

- *Service provider.* It offers services. It publish all the available services with their description in the public Service registry and when receives a request it bind the requestor with the correct service.
- *Service registry.* It provides information about all the available services.

Our application and adaptation of MVC and SOA for myTaxyService, is fully described before in section 2.2.

According to the class diagram presented in the RASD, we have chosen to follow also some design patterns, such as the Factory Pattern and Singleton.
The *Factory Pattern* consists in the creation of objects without exposing the creation logic to the client and referring to newly created object using a common interface or super class. In the Factory Pattern a specific class (a Factory class) offers public methods to create objects using a common interface or abstract class and it knows the creation logic of the concrete implementation of the common interface. The client that calls the factory methods (the methods used to create objects) of the factory class, passing a parameter identifier of the required object, don't know the real creation logic of the object, they knows only the Factory class and its factory methods and the common interface or abstract class which the desired object belongs to.
In the case of myTaxyService we use this pattern to create a User of a specific type (Driver or normal User) and also a Ride of a specific type (zerotime or future ride), without knowing in fact the real create logic of the users and rides depending by the type of them. There will be a factory class for the users and one for the rides that know the real create logic of them.

The *Singleton Pattern* involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. The constructor of the class is hidden to the external world to make sure that the only way that an instance of the object can be obtained is by a static method of the class that doesn't create more than one instance of its object.

We use the Singleton pattern for the internal components of the Provider to make sure that have always only one instance of them and they can be easily accessible with a static method. (for more information about the provider see subsection 2.3.2).

Chapter 3

Algorithm Design

In this chapter the most interesting algorithm of myTaxiService are presented using pseudo-code (every developer can easily translate the pseudo-code into the desired programming language). In addition this pseudo-code is referred to an object-oriented programming language, like C++ or Java (the reader can think about the equivalent algorithm in non object-oriented programming language like C, so he has to create manually all the object data structures, the objects himself and he has to define a way to manage and save all the created objects).
The meaning of word *interesting*, used above to define the algorithms which we will present in this chapter, is the following: a characteristic and unique algorithm, used to implement a specific functionalities of this system. For instance, an algorithm to manage the backup of the database can be very complicated in describing the policies, the exceptions and the situations when execute its, but it is a common algorithm for all the systems that store data into a database.

In this chapter the algorithms analysed are the following:

- the city's areas creations and management;
- the queue's management;
- all the algorithms used to handle the special situation which occur when no cabmen are available in the area where the ride (which the Ride Allocator is now assigning) starting position is.

3.1 Map and Areas' Creation Algorithms

Premise. The system has, at its disposal, an XML document that describes the city and its streets. The structure of the document is shown below and it is the following: first there are the four extreme coordinates in order to create a rectangle which contains all the city's area (if the borders of the city are irregular the rectangle's area is bigger than the city's area), then all the streets are listed with the name and two coordinates. In fact, with this two points the street can be represent as the line that joins these two points.

```
<City>
    <Extreme Coordinates>
        <WestX> . . . </WestX>
        <SouthY> . . . </SouthY>
        <EastX> . . . </EastX>
        <NorthY> . . . </NorthY>
    </Extreme Coordinates>
    <Streets>
        <Street direction = "H">
            <Name> . . . </Name>
            <Left Position>
                <X> . . . </X>
                <Y> . . . </Y>
            </Left Position>
            <Right Position>
                <X> . . . </X>
                <Y> . . . </Y>
            </Right Position>
        </Street>
        [. . . ]
        <Street> [. . . ] </Street>
    </Streets>
</City>
```

Figure 3.1: Structure of XML document that describes the city.

For each street an attribute direction is defined: the value H indicates a horizontal road so the two coordinates are the left and the right, while the value V indicates a vertical road so the two coordinates are the high and the low.

When the system starts the map creation it first creates the areas objects. The idea is simple: starting form the Northwest angle of the map it create square areas of side 1.5 kilometres[1].

```
MapCreator.createMap(WestX , SouthY , EastX , NorthY);
horizontalSectors = (EastX - WestX) / 1.5 + 1;
verticalSectors = (NorthY - SouthY) / 1.5 + 1;
for i = 0 -> verticalSectors then
        for j = 0 -> horizontalSectors then
                MapCreator.createArea(j , i);
```

We point out two aspect of the code shown above: first, in the calculus of the number of areas, both in vertical and horizontal, we sum one at the result to count the final area which has a size less than the fixed dimension (1.5 km); second, the notation used in the *for* indicates a cycle of n interactions where n is equal to the value written at the right of the edge.

Now, the objects of type *area* are created, but they do not contain any street. The algorithm used to add the streets is simple: for each street into the XML document (we suppose we have a parser which gives all the streets found into the document as an object), the belonging area is the one where the first coordinates is in. To avoid strange situations where a street is assigned to an area even if only a small part belongs to the area[2] an additional parameter *CORRECTOR* is defined. The parameter assumes on value into the interval [0,1] (the value we have chosen is 0.7). To assign a horizontal street (for the vertical ones is similar)

[1] Due to the city is not perfectly rectangular, this algorithm can create some areas that cover lands out of the city borders. This is not a relevant problem in memory usage.

[2] This situation happens, for instance, when the left coordinate is near the right bound of the area and it has an horizontal direction.

the rules are the following (the map can consider as a grid):

- the row into the map is exactly the one where the left coordinate is in;
- the column into the map is exactly the one where the left coordinate is in if and only if its position is not near the right bound (on the other hand the street is assigned to the next area on the right). Called *size* the dimension of an area and x the distance between the coordinate and the starting of the area *start*, the coordinate is near the area's right bound if:

$$x > start + size * CORRECTOR;$$

Now the algorithm is shown by restricting the use of chain invocations in order to make the algorithm easy to read.

```
//We suppose we have an iterator between the streets, given by the
parser. From now we'll call it parserIt.

  while (parserIt.hasNext()) then
        street = parserIt.next();
        x = street.firstCoordinate().getX();
        y = street.firstCoordinate().getY();
        row = (x - WestX) / 1.5;
        col = (NorthY - y) / 1.5;
        if ( street.type().equalTo('V') && checkY(y , row)) then
              row = row + 1;
        else if ( street.type().equalTo('H') && checkX(x, col)) then
              col = col + 1;
        Map.getArea(row,col).addStreet(street);
```

Now the two methods checkY and checkX are shown.
Both the methods have a boolean return type and try to verify it is needed to increase by one the calculated area (see above for the reasons). The parameter are different according to the type of the street: for a vertical road are required the y-part of the first coordinate and the calculated row while for a horizontal road the x-part of the first coordinate and the calculated column.

```
boolean checkY (double y , int row) {
        start = NorthY - row * 1,5;
        if (y >(start + 1,5 * CORRECTOR)) then
                return true;
        return false;
}

boolean checkX (double x , int col) {
        start = EastX + col * 1,5;
        if (x >(start + 1,5 * CORRECTOR)) then
                return true;
        return false;
}
```

This algorithm generates a first division of the city and its streets into the areas. The algorithm is not perfect and does not consider same particular situations as restricted-traffic zones or busy roads. To improve the quality of the city areas the administrators are able to move some streets between two areas. The administrators are supposed to decide to change a street with some criteria, so no checks are performed on those action.
Finally, the final version of the city is saved on the database, even if a representation is maintained into the Ride Allocator (to be precisely the constructed version).

3.2 Queue Creator Algorithms

The Queue Creator is a subcomponent of the Ride Allocator[3].
The queues' creation is an iterative process performed at the Ride Allocator creation and initialization. After the definition of the map and its areas, the Queue Creator is involved to create one queue into each area.

[3]see the subsection 2.3.2 for a complete description.

```
forall Area a in Map then
    a.createQueue();
```

The *createQueue* method is defined into the class Area: it is able to create one object *queue* if and only if it has never created another queue yet. This definition does not require a Factory Method[4] pattern, because exists only one type of queue and no other types can be designed in future. In addition, an exception will be thrown if the method is called when a queue already exists[5].

Into the database this method has no effects because no driver is waiting at the creation of the queues that are empty.

3.3 Queue Manager Algorithms

The Queue Manager has two main algorithms: one to add a driver to a queue by a position and the other one to update a queue (moving down or remove a driver).
When a driver needs to be added to a queue, first the Queue Manager uses the Google Maps APIs to find the street by the position, then identify the area by a method into the Map component. (not shown here). At last is able to add the driver to the correct queue in the last position.

```
void addDriver (Driver d , Position pos) {
        street = GoogleMaps.getStreetByPosition(pos);
        area = Map.getArea (street);
        area.enqueue(d);          //The method enqueue does not
involves any methods into the DBMS. The System Controller, when
asks for a driver adding, also call a similar function into the DBMS.
}
```

[4] see the section 2.7 for a definition.

[5] In a non object oriented programming language this error can be notified using a return value of the method createQueue. However this method can be called only by the Queue Creator, that is involved only once.

When a driver has to be removed from the queue, the Queue Manager gives a specific method which is so simple: it removes only the first element into the desired area's queue. In addition, the method also gives the possibilities to move the driver from the first position to the last one by a flag. The modification need to be stored into the database, so the Ride Allocator, at his return, passes to the System Controller the results of the operations and the DBMS is able to correctly update the queue information. Hence, the Queue Manager only updates the queues, but it does not save any data.

```
void moveDriver (Area a , boolean moveDown) {
        driver = area.removeFirstElement();
        if (moveDown) then
                area.enqueue(driver);
}
```

3.4 Ride Assignment Algorithm

The ride are assigned by the Ride Allocator and inside that, by the Allocator subcomponent. The allocator can access to the Map and, as consequence, to the areas and to the related queues. When the System Controller involves the Ride Allocator to assign a driver to an imminent ride, it delegates also the possibilities to pass through the Client and Users Handler in order to communicate with the drivers, then it waits for the results (they will be saved on the database).
The method used to assign a ride has only two parameters, the starting and the destination position. In an object oriented programming language this method will be defined using the overload principle to allows the possibility to use as starting position either a position or an address. For the destination position the method can accept only addresses. In this description we suppose that the

method receives only addresses[6].
The first passage for the algorithm is to find the correct area associated to the address and in this case the methods of the Map are used, as in section 3.3. Afterwards, the Ride Allocator calls the first driver in the queue and waits for his answer one minute.[7] Here, there are two possibilities: first the driver accepts, so the Ride Allocator immediately removes the driver from the queue and returns to the System Controller; else the driver denies, so the Ride Allocator move the driver to the bottom of the queue and then asks to the next driver into the queue. In both the possible cases the Ride Allocator has a list (the real implementation of this list is not given here) where save the key information about the ride that it is now assigning: so the detected area and the sequence of driver involved. The answers of each driver are not needed because only the last driver in the list has accepted the ride.

```
list assignRide (Address start, Address destination, linkTo-
ClientAndUsersHandler handler) {
        area = Map.getArea(start);
        listToReturn.add(area);
        do
            driver = area.removeFirstElement();
            area.enqueue(driver);
            //Above the driver is re-added to the queue
            listToReturn.add(driver);
            accepted = handler.ask(destination, timer value);
        while (accepted == false);
        area.removeLastDriver();
    }
```

[6]in the version with position, the reader can see the section 3.3 to have an idea about the "conversion"from a position to an address.

[7]In the RASD document the driver has been supposed to answer immediately to a call, but to avoid some infinite waiting of the system (for instance for connecting problems on the driver's device) we define now a timer after which the cabman's answer is consider to be a deny.

Observation: the method ask of the handler asks to a cabman if he want to drive in a ride to a certain destination. The second parameter is the maximum time that the driver has to answer: if the timer expires the method return false (the notification of this event to the cabman is then handled by the Client and Users Handler).

3.5 Special Algorithms

In this section will be given only an accurate description of the algorithms used to administrate the special situation that occur when in some area there is no driver into the queue.
In the previous section, the algorithm that assign a ride does not consider this possibility. In some objected oriented languages this can be accepted and the special algorithm is invoked by an exception. In alternative is possible define a more exhaustive control flow that call the special algorithms into an if condition.
The system does not allow an human-defined sequence of near areas to be called, so in this situation the first area chosen to find a driver will be randomly selected among the areas closest to the one where the starting position is in.
After the choice of the next area the algorithm shown in section 3.4 is called to find a driver for the ride. Obviously, if also this area has no available driver, a new area will be chosen to search a driver who is waiting.

3.6 Conclusion

In this chapter all the algorithms have been presented without consider the possibility to receive more than one rides to be assigned at the same time. If the two or more rides are in different areas, no problem occurs even if the algorithms are called in parallel.
A precisation about the case that happen when two contemporaneous rides start from the same area is required. Calling in parallel the shown algorithm to assign the rides may lead to inconsistent and undesired situations. This prob-

lem can be easily solved by the introduction of synchronization strategies. The implementation of this strategies will not be discussed here, but to have an idea a possible (partial) solution is to have more than one thread to execute ride assignment. When the Ride Allocator is waiting for the driver's answer, it can accepts new requests on an other thread and, if the same area is involved the following driver in the queue will be involved, and so on. Note that this description does not describes all the aspects concerning the synchronization: for instance what happen if the first thread need another driver at the same time of the second one? In addition, if there is only one driver into the queue and he rejects the first ride, the system has to asks to him before enter into the special mode (this is a very strange and rarely case).

Chapter 4

User Interfaces Design

In this chapter are shown the main mockups of myTaxiService, due to present the graphical user interfaces of the system for each functionality. All the mockups are displayed with an accurate description to prevent misunderstanding and to recall the functionality.

4.1 Registration/Login

When an user connects to the WS or opens the MA, the first screens is the login page. For both the versions, there is a button to redirect to the registration page, if the user is not registered yet.
The figure 4.1 is the WS version where the registration button is made by a link on the phrase "Not Registered yet? Click here". At the same time the user can insert his credentials into the specific boxes.
The figure 4.2 is the corresponding mockup for the mobile application: at the application opening the user can inserts his credentials or clicks on the registration button.

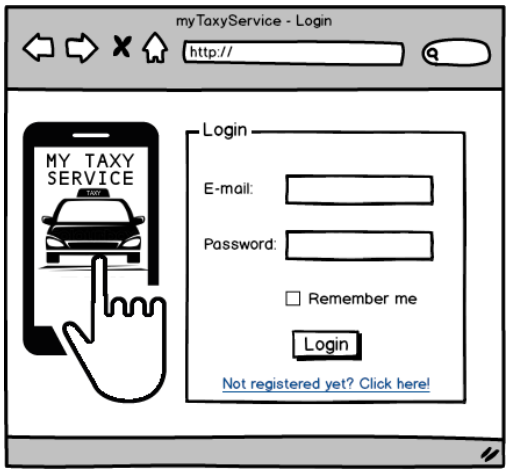

Figure 4.1: Login page into website.

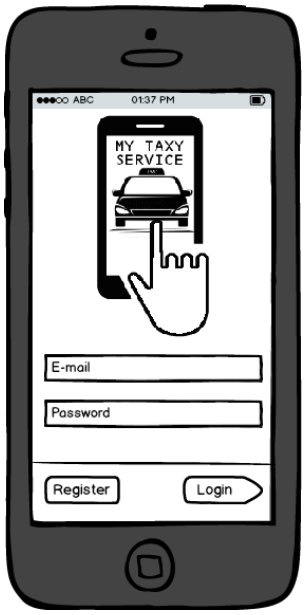

Figure 4.2: Login page into mobile application.

A non registered user has to enrol into the system, so he clicks on the corresponding button and he is redirected to that page. The figures 4.3 and 4.4 shows the mockups for this: into the MA the box labels are written into them and disappears when the user fills in; into the WS the labels are shown near the corresponding one.

4.2 Personal Information Management

The figures 4.5 and 4.6 show the mockups for the management of personal information.
The only information that can be modified are the email, the password and the city of residence. In both the myTaxiService's versions the other information are shown into a non-modifiable boxes (so they have a different color, typically grey, to mark this property).

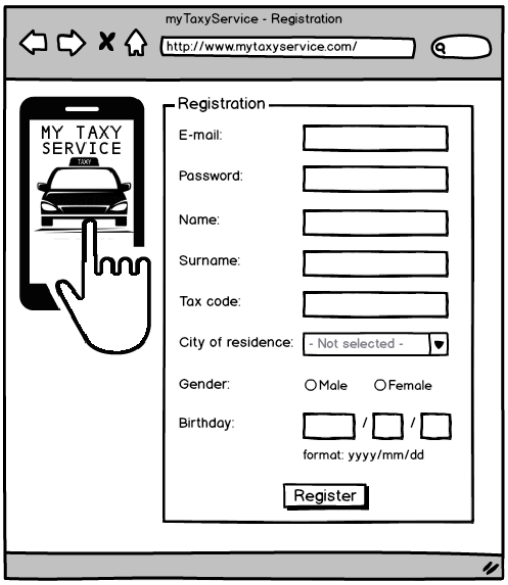

Figure 4.3: Registration page into website.

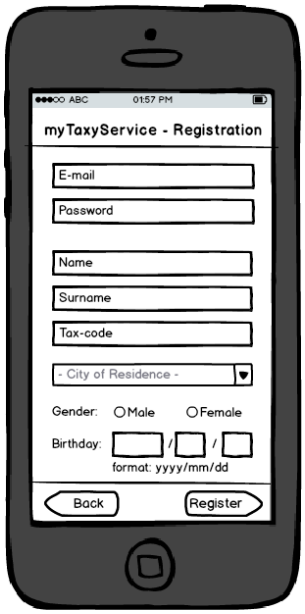

Figure 4.4: Registration page in mobile application.

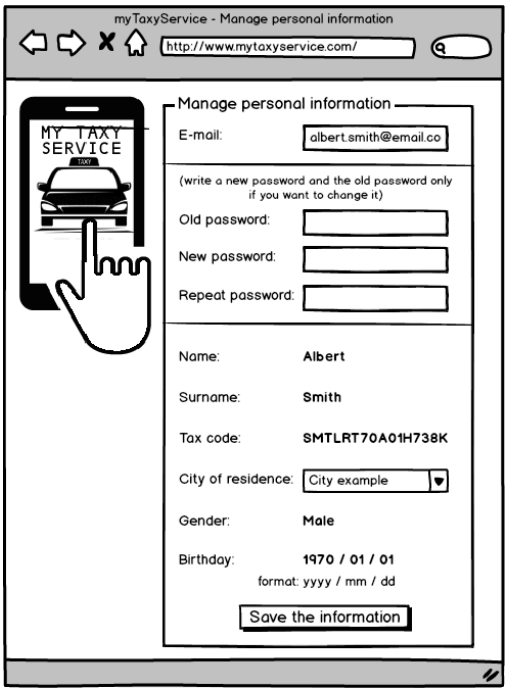

Figure 4.5: Personal Information Management page into website.

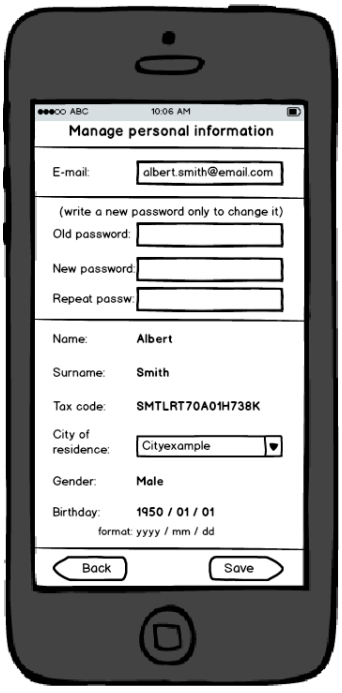

Figure 4.6: Personal Information Management page in mobile application.

4.3 Driver functionalities

The functionalities reserved for the cab-men are developed only for the MA version of myTaxiService: the reasons are explained in section 2.1. When a cab-man has to notify the system that it is starting his work shift or he has ended a ride, so he is now waiting for a new ride, he uses the Start Waiting Time functionality that is shown in figure 4.7: the displayed position is calculated by GPS.

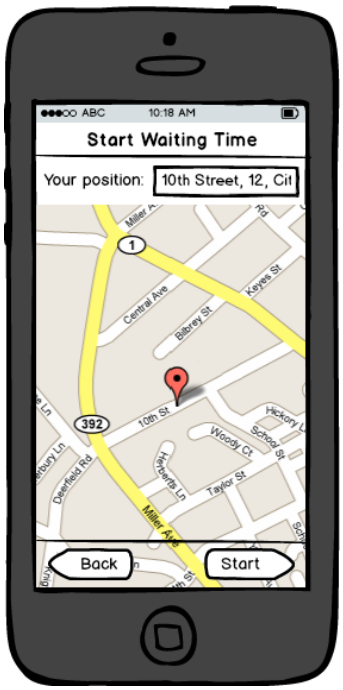

Figure 4.7: The Start Waiting Time page.

Figure 4.8: The request for a ride page.

During this period, the system can asks the driver to make a ride, so in figure 4.8 it is shown the screen for this request.
At last, in figure 4.9 there is a mockup for the page where the driver can modify his workshifts. He can add, modify or remove them. The page where the driver can see his workshifts is not displayed, but it is similar to the shown one, without the modification symbols.

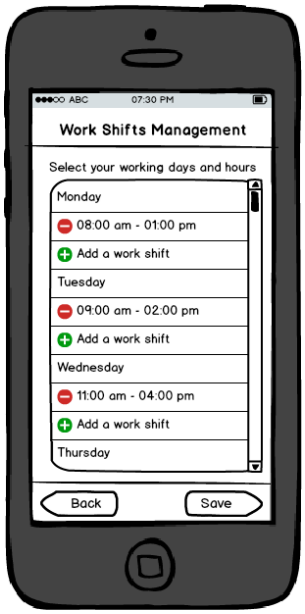

Figure 4.9: The workshifts management page.

4.4 Zerotime Ride

When a user asks for a zerotime ride, he has to insert the starting and the destination positions (the first one can be calculated by the GPS) and then to confirm the request.
In the MA this sequence of operations is performed with three screens (mock-ups in figures 4.10, 4.11 and 4.12), while into the WS it is used only one screen split into the correspondent sections, shown in figure 4.13.

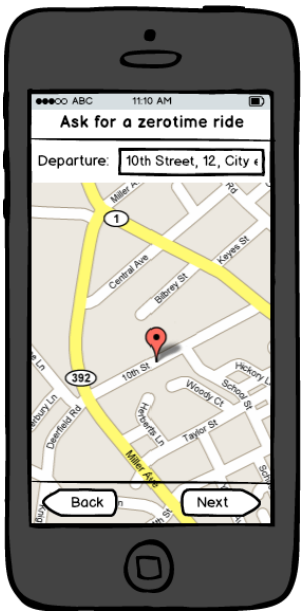

Figure 4.10: Zerotime ride request into mobile application: starting position insert (manually or by GPS) page.

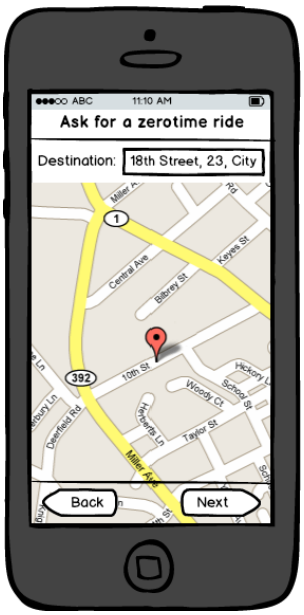

Figure 4.11: Zerotime ride request into mobile application: destination position insert page.

Figure 4.12: Zerotime ride request into mobile application: confirmation page.

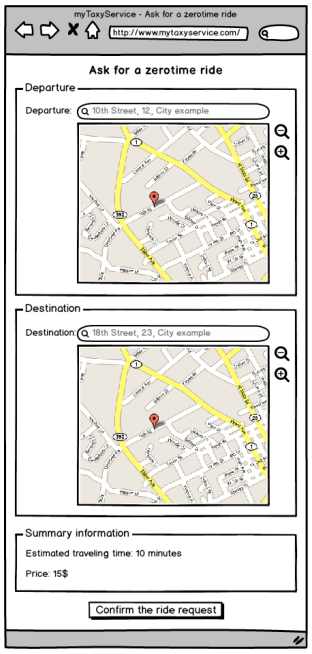

Figure 4.13: Zerotime ride request into website page.

4.5 Future Ride

When a user desires to book a future ride, he has to insert the starting and the destination positions (the first one can be calculated by the GPS), the departure (both the date and the time) and then to confirm the request.
In the MA this sequence of operations is performed with three screens (mock-ups in figures 4.14, 4.15 and 4.16), while into the WS it is used only one screen split into the correspondent sections, shown in figure 4.17.
In both the MA and the WS, the departure inserting is made with a pop-up that uses the apposite operative system functionalities to insert the dates (for example a computer OS can use a calendar for dates and a box for time, while a mobile OS can use a dropdown list or something that slides following the finger's touch).

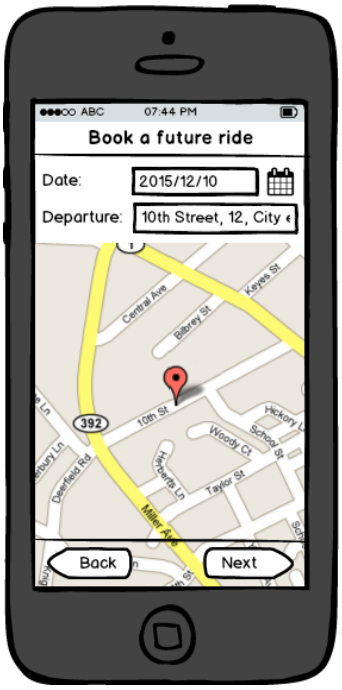

Figure 4.14: Future ride request into mobile application: departure starting position insert (manually or by GPS) page.

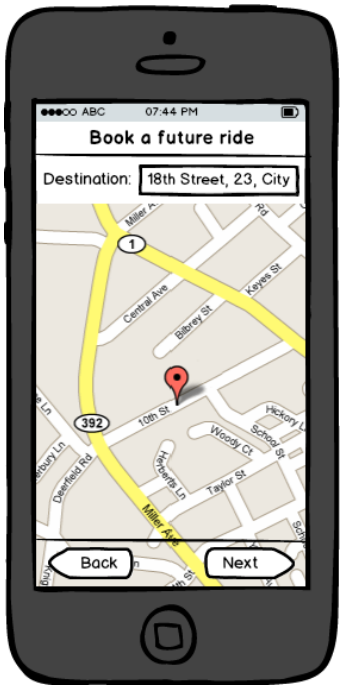

Figure 4.15: Future ride request into mobile application: destination position insert page.

Figure 4.16: Future ride request into mobile application: confirmation page.

Figure 4.17: Future ride request into website page.

4.6 Other User Functionalities

myTaxiService gives other useful functionalities to the users.
In MA there are two screens: in figure 4.18 the users can see all his reservation split into two sections, one for the future reservation (they can be edited) and one, at the bottom, for a list of the past rides; in figure 4.19 the user can edit a specific reservation (either the departure or the positions can be modified) that it was selected in the previous page by a click on the “edit, delete”button.

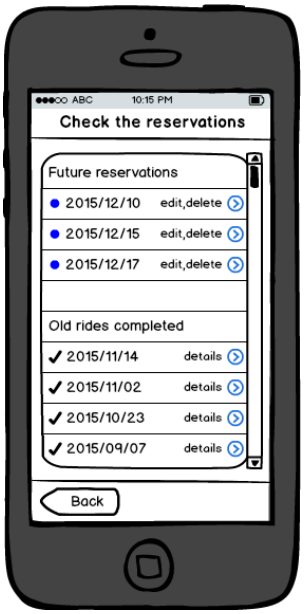

Figure 4.18: The view reservations page into mobile application.

Figure 4.19: The page where a reservation can be modified or be cancelled into mobile application.

In WS the same operations can be performed into the same page, displayed in figure 4.20.

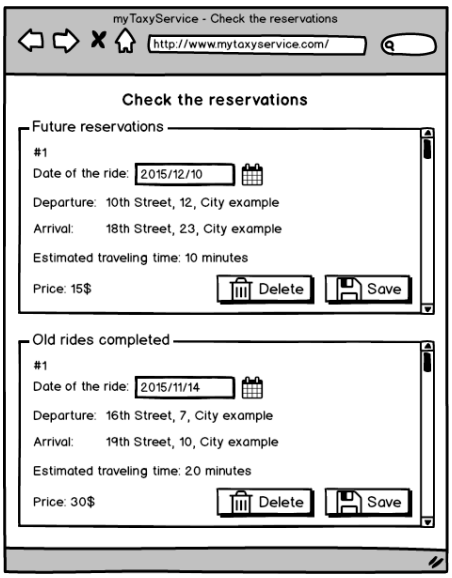

Figure 4.20: The check reservation page into page.

Finally, the user can received some alerts regarding, for instance, the assignment of the ride for which he asked for. The alerts can also be a general information, related to some problems of the service or to a strike.
The corresponding screen for the MA and the WS are shown, respectively, in figures 4.21 and 4.22.

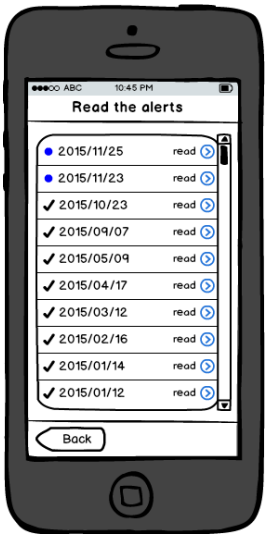

Figure 4.21: The alerts page into mobile application.

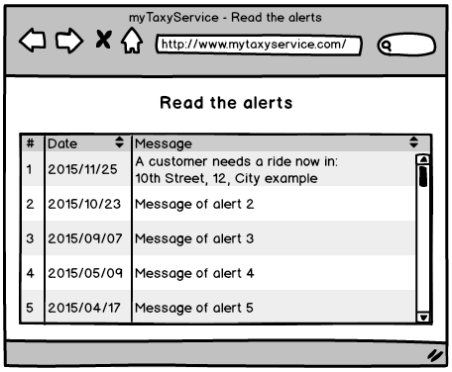

Figure 4.22: The alerts page into website.

Chapter 5

Requirements Traceability

This chapter explains how the functional requirements from the Requirements Analysis And Specification Document (RASD) have been designed and described in this Design Document (DD).
Every section of this chapter refers to a functional requirement.

5.1 Registration

The section 3.2.3 of the RASD describes the registration functionality. A visitor can register himself to myTaxyService only if he's new to the system, i.e. doesn't exist a user with the same tax code.
In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to register both via MA and WS in section 4.1.

5.2 Login

The section 3.2.4 of the RASD describes the login functionality. A visitor can login into myTaxyService only if he's registered. In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to login both via MA and WS in section 4.1.

5.3 Personal Information Management

The section 3.2.5 of the RASD describes the personal information management functionality. A logged user can modify some of his personal information: the e-mail, the password and the city of residence.
In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to manage personal information both via MA and WS in section 4.2.

5.4 Ask for a Zerotime Ride

The sections 3.2.6 and 3.2.7 of the RASD describe the functionality of asking for a zerotime ride both via MA and via WS. A logged user can ask for a zerotime ride.
In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to ask for a zerotime ride both via MA and WS in section 4.4.

5.5 Book a Future Ride

The section 3.2.8 of the RASD describes the functionality of booking a future ride both via MA and via WS. A logged user can book a future ride.
In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to book a future ride both via MA and WS in section 4.5.

5.6 Accept or Deny a Ride

The section 3.2.6 of the RASD also describes within the functionality of the Driver to accept or deny a ride, via MA. A logged driver can accept or deny a request for a ride when he receives it.

In the DD we have satisfied this functional requirement and we have represented the user interface to accept or deny a ride via MA in section 4.3.

5.7 Start Waiting Time

The section 3.2.9 of the RASD describes the functionality of starting the waiting time via MA. A logged driver notifies the system that he's available in an area and waiting for a ride and he is added to the queue of the area.
In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to start waiting time via MA in section 4.3.

5.8 Work Shifts Management

The section 3.2.10 of the RASD describes the functionality of management of the work shifts via MA. A logged driver can add or delete a work shift in the week days.
In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to manage the work shifts via MA in section 4.3.

5.9 Check the Reservations

The section 3.2.11 of the RASD describes the functionality of checking the reservations (both zerotime and future rides) both via MA and via WS. A logged user can view the his reservations, modify the date or cancel them.
In the DD we have satisfied this functional requirement, we have described the sequence of actions in the Sequence Diagram of section 2.5.2 and represented the user interface to check the reservations both via MA and WS in section 4.6.

5.10 Read the Alerts

The section 3.2.12 of the RASD describes the functionality of reading the received alerts both via MA and via WS. A logged user can only view his alerts, he cannot edit them.
In the DD we have satisfied this functional requirement and we have represented the user interface to read the alerts both via MA and WS in section 4.6.

Chapter 6

Other Info

This chapter contains information about the used tools and the hours of work by the members of the working group.

6.1 Tools

In this first requirements study phase the following tools were used:

- LaTeX and TexStudio editor
- starUML
- MySQL Workbench 6.3.5
- Balsamiq Mockups 3

6.2 Working hours

Date	Costanzo's hours	Disabato's hours
2015/11/16	1h	1.30h
2015/11/17	2h	2h
2015/11/18	1h	1h
2015/11/20	2.30h	2.30h
2015/11/21	1h	2h
2015/11/22	1.30h	-
2015/11/23	1h	-
2015/11/24	4h	5h
2015/11/25	-	1.30h
2015/11/26	2.30h	2.30h
2015/11/27	-	2h
2015/11/28	4h	2h
2015/11/29	-	1.30h
2015/11/30	-	1h
2015/12/01	-	1.30h
2015/12/02	2.30h	-
2015/12/03	1h	-
2015/12/04	2h	1h
Total DD	26h	26h
Global	55h	55h