myTaxiService

Design Document

Monica Magoni 854091

Alberto Cibari 852689

December 4, 2015

Contents

1 Introduction

1.1 Purpose

This document represents the Design Document of our project. Its aim is to describe the components of our system and how they will interact, and explain the choices we made about the architectural styles and pattern.
Moreover, this document will be provided with Sequence Diagrams in order to show how the entire system will work dynamically.
Further, this document will give an idea of how the system will look like, giving an overview of user interfaces in a more specific way than in the RASD.

1.2 Scope

The scope of this document is to focus the attention on the architectural structure that our system will have, the hardware and software components that it will be composed of and to offer a perspective of the users interfaces of our applications.
In particular, all of these decisions are based on the idea of satisfying in the best possible way the requirements, functional and non-functional, already presented in our RASD.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

We rewrite some of the definitions that were given in the RASD document and that will be widely used also in this document.

- PASSENGER: for passenger we mean a person, already registered in the system, who has requested or reserved a taxi either through the web or the mobile applications.

- USER: a person who is already registered in the system that can use the application to request or reserve a taxi.
- GUEST: a person who is not registered in the system.
- REQUEST: message sent by the user's application to the system in order to require a taxi or make a reservation.
- CALL: a task received by the taxi drivers after a user made a request.
- TAXI ZONE: for taxi zone we mean a zone that is approximately of 2km2. Each taxi zone has a queue of taxi associated.
- QUEUE: for queue we mean the list of taxi available at a specific moment in a certain zone.
- NOTIFICATION: for notification we mean a message that could be send by the system to the user (in order to notify the taxi identifier that will satisfy his request and the waiting time) or by the system to a specific taxi driver (in order to notify that a user has a request or a reservation to satisfy).
- RESERVATION: the action of the user to book a taxi followed by a request to the system.
- TAXI SHARING: the service that allows passengers to share a taxi.

1.3.2 Acronyms

- DBMS: Database Management System.
- DD: Design Document.
- DMZ: demilitarized zone.
- GPS: Global Positioning System.
- JPA: Java Persistance API.

- MVC: Model-view-controller.
- RASD: Requirements Analysis and Specification Document.
- REST: Representational state transfer.
- SD: Sequence Diagram.
- UX: User Experience.
- XML: eXtensible Markup Language.

1.4 Reference Documents

The starting points of this document are:

- AA 2015-2016 Software Engineering 2, Assignment 1 and 2.
- AA 2015-2016 Software Engineering 2, Project goal, schedule and rules.
- Structure of the Design Document.
- Requirement Analysis and Specification Document (our previous deliverable).

1.5 Document Structure

Our DD document is structured as follows:

- Introduction: in this section, we want to explain the purpose of this document, give some definitions and acronyms and list the documents we referred to.
- Architectural Design: in this section, we focus our attention on the components of our system and how they are deployed in hardware. First, we give an overview of the main components and their interaction; then, we explain in details how they are interconnected, using a Component Diagram. In the end, we explain how components are

physically organized, how the system runs dynamically and what are the architectural styles and patterns we chose for our system.

- Algorithm Design: in this section, we propose algorithms in order to find the best solution of the main issues of our system.
- User Interface Design: using the UX Diagram and referring to the mockups presented in the RASD, we give an idea of how the users interfaces will look like.
- Requirement Traceability: this section has the aim to show the relationships between the DD document and the RASD.
- References: here we list all the documents we referred to during the writing of the DD document.

2 Architectural Design

2.1 Overview

By considering the traditional client-server architectural pattern, we have decided to use for our system one of the extended versions of that approach: a distributed three-tier application.
Further, we have decided to start from the model provided by the Java EE standard, which offers the possibility to reduce the development time (by using conventions and annotations) and to abstract from the application complexity by organizing the architecture into components and containers.

2.2 High level components and their interaction

According to the architecture we have chosen, our system will be composed of these high-level components:

- Client-tier components: run on the client machine; these components will be represented as a unique main component in our Component Diagram;
- Web-tier components: run on the Java EE server; they are organized in Servlets and JSP pages and they will be used to manage the interaction with the web application;
- Business-tier components: run on the Java EE server; these components manage the internal logic of the system and they will be analyzed in the Component Diagram;
- Enterprise Information System (EIS)-tier components: run on the database server and handle enterprise infrastructure systems, database systems and legacy systems.

The following diagram shows the architectural model of our system, based on the Java EE approach. It also shows how high-level components interact

one with the others.

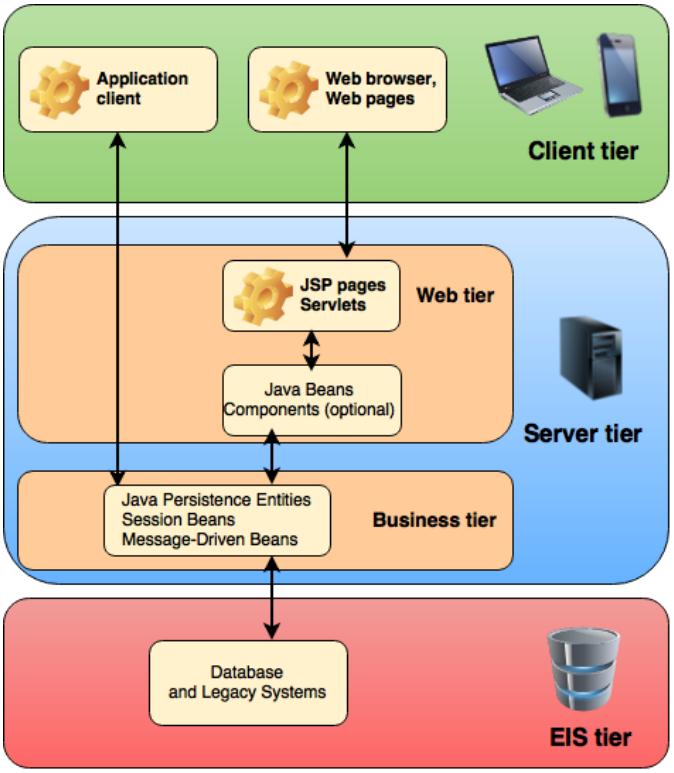

Figure 1: Architectural model of our system

As the figure 1 shows, the business tier is composed of session beans, JPA and message-driven beans. These components will be useful to manage our business logic:

- Session beans: in our system, we will choose stateful beans in order to track the session of a specific user; in this way, a client will be associated to a specific bean during all the session's duration and the bean will be destroyed after the termination of the session.
- JPA entities: these entities are associated to Java classes and each of them is mapped to a table in the database; thanks to JPA implementations, it is possible to materialize those entities in the database at runtime.

- Message-driven beans: these beans allow our applications to process messages asynchronously; for example, when the client makes a new request, a message is sent to the CallManager component; this process is asynchronous and so the client application can continue doing its tasks, it does not have to wait for the messages to be received and processed. The good thing is that the EJB container will manage all this process.

In order to explain how logical layers are distributed among the tiers, we have to distinguish into two case: the mobile application and the web one.

2.2.1 Mobile application

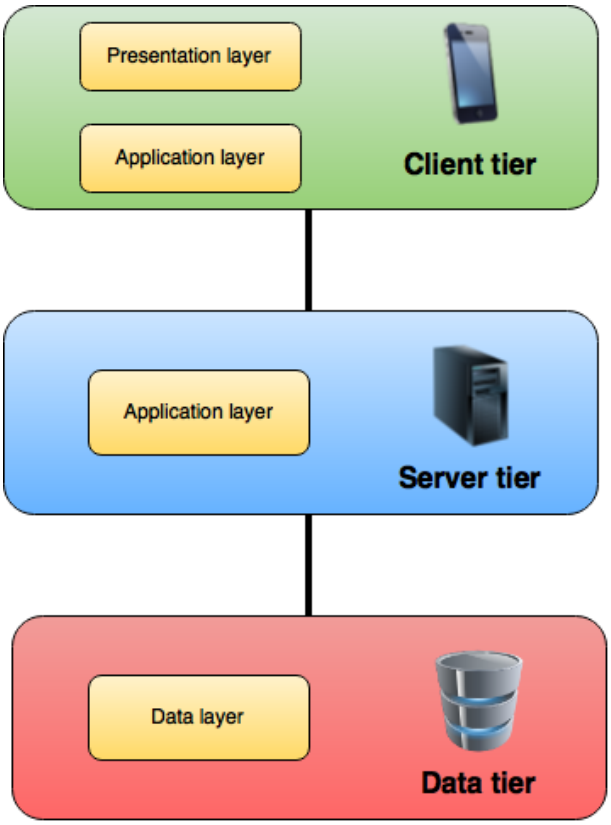

Figure 2: Three-tier architecture for the mobile application

- Client tier: it will implement the presentation layer but also the application layer: when the client's device will receive from the server JSON data, it will have to parse them and then, according to the business logic, create the view that will be visualized on the device. These clients will be not thin because they need to implement part of the application layer, in addition to the presentation layer.

- Server tier: it will implement the business logic; this means that this server is in charge to manipulate the data and perform detailed processing. For example, this level will be responsible for managing all the requests from users and changing taxi drivers' status dynamically.

- Data tier: it refers only to the data layer, where all the data of our system will be stored. For example, it will store all the details about requests, taxi drivers' information and so on.

2.2.2 Web application

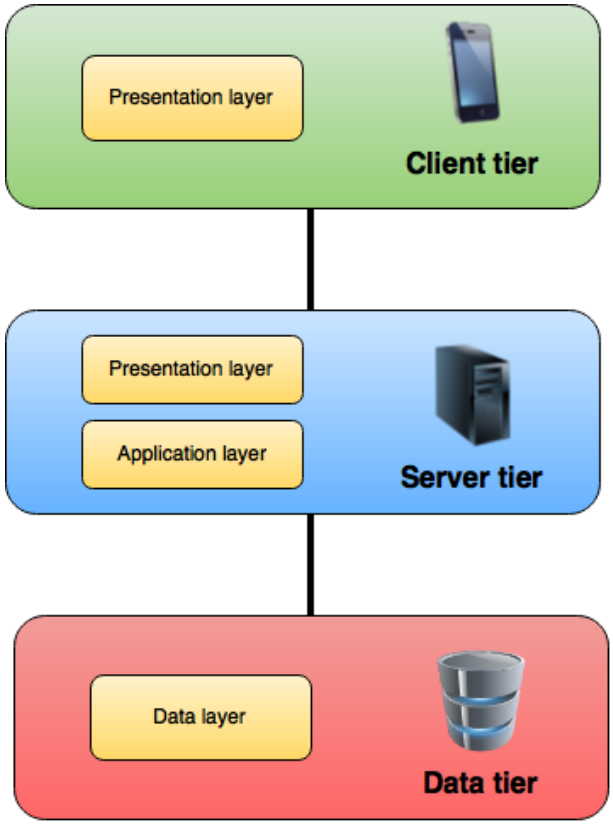

Figure 3: Three-tier architecture for the web application

- Client tier: it will implement only the presentation layer; in fact, the client's device will receive from the server HTML pages and its main function is to translate those pages so that the user can understand the result.
  This tier will also have to perform some controls on the users' input (for example it will check if the date of a reservation exists) but, due to the fact that these actions are minimal, we do not consider them as an implementations of the application layer. So, web clients will be thin clients because they will not need to perform complexes business rules.

- Server tier: it will implement the business logic and the presentation

layer; this means that this server is in charge to create dynamic web pages in HTML and manipulate the data.

- Data tier: it refers only to the data layer, where all the data of our system will be stored. For example, it will store all the details about requests, taxi drivers' information and so on.

2.3 Component view

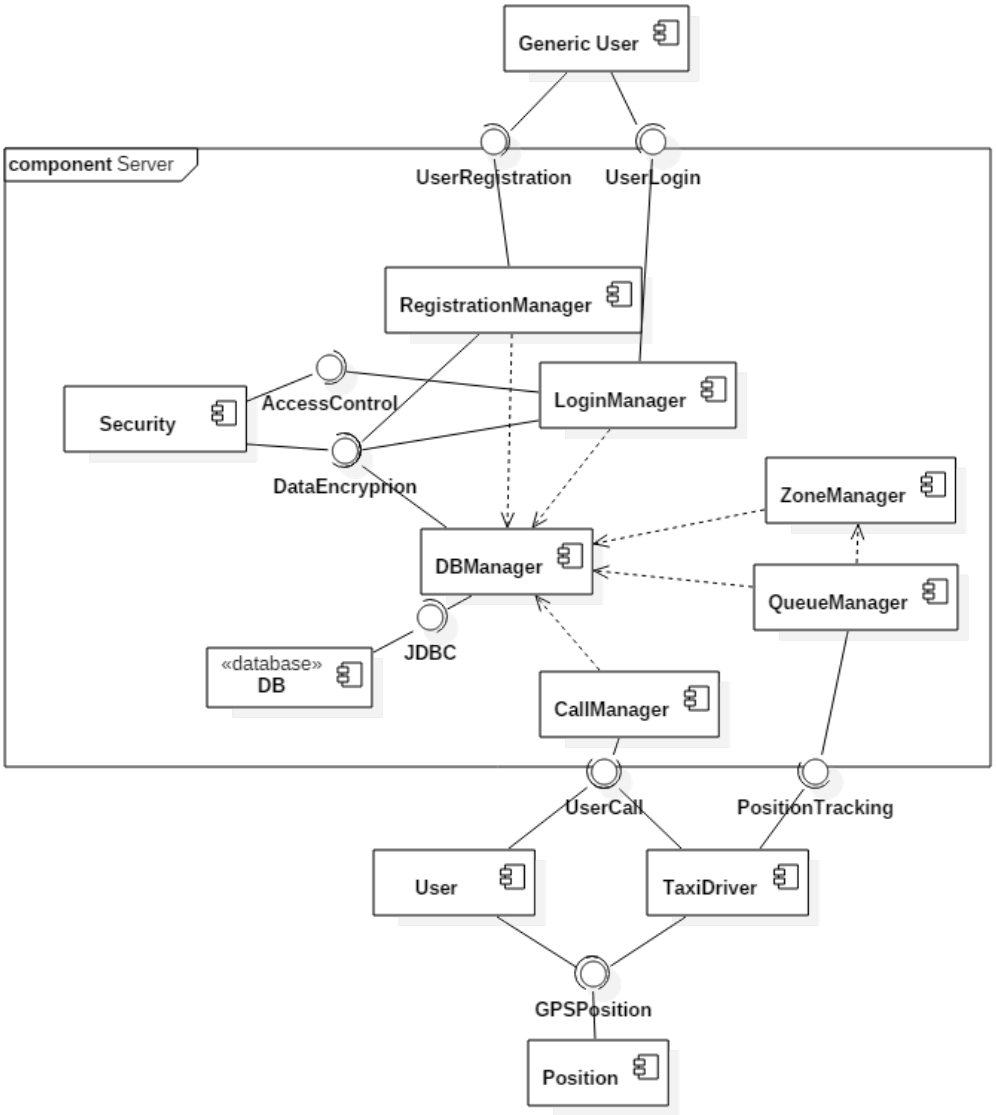

Figure 4: Component diagram of our system

Registration Manager

This component handles the registration of a new user into the system, using some interfaces offered by the Security component.

Login Manager

This component handles the login of a user into the system, using some interfaces offered by the Security component.

Database Manager

This component manages the interaction with the database providing all the information asked by all the other components.

Zone Manager

This component coordinates all the zones the city is divided in.

- It create the queues of every zone.
- When a taxi driver leaves a zone, the taxi driver must be removed from the previous queue in which he was, so the ZoneManager notifies the QueueManager, telling him the new zone of the taxi and the associated queue.

Queue Manager

This component manages the taxi drivers in a certain zone.

- Add/delete a taxi driver from the queue
- Change the position of taxi drivers in the queues.
- When a request is incoming it searches for the right taxi driver to send the request to.

Call Manager

This component manages all the incoming/waiting calls for a taxi.

- Retrieves all the information regarding the call.
- Updates the information of the calls when their status is changed.

Position

The scope of this component is to get the GPS positions of the devices that use it. Those components after using it send the data to the various components in the system. In particular The Taxi Driver component uses the data got by the Position component to track his position sending the information to the Queue Manager.

Security

This component offers some interfaces to grant security to the whole system. For example it prevent a user to have multiple login session on different devices at the same time.

2.4 Deployment view

In order to offer a deployment view of our system, we have decided to use the Deployment Diagram. This choice is due to the fact that this diagram is strictly related to the Component Diagram: indeed, it shows how software components, previously described in the Component Diagram, are deployed in hardware.
Referring to figure 5, we have to specify that the web browsers and the Operating Systems supported by our applications, in addiction to the database server and the web and application server we have chosen, have been already listed in the RASD.[1]

[1] see the 'Constraints' subsection in the RASD

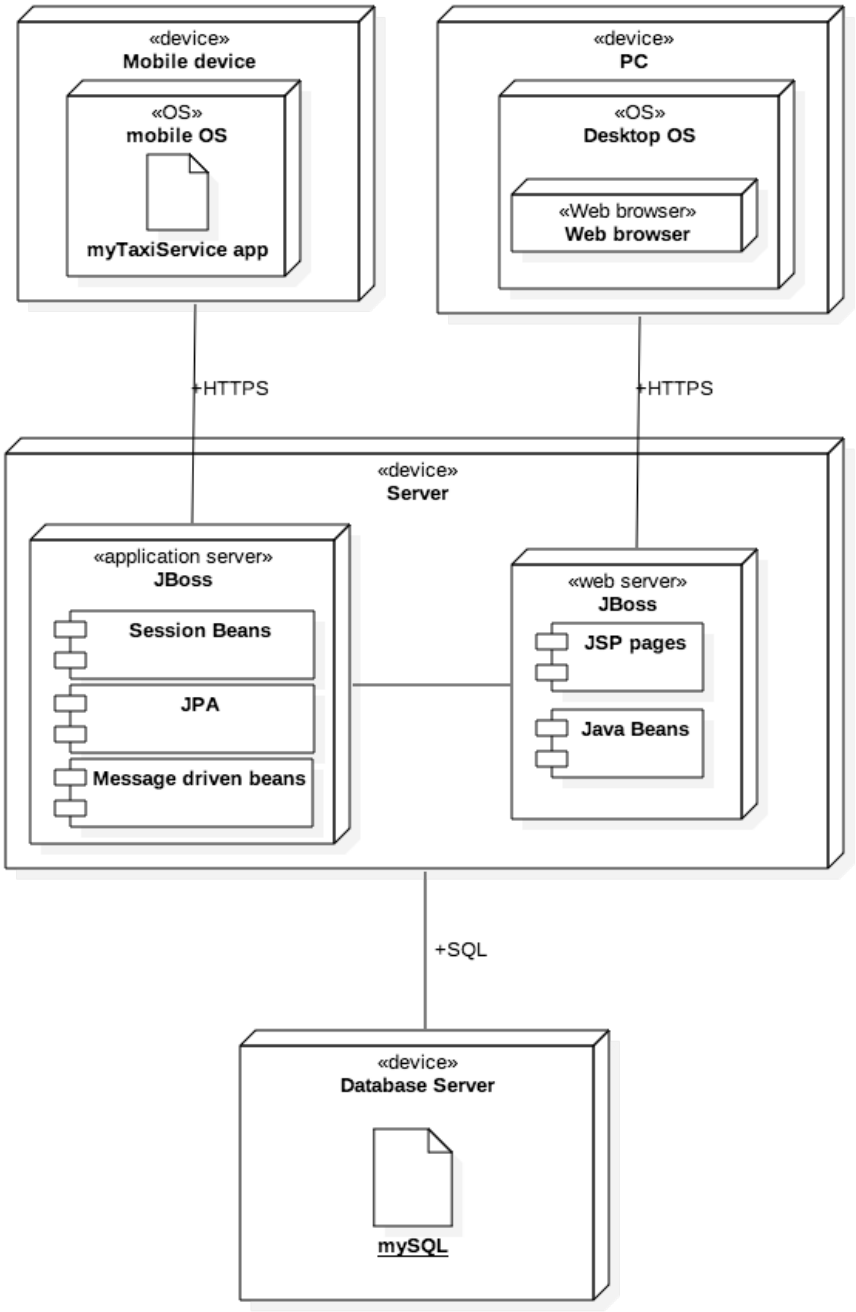

Figure 5: Deployment Diagram

We also give a brief description of the hardware infrastructure. We decided for a dual host configuration: this means that the web server and the appli-

cation server will be placed on the same machine, while the database will be located on another host. This choice has been preferred to the single host configuration because we want to assure that the data stored in the database are well protected and so, by separating into two hosts, we can put a firewall between them in order to guarantee an higher security.
Our system will have another firewall in order to filter packets coming from the external network and so it will be placed before the DMZ to protect the internal network.

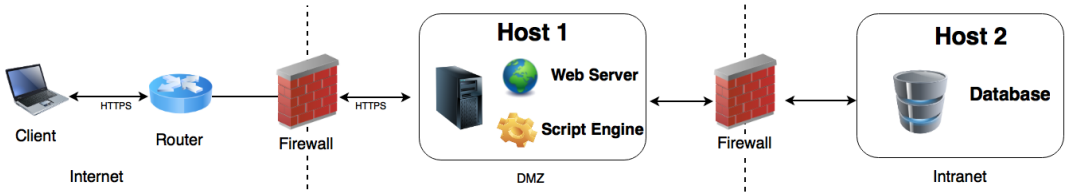

Figure 6: Hardware architecture

2.5 Component interfaces

This section provides a description of the interfaces of the system that allow the interaction between the components (shown in figure 4)

UserRegistration

This interface is invoked every time a generic user tries to register into the system:

- A user submits his info to the system.
- A user clicks on the link provided in the email to complete the registration.

UserLogin

This interface is used every time a user tries to login into the system from both the mobile and the web application.

UserCall

This interface is used when a logged user interacts with the taxi requests.

- He creates a request.
- He accepts/rejects a request, if he is a taxi driver.
- System sends notifications to the users.
- A new call is incoming.

AccessControl

This interface provides the functionalities to grant the unique access to the user in the login phase.

Position Tracking

This interface provides the functionalities to track the position of the taxi drivers receiving the GPS position and other info. It is implemented in the Queue Manager and used by the Taxi Driver component.

2.6 Runtime view

This section aims to give a view of the system working dynamically. So, we have redone the Sequence Diagrams described in the RASD in a more detailed way, considering also the components presented in the Component Diagram.

Login

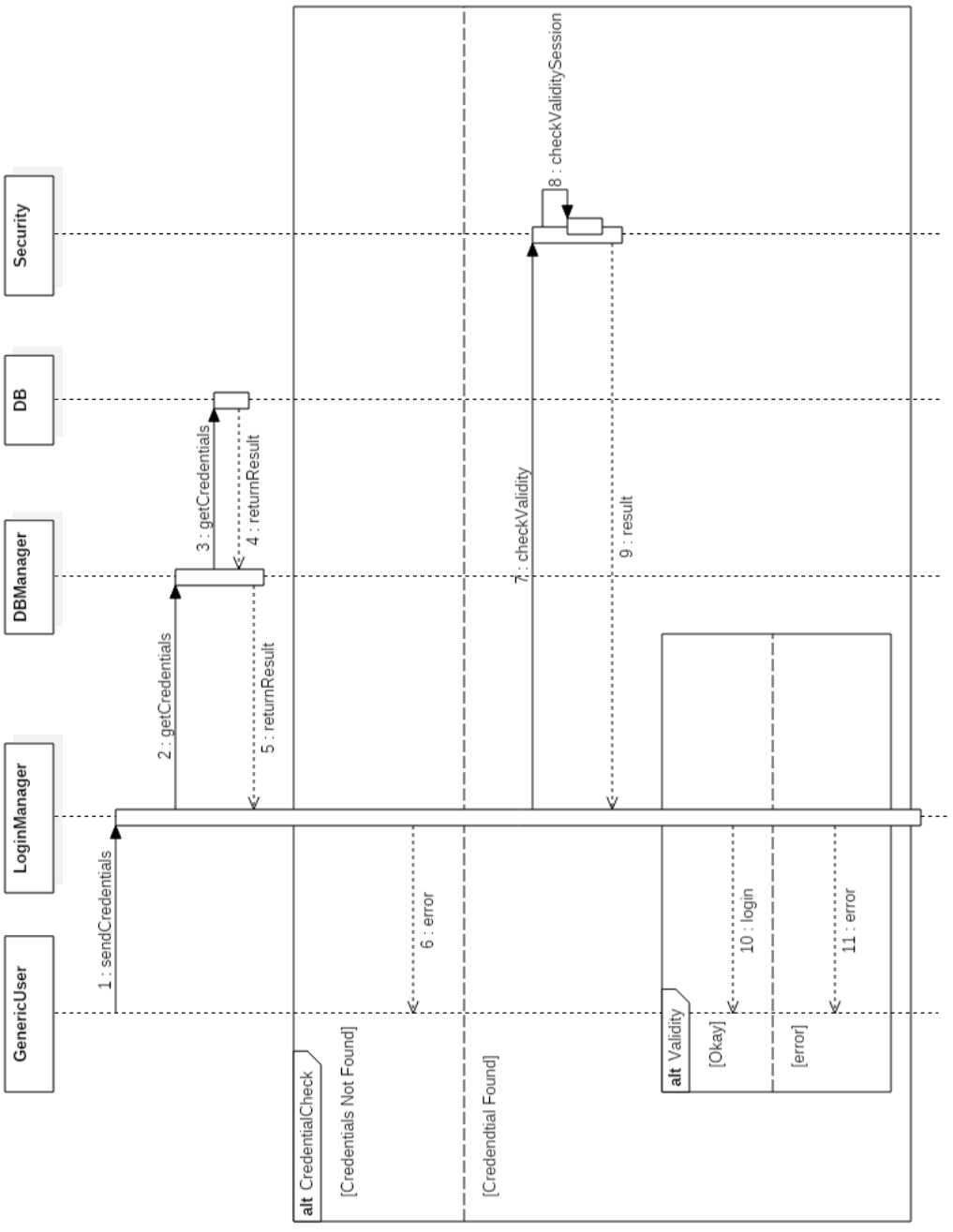

Figure 7: Login SD

User Registration

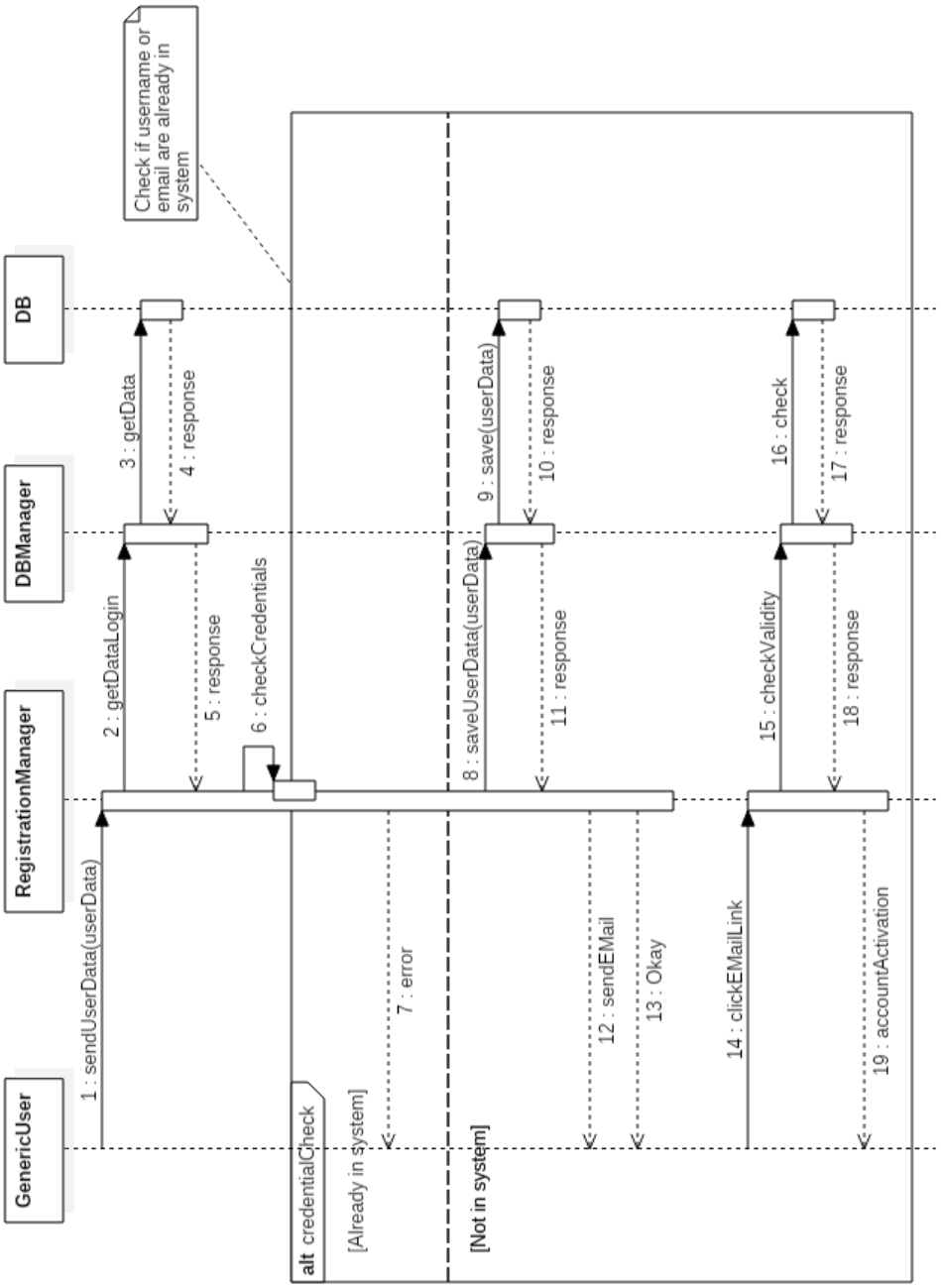

Figure 8: User Registration SD

Taxi Request

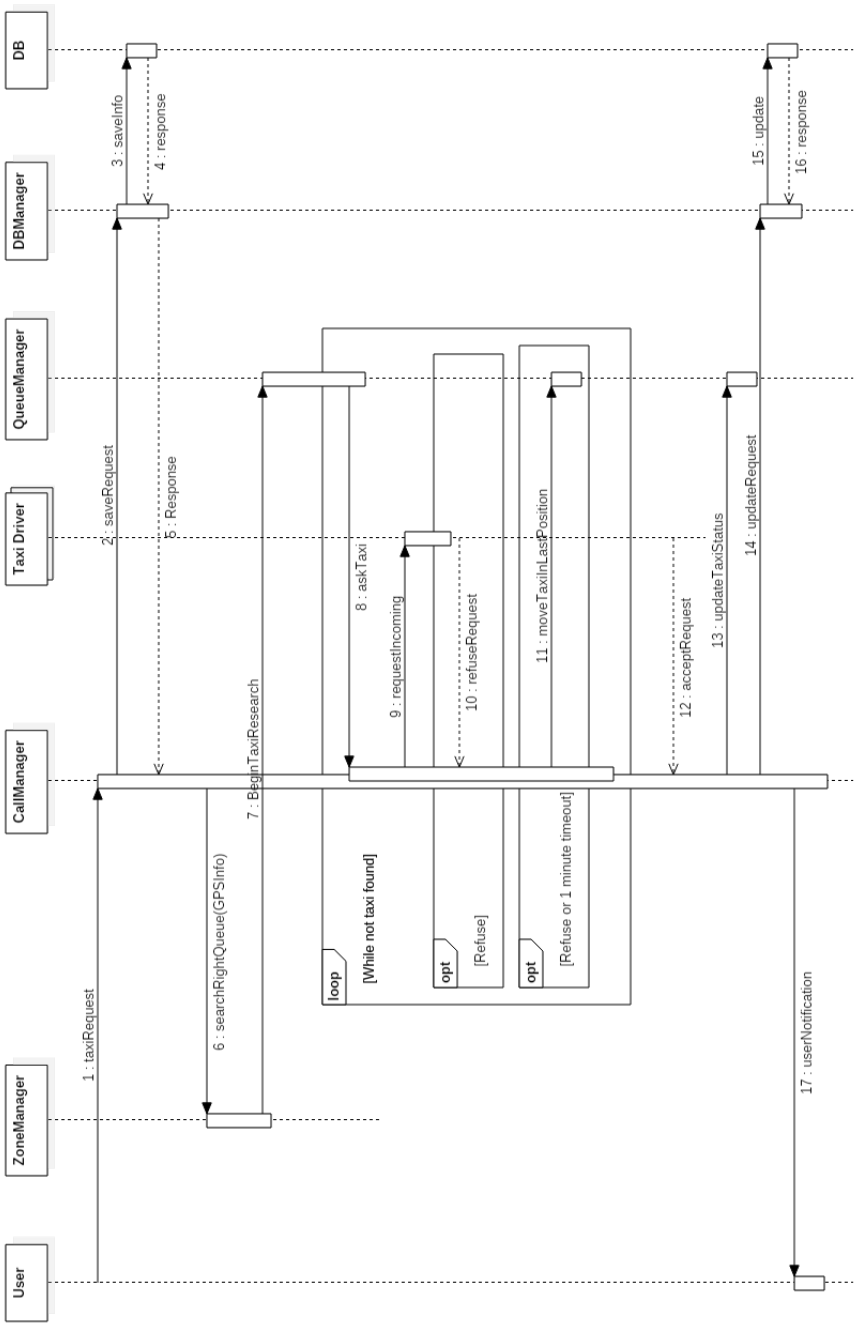

Figure 9: Taxi request SD

2.7 Selected architectural styles and patterns

2.7.1 MVC pattern

Our system will be strongly based on the MVC architectural pattern; this choice is due to the fact that this pattern will help us reducing the complexity of the structure of the system, by distributing all the functionalities between three interconnected parts (model, view and controller).

The following schema explains this pattern:

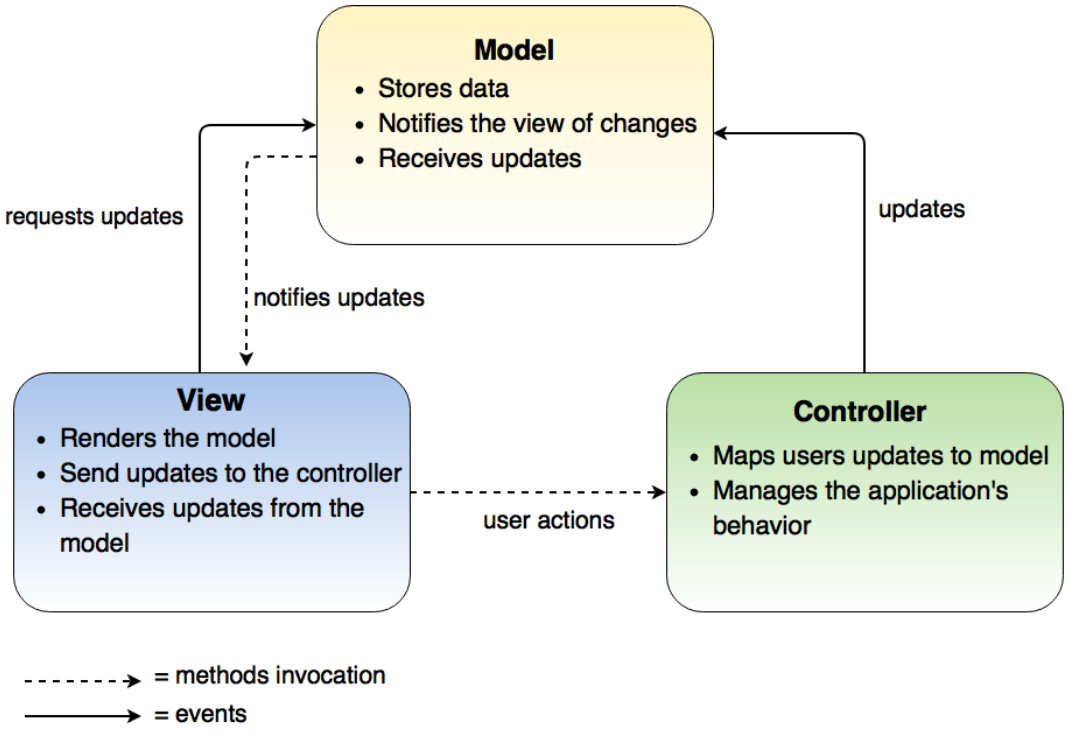

Figure 10: MVC pattern

2.7.2 State pattern

This pattern will be used to manage the behaviour of the taxi drivers; indeed, taxi drivers will be provided with a status (that can be 'AVAILABLE', 'NO-

TAVAILABLE' or 'OFFLINE' as discussed in the RASD[2]) and, according to their state, they will be able to perform only some specific actions. For example, if the status of a taxi driver is 'NOTAVAILABLE', he will not be inserted in any queue and so he will be not allowed to accept any request.
Further, this pattern will be used also for request; each request has a status (that can be 'WAITING', 'RIDING', 'COMPLETED' as discussed in the RASD[2]) and also in this case, according to the status of the request, only some specific kind of actions can be performed. For instance, if the status of a request is 'RIDING', it means that there must be a taxi driver associated to that request, while if a request is 'WAITING' there are no taxi drivers associated to this request.

2.7.3 Client-server model

In our project, we have referred to this model two times: for the communication between the clients and the web server and the one between the application server and the database.

2.7.4 REST

The mobile applications on the front-end of the system will communicate with the application server thanks to a RESTful interface that is implemented on the application server. So, the mobile device will send requests using the HTTPS protocol and then it will receive from the application server JSON data that it will parse to extract information.
We have preferred REST to SOAP because a RESTful interface is easier to implement, while SOAP requires a more rigid and complex structure. Further, SOAP returns XML data that are more difficult to parse than JSON messages, which on the contrary are concise and not so verbose. This last consideration was a very important point for our choice: we want to avoid that our mobile application will be too heavy and so we have preferred the REST+JSON structure.

[2]see the Class Diagram in the 'Specific Requirements' section

2.7.5 Web services

Our system is also provided with a web service that is offered in the mobile application of taxi drivers. Once a taxi driver have accepted a request, the application will give geolocation information about the journey he has to take care of. Moreover, this service will provide the average time a passenger has to wait for the taxi arrival and also the average time the taxi will take to reach the destination.
In order to do so, we have decided to use an external service, Google Maps, and our choice is due to different reasons: first of all, we have decided not to implement this service manually because it would be so difficult and secondly, Google Maps is a very powerful and well-known software, that almost everybody is able to use.
Google Maps API provide interfaces to Google Maps services and allow to request data that can be used in our application; in our case, this means that our mobile application will send HTTP requests to a specific URL (according to the specific service we need), passing some parameters; then, the service will return JSON data that will be parsed by the application in order to get some geolocation data or distance values.

3 Algorithm Design

3.1 Taxi queue management

In this section are described the procedures the system uses to manage the taxi queues. The taxi are distributes in various zone the city is divided in. Each zone has a taxi queue in which available taxi are. The management of taxi is executed by the QueueManager and ZoneManager components.
The algorithm is executed when one of these events occur:

1. taxi driver changes his status
2. taxi driver changes zone
3. taxi driver accepts the call
4. taxi driver refuses the call
5. taxi driver sets a call as 'COMPLETED'

For each event the system behaves in a different way.
The system variables used by this algorithm are:

- Q_i: the queue list for the zone i
- T: the list of taxi drivers that are not in a queue, but are logged into the system.

Whenever the system receives an event e with the the algorithm is executed

Event: taxi driver changes his status

- from 'OFFLINE' or 'NOTAVAILABLE' to 'AVAILABLE': the taxi driver reference is put into the queue Q_i referred to the Zone i. To find the right zone and queue the algorithm uses the parameters passed to the event e, which are: the reference of the taxi driver, his location and the type of event.

- from 'AVAILABLE' or 'NOTAVAILABLE' to 'OFFLINE': the taxi driver is removed from the Q_i or the T queues.
- from 'AVAILABLE' to 'NOTAVAILABLE': the taxi driver is removed from the queue Q_i and put into the T queue.

Event: taxi driver changes zone

This event is triggered when the ZoneManager realizes that the taxi driver has changed zone and notifies the QueueManager. When this event occurs the taxi driver is removed from the Q_i queue list where the taxi is currently in and he is put into the new Q_k queue referred to the new zone he entered.

Event: taxi driver accepts the call

When a taxi driver accepts a request this event is fired. The algorithm removes the taxi driver from the Q_i zone and put it into the T queue

Event: taxi driver refuses the call

When a taxi driver refuses a request this event is fired. The algorithm removes the taxi driver from the first position of the Q_i queue and he is put in the last position of the same queue.

Event: taxi driver sets a call as 'COMPLETED'

When a taxi driver sets a call as 'COMPLETED' he becomes available, so he is put at the end of the Q_i queue.

3.2 Waiting time

In this section are described the procedures the system uses to tell the users the remaining time before a taxi arrives or the remaining time before the arrive to the destination of a ride. The calculations are made by the Google Maps API.

These APIs are used making HTTPS requests to *http://maps.googleapis.com/maps/api/directions/xml* with the following parameters:

- origin: the current location of the user
- destination: the arriving point
- departure_time: the current time
- traffic_model: 'best_guess'. This is the best estimate of travel time given what is known about both historical traffic conditions and live traffic.
- mode: 'driving'
- key: API_KEY. This is the application's API key.

Whenever a user is waiting for the arrive of a taxi or he is waiting to reach the destination, he can view the remaining time for the arrive.

Case 1: taxi waiting

In this case the user is waiting for the arrive of a taxi driver. After he received the notification for his request acceptance, he can view the remaining time.

- The device of the user makes a request to the CallManager telling it to calculate the remaining time.
- The CallManager makes a request to the Google Maps services requesting the data in XML format.
- after receiving the XML response the CallManager send it to the client device that displays it.
- Every minute the device redoes the request for displaying a new waiting time.

Case 2: arrive to destination time

In this case the user wants to know when he will arrive to the destination during a taxi ride.

- The device, when the user wish to know, makes a request to the CallManager telling it to calculate the remaining time.
- The CallManager makes a request to the Google Maps services requesting the data in XML format.
- after receiving the XML response the CallManager send it to the client device that displays it in the request details page.

4 User Interface Design

To give an overview of the user interfaces, we have implemented a UX diagram and we also refer to the mockups already shown in the RASD. [3]
In order to be more precise, we have added a mockup that shows the use of the geolocation service provided by Google Maps.

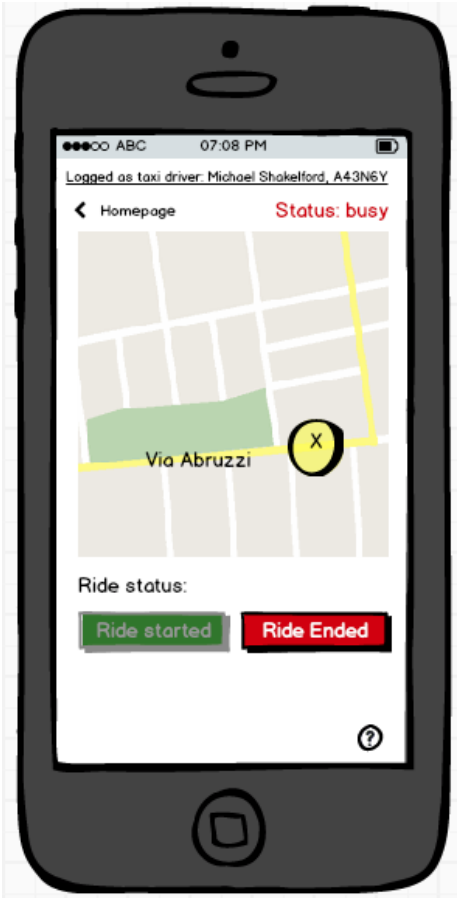

Figure 11: Geolocation service

[3] see the 'System functions' in the 'Specific Requirements' section

UX Diagram

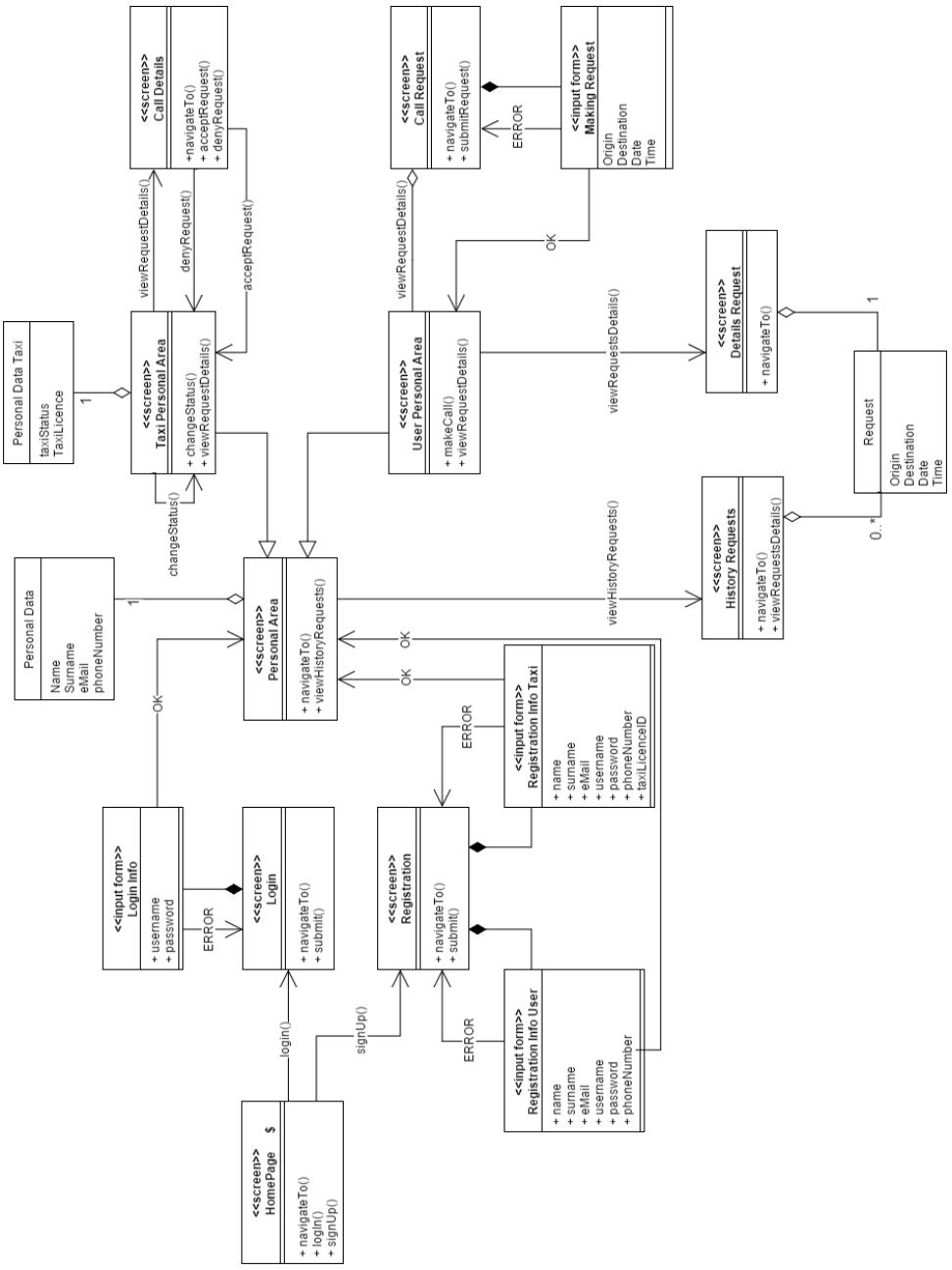

Figure 12: Ux Diagram

5 Requirements Traceability

This section has the aim to map the DD document with the RASD.
In particular, our intent is to provide a specific mapping between the functional requirements identified in the RASD to the components described in the DD document. This mapping can be easily verified in the 'Runtime View' section, by analyzing the Sequence Diagrams.

Functional requirements	**Components**
Guest registration	RegistrationManager, GenericUser, DB Manager, DB
User/taxi login	LoginManager, User/TaxiDriver, DB Manager, DB, Security
User/taxi home page	AccountManager, User/TaxiDriver, DB Manager, DB, Security
Activation/deactivation of taxi service	TaxiDriver, Position, ZoneManager, QueueManager, DB Manager, DB
Request making	CallManager, User, TaxiDriver, QueueManager, ZoneManager, Position, DB Manager, DB
Reservation making	CallManager, User, TaxiDriver, QueueManager, ZoneManager, Position, DB Manager, DB
Taxi driver notification handling	TaxiDriver, CallManager, ZoneManager, QueueManager, Position, DB Manager, DB
Zone queue handling	ZoneManager, QueueManager, Position, TaxiDriver, DB Manager, DB

Table 1: Mapping between functional requirements and components

6 References

Our DD document also refers to these documents:

- Google Maps API Web Services: *https://developers.google.com/maps/web-services/overview*
- UML Component Diagram: *http://www.uml-diagrams.org/component-diagrams.html*
- UML Deployment Diagram: *http://www.uml-diagrams.org/deployment-diagrams.html*

Appendix

Tools used

- Balsamiq: to create mockups. [4]
- Draw.io: to create the Component Diagram, the UX diagram and some images. [5]
- GitHub: to share our work. [6]
- ShareLaTeX: to write this document in LaTeX. [7]
- StarUML: to create the Deployment Diagram and the Sequence Diagrams. [8]

Hours of work

- Alberto Cibari: 27 hours
- Monica Magoni: 27 hours

[4] `https://balsamiq.com`
[5] `https://www.draw.io`
[6] `https://github.com`
[7] `https://it.sharelatex.com`
[8] `http://staruml.io`