Design Document

myTaxiService

Version 2.1

Fabio Catania, Matteo Di Napoli, Nicola Di Nardo

February 14, 2016

Contents

1 Introduction **4**
- 1.1 Purpose . . . 4
- 1.2 Scope . . . 4
- 1.3 Definitions, acronyms and abbreviations . . . 4
  - 1.3.1 Definitions . . . 4
  - 1.3.2 Acronyms . . . 5
  - 1.3.3 Abbreviations . . . 6
  - 1.3.4 References . . . 6
- 1.4 Document Overview . . . 6

2 Architectural Design **7**
- 2.1 Overview . . . 7
- 2.2 Selected Architectural Styles and Patterns . . . 7
- 2.3 General Design Description . . . 7
- 2.4 Paas Provided by Google Cloud Platform . . . 8
  - 2.4.1 App Engine . . . 9
  - 2.4.2 Cloud Endpoints . . . 9
  - 2.4.3 Cloud Storage and Datastore . . . 10
- 2.5 General Packages Design . . . 10
- 2.6 Component View . . . 11
  - 2.6.1 Model-Entity Diagram . . . 11
  - 2.6.2 Taxi Driver View Model . . . 13
  - 2.6.3 Standard-Reservation View Model . . . 14
  - 2.6.4 Shared Call View Model . . . 15
- 2.7 Runtime View . . . 16
  - 2.7.1 Reservation Sequence Diagram . . . 16
  - 2.7.2 Taxi Driver Login Sequence Diagram . . . 17
- 2.8 Component Interfaces . . . 18
- 2.9 Database Design . . . 18
  - 2.9.1 Conceptual Design . . . 19
  - 2.9.2 Logical Design . . . 21
- 2.10 Other Design Decisions . . . 23

3 Algorithm Design **23**
- 3.1 lookForTaxi() . . . 23
- 3.2 getTaxi() . . . 23
- 3.3 updateQueue() . . . 23
- 3.4 arrangeRoute() . . . 24
- 3.5 arrangeCost() . . . 24

4 User eXperience Design **25**
- 4.1 Customer Side UX . . . 25
- 4.2 Taxi Driver Side UX . . . 26
- 4.3 User Interface Design . . . 27

5 Requirements Traceability **28**

6 RASD Modifications **28**

7 Appendix **28**
7.1 Software Employed 28
7.2 Working Hours 29

1 Introduction

1.1 Purpose

This Design Document (DD) is a detailed description of the myTaxiService system. It is written by software designers in order to give the software development team overall guidance to the architecture of the software project.

The document explains all the architectural decisions and the trade-offs chosen in the design process and their justification. Each decision is taken in order to achieve the goals of the application set in the previous phase of design. It has been used a top-down approach to have an overall idea of the entire system.

Practically, this supporting material is required to coordinate a large team under a single vision, needs to be a stable reference and outlines all parts of the software and how they should work.

1.2 Scope

The system aims at optimizing the taxi service in London. The software is intended to:

- integrate the current telephonic reservation system by offering the possibility to request a taxi through a web application and a mobile app;
- guarantee a fair management of the taxi queues;
- offer the possibility to share the taxi with other people. The goals of the application are illustrated more in detail in the section 1.5 Goal of the Requirements Analysis and Specifications Document.

1.3 Definitions, acronyms and abbreviations

1.3.1 Definitions

- Cloud Computing: a kind of Internet-based computing, where shared resources,data and information are provided to computers and other devices on-demand.
- Design Pattern: a general reusable solution to a commonly occurring problem within a given context in software design;
- Distributed system: a software system in which components located on networked computers communicate and coordinate their actions by passing messages;
- Software architecture: the high level structures of a software system and the documentation of these structures;
- Software Design: the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints;

- Software Component: a unit of composition with contractually specified interfaces and explicit context dependencies only;
- Software Interface: a shared boundary across which two separate components of a computer system exchange information;
- Software Layer: A group of classes that have the same set of link-time module dependencies to other modules.

1.3.2 Acronyms

- API: Application Programming Interface
- AS:Application Server
- CPU: Central processing unit
- DB: Data Base.
- DBMS: Data Base management system.
- DD: Design Document
- EJB: Enterprise Java Bean
- ER: Entity Relationship
- GPS: Global Position System
- GUI: Graphical User Interface
- HTML: HyperText Markup Language
- ICT: Information and Communication Technology
- IEEE: Institute of Electrical and Electronics Engineers
- JB:Java Bean
- JEE: Java Enterprise Edition
- LD: Logical Design
- MVC: Model-View-Controller
- OS: Operative System
- PaaS: Platform as a Service
- QoS: Quality of Service
- RASD: Requirements Analysis and Specification Document
- REST: Representational State Transfer

1.3.3 Abbreviations

[Gn]: n-goal.

[Rn]: n-functional requirement.

[Dn]: n-domain assumption.

[Mnn]: n-mock up.

1.3.4 References

- IEEE Software Engineering Standards Committee IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specications, October 20, 1998.
- Specication Document: Software Engineering 2 Project, AA 2015-2016 Assignments 1 and 2.pdf
- RASD_definitive.pdf
- Structure of the design document.pdf

1.4 Document Overview

This document is organized taking a leaf from Structure of the design document.pdf. In order to simplify his consultation it is divided in the following sections:

- Section 1: Introduction, it gives a description of entire document and some basic information about how to consult it.
- Section 2: Architectural Design, it provides a careful description of the system architecture and of the software design.
- Section 3: Algorithm Design, it focuses on the definition of the most relevant algorithmic part of your project.
- Section 4: User Experience Design, it includes the choices made about the usability of the application from the taxi driver's and from the customer's point of view. It is specified how the graphical user interface screens should look like and all the possible movements from a screen to another.
- Section 5: Requirements Traceability, it explains how the requirements defined in the RASD map into the design elements defined in this document.
- Section 6: RASD modifications: it includes all the rivisitations of the Requirements Analysis and Specification Document
- Section 7: Appendix, this part contains some information about software and hours employed to redact the document.

2 Architectural Design

2.1 Overview

This section provides a detailed description of the design of the system. It includes the explanation of the choices made about the architecture and their justification and effects, the description of the sub-components, their interaction and a note about the storage of data.

As already said, the approach used by passing this document is top-down. A top-down approach is essentially the breaking down of the system to gain insight into its compositional sub-systems in a reverse engineering fashion. In the top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Later each subsystem will be refined in greater detail using a bottom-up approach.

2.2 Selected Architectural Styles and Patterns

myTaxiService is thought as a distributed system. As a consequence, our choice is to use a client-server style to design it. The clientserver model of computing partitions tasks and workloads between the provider of a resource or service, called server, and service requesters, called clients.

We opted for adopting a server hosted on a cloud platform, both as software engine and data storage. Platform as a Service (PaaS) allows to develop, run, and manage the web application without the complexity of building and maintaining the infrastructure typically associated with developing and launching a mobile or web app.

In order to obtain a better organization of the code and therefore to have an easier maintainability we also chose to apply the Model-View-Controller architectural pattern. It divides the software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to the user.

2.3 General Design Description

myTaxiService application is designed as a 3-tier-architecture, that is a clientserver architecture in which presentation, application processing and data management functions are physically separated. This choice is very common among web applications because this kind of model lend flexibility, robustness and scalability to the application. Each tier is described as follows:

- Client Tier: it is the topmost level of the application and aims at the communication of information to the user and to the translation of the user's action in system's command. In simple terms it is represented by the web pages and by the GUI of the mobile application;

- Business Logic Tier: it processes the data according to the requests by the client tier using the data from the persistence tier;

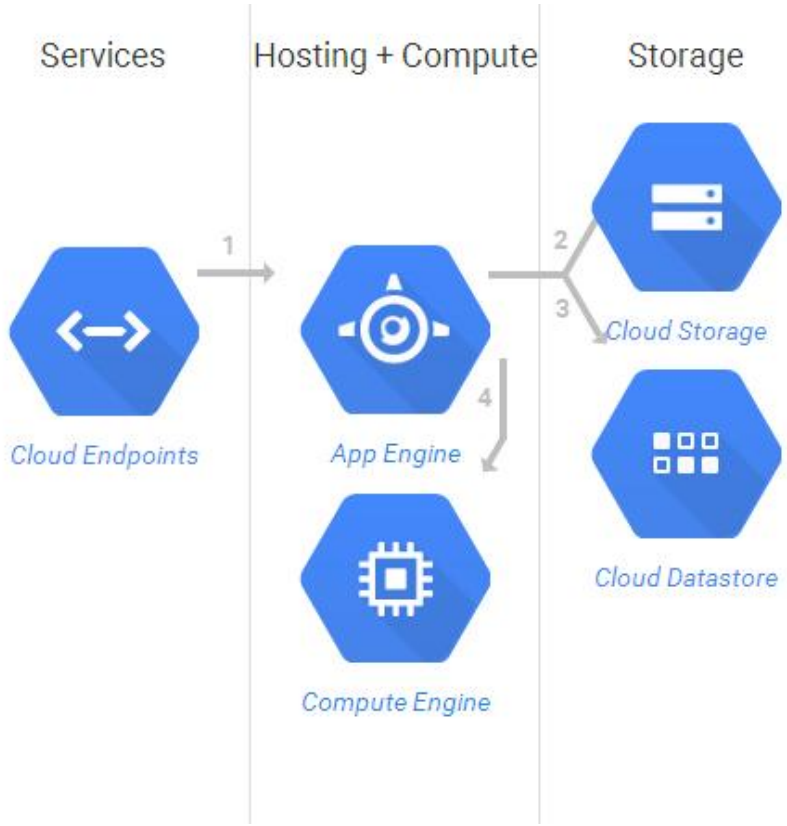
- Persistence tier: it is in charge of storing and retrieving information from the database. The data access layer provides an API to the application tier that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms.

2.4 Paas Provided by Google Cloud Platform

We decided to develop our application relying on Googles Paas to rapidly develop and deploy it without worrying about system administration: we let Google manage accessibility, database and storage servers.

The services provided on the platform automatically scale up to handle the most demanding workloads and scale down when traffic subsides: this is a strong advantage for an application like ours that will deal with very variable amounts of traffic during the day (we can imagine the traffic to intensify during rush-ours and then dramatically re-size in other periods, like night-time, for example).

In addition, Googles compute infrastructure can provide the application consistent CPU, memory and disk performance, allowing it to run also heavy algorithms for the most demanding tasks (like computing sharing associations and meeting point).



2.4.1 App Engine

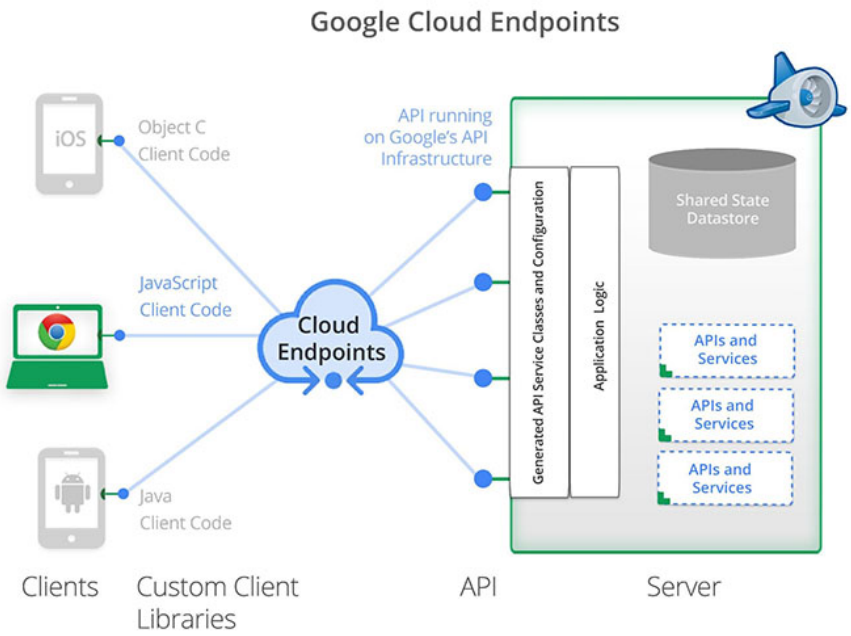
Google App Engine is a platform for building scalable web applications and mobile backends, that provides built-in services and APIs such as NoSQL datastores, memcache, and a user authentication API (useful in our case for the sole taxi drivers authentication, since the application is intended to be used by customers without authentication).

App Engine will scale the application automatically in response to the amount of traffic it receives, so that there are no servers to provision or maintain. It lies on Compute Engine Google virtual machines hosted at Google's infrastructures that take advantage of Google's worldwide fiber network, and It provides open interfaces by Cloud Endpoints service.

2.4.2 Cloud Endpoints

Cloud Endpoints automatically generates client libraries to make wiring up the frontend easy. It creates RESTful services and makes them accessible to iOS, Android and Javascript clients.

Built-in features include denial-of-service protection, OAuth 2.0 support and client key management. Endpoints allows to build client libraries for Android, iOS and web-based clients in Javascript: it wraps the code provided to build an API server. Google Cloud Endpoints requires SSL, so web connections should be wired by https protocol.



2.4.3 Cloud Storage and Datastore

The persistency tier in Google Cloud Platform is managed by Cloud Storage and Datastore services, that differs in the kind of database service they provide (SQL the former while NoSQL the latter). Both grant fast access to the apps data from any location.

We have choosen the cloud storage solution because is based on SQL and therefore on the relational model. Data are replicated across multiple datacenters and this provides a high level of availability for reads and writes. Data are read and written through transactions. A transaction can execute multiple operations. By definition, a transaction cannot succeed unless every one of its operations succeeds; if any of the operations fails, the transaction is automatically rolled back. This is especially useful for distributed web applications like our one, where multiple users may be accessing data at once.

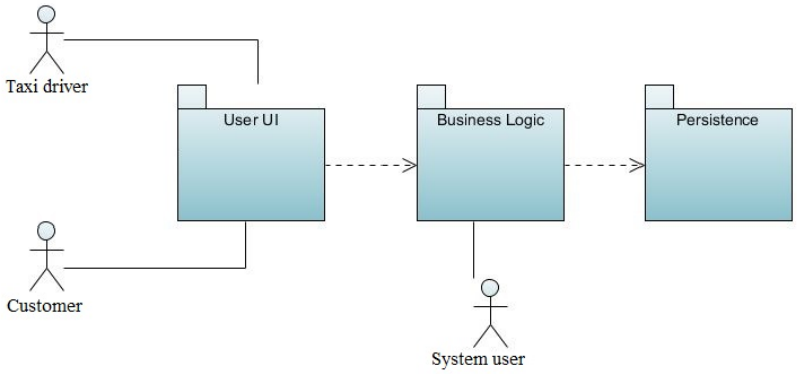
2.5 General Packages Design

This subsection explains the partitioning of code in three main packages: User Interface package, Business Logic package, Persistence package. There is a correlation between use cases (the functionalities) and package design. Each package is coded to accomplish a determinate function and will run on a specific tier:

- User Interface UI: it contains the user interfaces. It is responsible for the interaction with the user such as getting UI requests, referring them to the Business Logic package and retrieving the data back for displaying. It runs on the client tier;

- Business Logic: it contains the business logic components. This package is responsible for handling the User UI package requests, processing them and accessing the Persistence package if required to provide a response. It runs on the business logic tier;

- Persistence: it is responsible for managing the data requests from the Business Logic package. It runs on the persistence tier. The main users (the taxi driver and the customer) access directly to the User Interface UI package. The system user has access to some functionalities encapsulated by the business logic package. None is allowed to access directly to the data.



2.6 Component View

In order to give a further high-level design schema of the application, it was decided to represent the various components of the system through Boundary-Control-Entiry diagrams. BCE diagram provides an overview of the entire system, identifying the main components that would be developed for the product and their interfaces.

In contrast with a low-level design, these diagrams use possibly functional terms instead of more technical ones that should be understandable to the administrators of the system. The need is to provide an overview of how the various sub-systems and components of the system fit together.

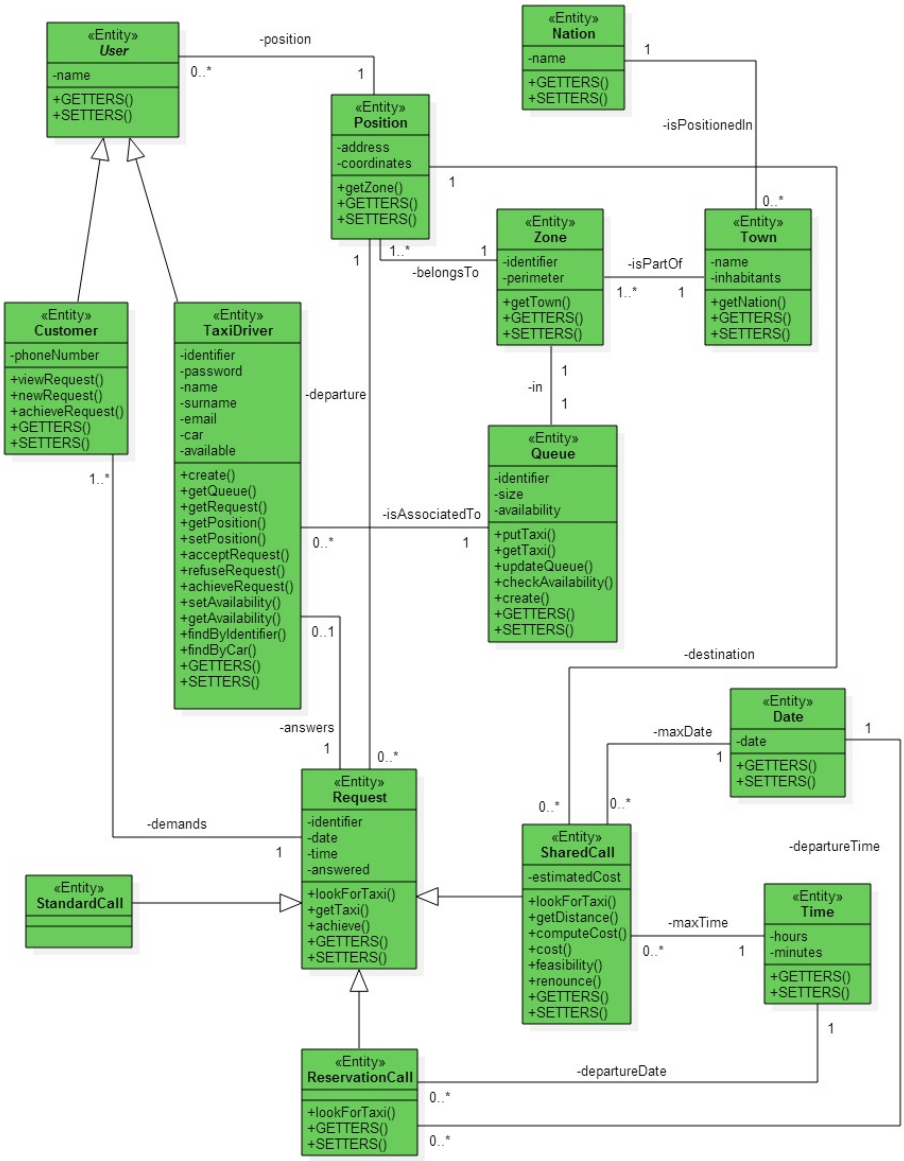
BCE diagram has been chosen because it is very close to represent the Model-View-Controller pattern, where boundaries map to the view, controls map to the controller and entities to the model of the application.

2.6.1 Model-Entity Diagram

This model is the central component of the MVC software architectural pattern. It captures the behavior of the application in terms of its problem domain, independent of the user interface.

The model directly manage the data, logic and rules of the application. In order to point out its importance the first diagram shows the structure of this package, displaying classes and their associations.

It is important to recall that boundaries map to the view that can be any output representation of information, controls correspond to the controller part that accepts input and coverts it to command for the model or the view and finally entities map to the model of the application. In order to prevent the diagrams became too chaotic not all the interactions between classes belonging to the model package are considered, because they are already included in the previous one.

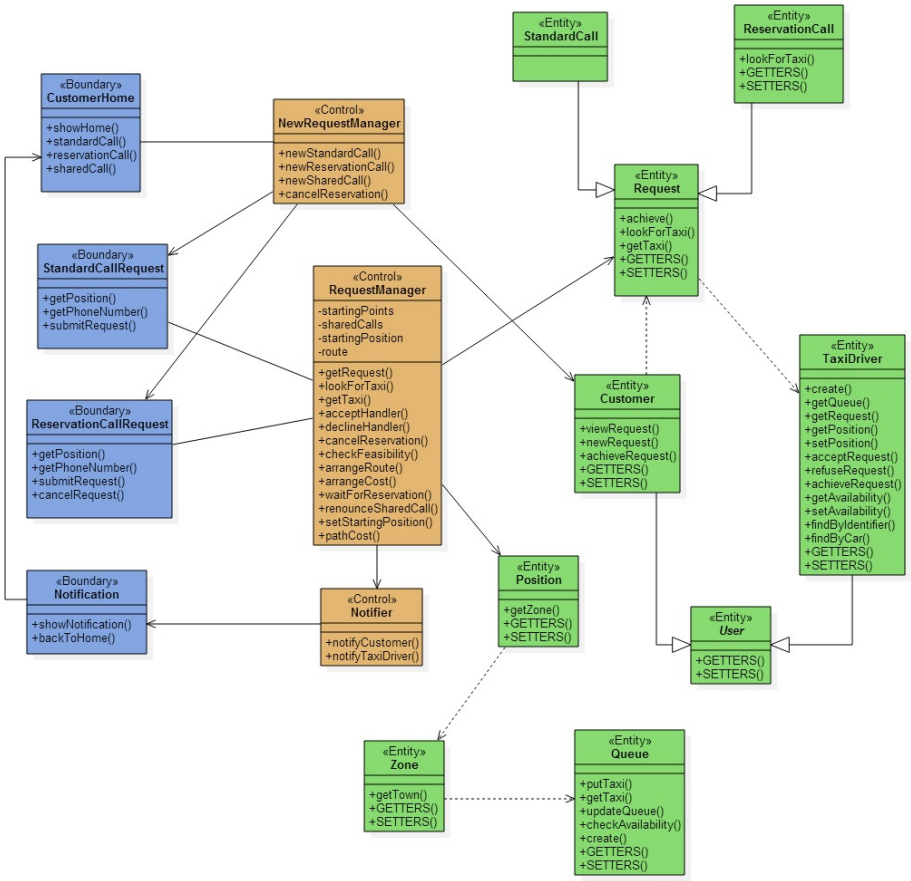

«Entity»
User
-name
+GETTERS()
+SETTERS()
-position
0..*
1
«Entity»
Position
-address
-coordinates
+getZone()
+GETTERS()
+SETTERS()
«Entity»
Nation
-name
+GETTERS()
+SETTERS()
1
-isPositionedIn
0..*
1
«Entity»
Zone
-identifier
-perimeter
+getTown()
+GETTERS()
+SETTERS()
1..*
1
-belongsTo
-isPartOf
1..*
1
«Entity»
Town
-name
-inhabitants
+getNation()
+GETTERS()
+SETTERS()
«Entity»
Customer
-phoneNumber
+viewRequest()
+newRequest()
+achieveRequest()
+GETTERS()
+SETTERS()
1..*
«Entity»
TaxiDriver
-identifier
-password
-name
-surname
-email
-car
-available
+create()
+getQueue()
+getRequest()
+getPosition()
+setPosition()
+acceptRequest()
+refuseRequest()
+achieveRequest()
+setAvailability()
+getAvailability()
+findByIdentifier()
+findByCar()
+GETTERS()
+SETTERS()
-departure
1
-in
1
«Entity»
Queue
-identifier
-size
-availability
+putTaxi()
+getTaxi()
+updateQueue()
+checkAvailability()
+create()
+GETTERS()
+SETTERS()
-isAssociatedTo
0..*
1
0..1
-answers
1
0..*
-destination
«Entity»
Date
-date
+GETTERS()
+SETTERS()
-maxDate
1
1
0..*
0..*
«Entity»
Request
-identifier
-date
-time
-answered
+lookForTaxi()
+getTaxi()
+achieve()
+GETTERS()
+SETTERS()
-demands
1
«Entity»
SharedCall
-estimatedCost
+lookForTaxi()
+getDistance()
+computeCost()
+cost()
+feasibility()
+renounce()
+GETTERS()
+SETTERS()
-departureTime
«Entity»
StandardCall
-maxTime
0..*
1
«Entity»
Time
-hours
-minutes
+GETTERS()
+SETTERS()
1
«Entity»
ReservationCall
+lookForTaxi()
+GETTERS()
+SETTERS()
-departureDate
0..*
0..*

2.6.2 Taxi Driver View Model

The diagram shows how the various components of the system interact to each other when a taxi driver, a specific class of user, uses the application.

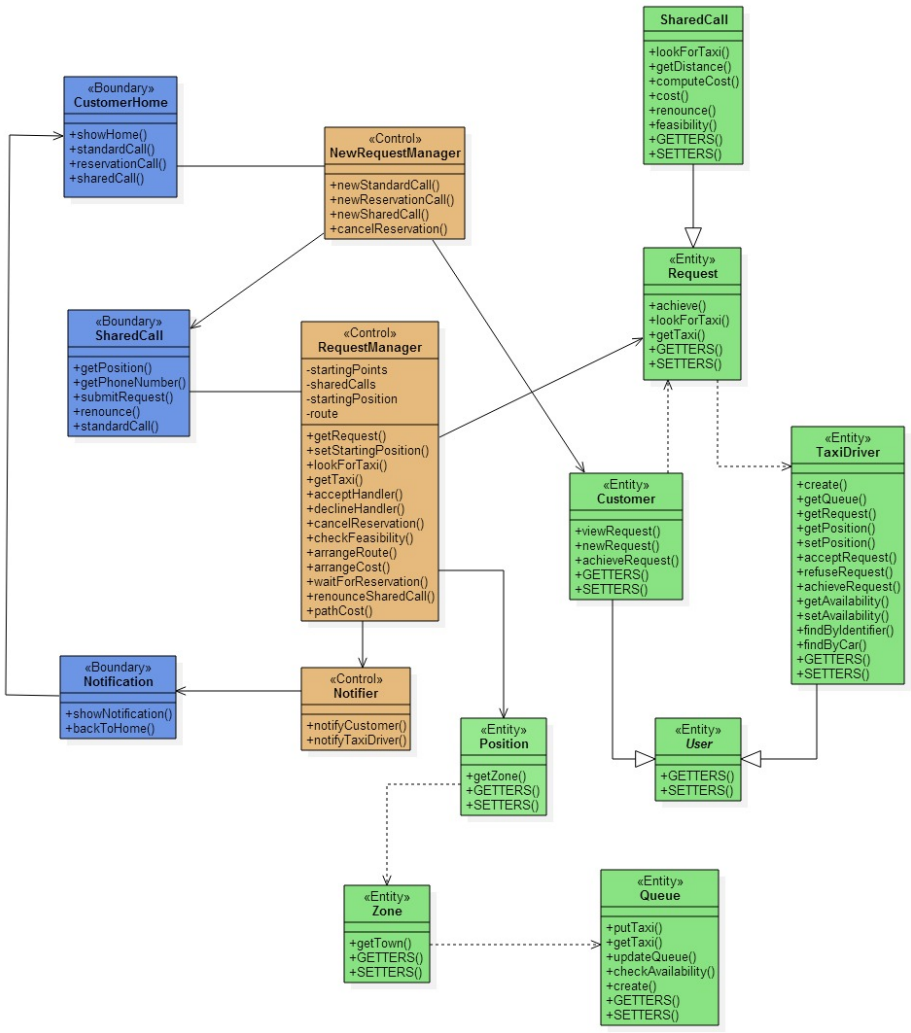
2.6.3 Standard-Reservation View Model

The diagram shows how the various components of the system interact to each other when a customer are using the application to request a standard call or a reservation one.



2.6.4 Shared Call View Model

The diagram shows how the various components of the system interact to each other when a customer is requesting a shared call.

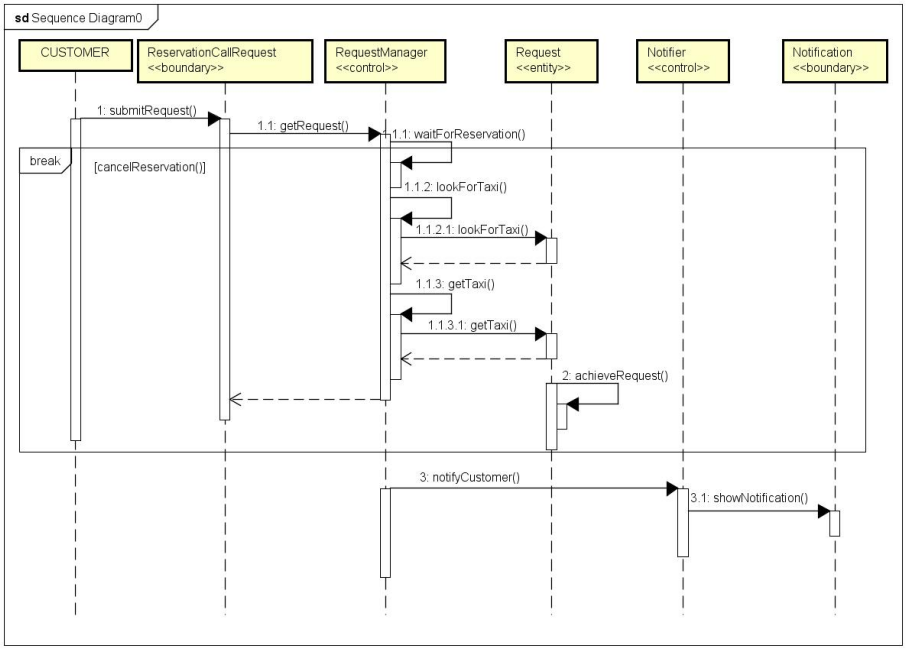


2.7 Runtime View

In this subsection are presented some sequence diagrams about two important use cases to describe the way components interact.

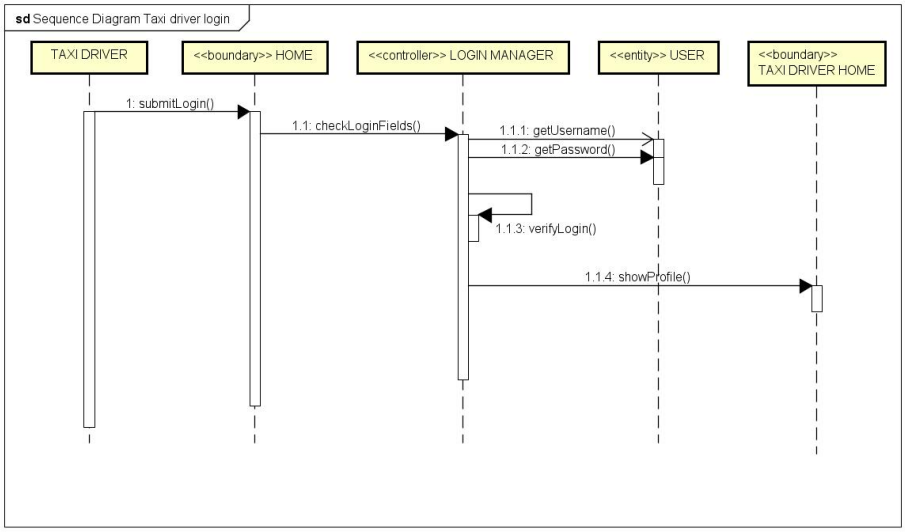
2.7.1 Reservation Sequence Diagram

The first sequence diagram illustrates the interaction among the components of the system during the positive ending process of login by the registered taxi driver. The procedure begins with the submission of the credentials by the user, involves components of model, view and controller as showed and concludes with the taxi driver personal home page loading.



2.7.2 Taxi Driver Login Sequence Diagram

The second sequence diagram illustrates the interaction among the components of the system during the procedure of logging in made by the taxi driver.



2.8 Component Interfaces

A component interface is the set of public methods of that component that are available from the outside. In our system, component interfaces are offered by the persistence tier to the application tier and by the business logic tier to the user interface tier. The use of high-level interfaces allows and simplifies the integration of new modules to the application and also its maintainability.

The database tier offers an external interface to provide to the business tier useful methods to access to the data, make a query, insert, upload and delete a tuple.

The business logic tier grants to the user interface tier functions to make a taxi regular call, a taxi reservation, a reservation cancellation and a taxi shared call. These four methods are implemented by the class (¡¡control¿¿) NewRequestManager.

Below there is a cursory description of the methods' parameters, their return values and comments about what they are thought to do:

- newStandardCall(): it receives the position where the request comes from as a parameter. It search for the first answering taxi driver of the queue in the zone corresponding to that position and returns the taxi driver id and the waiting time for it;

- newReservationCall(): it receives the starting position and time of the reservation as parameters. It stores the reservation request and returns its identification number;

- cancelReservation(): it receives the reservation id number that it to be cancelled and accomplishes his cancellation. It returns void;

- newSharedCall(): it receives the starting and the arrival position, the date and the maximal departure time of the shared call as parameters. It stores the shared call request, verifies if there are any other shared call that can be coupled with that in case and returns the shared call identification number.

2.9 Database Design

Data are stored into a relational database. This model organizes data into tables (or *relations*) of columns (*attributes*) and rows (*tuples*), with a unique key identifying each row. Generally, each relation represents one "entity type" (such as user or taxi request). The rows represent instances of that type of entity and the columns represent values attributed to that instance. Relationships exist among the tables. These relationships take three logical forms: one-to-one (1:1), one-to-many (1:N), or many-to-many (N:N).

In the following section is provided a detailed description of the database by providing both the conceptual and the logical design diagrams.

2.9.1 Conceptual Design

Conceptual design allows to summarise the data to store and the relations between them.

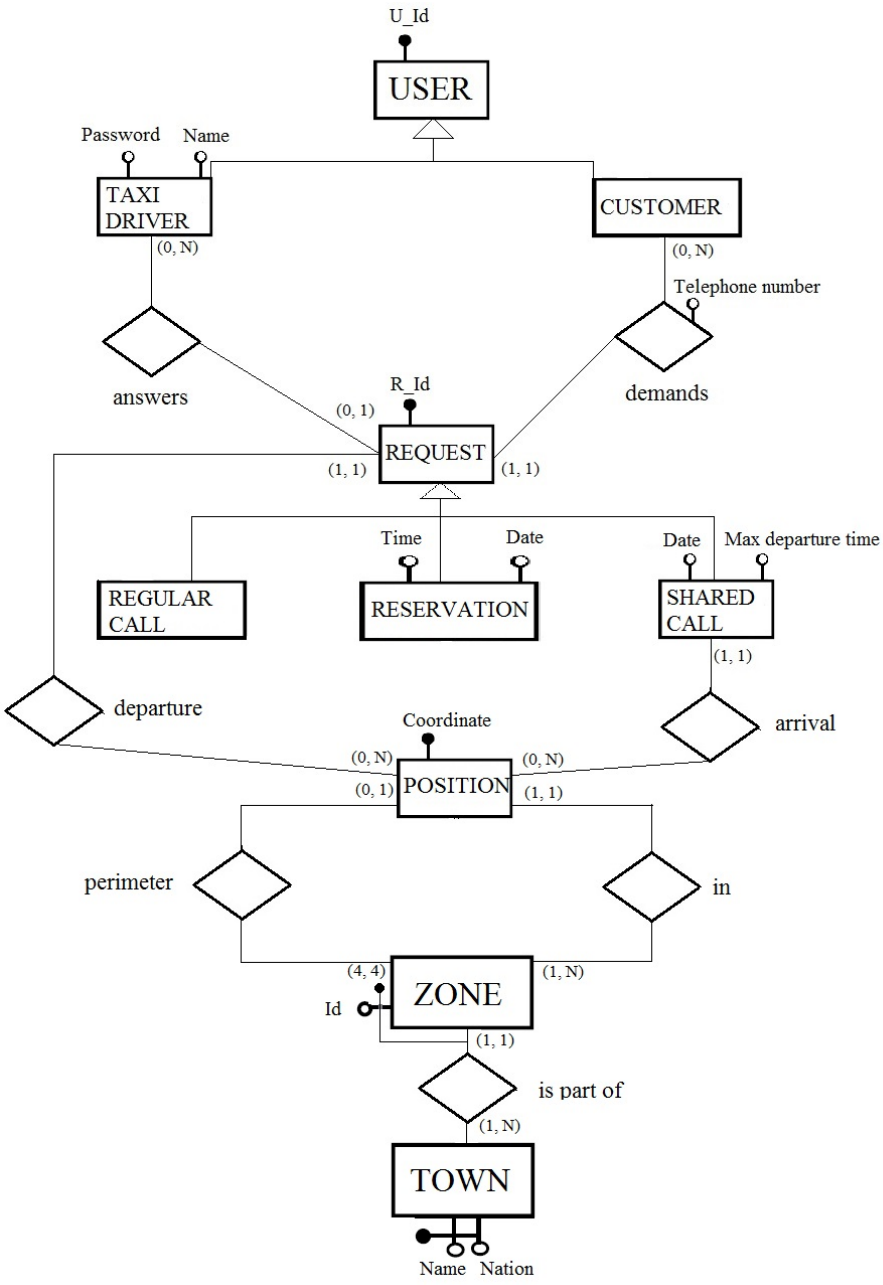
Towns are classified through the entity TOWN, which presents a compound key NAME,NATION. Zones are stored as object of the weak entity ZONE, that is identified by an Id and his corresponding town's key. Positions are unequivocally memorized by the database with the entity POSITION and his primary key COORDINATE.

It is necessary to save informations about two type of users: the taxi driver and the customer. Therefore our database consists of a father-entity USER and two child-entities TAXI DRIVER and CUSTOMER. USER has a numerical U_Id as key. TAXI DRIVER is characterized by a name and a password. To store the requests the database provides another father-entity REQUEST with a numerical R_Id as key.

Entities REGULAR CALL, RESERVATION and SHARED CALL inherits it from the super-entity. Moreover, RESERVATION has two attributes TIME and DATE and to SHARED CALL belong MAX DEPARTURE TIME and DATE. Entities are related to each other through the following relations. IS PART OF between TOWN and ZONE describes that each zone is part of a single town and each town stored in database is divided in more zones.

Relations IN and PERIMETER are designed between the entities ZONE and POSITION. The former specifies that each position belongs to exactly one zone and each zone has more than one position. The latter explains that each zone has exactly four positions identifying his perimeter and that each position could be part of at most the perimeter of a single zone. Relation DEMANDS between CUSTOMER and REQUEST illustrates that each customer could make some requests specifying his telephone number as relational attribute, but each request is made by an unique customer.

The same reasoning is followed by the relation ANSWERS between TAXI DRIVER and REQUEST: each taxi driver could answer to some requests, but each request could be answered just by a single taxi driver. DEPARTURE is a relation between the entities REQUEST and POSITION and explains that each request has exactly one departure position and each position could be the starting point of more requests. Finally, each shared call has an unique arrival position and each position could be arrival for more shared call and that is what is specified by the relation ARRIVAL between the entities SHARED CALL and POSITION.

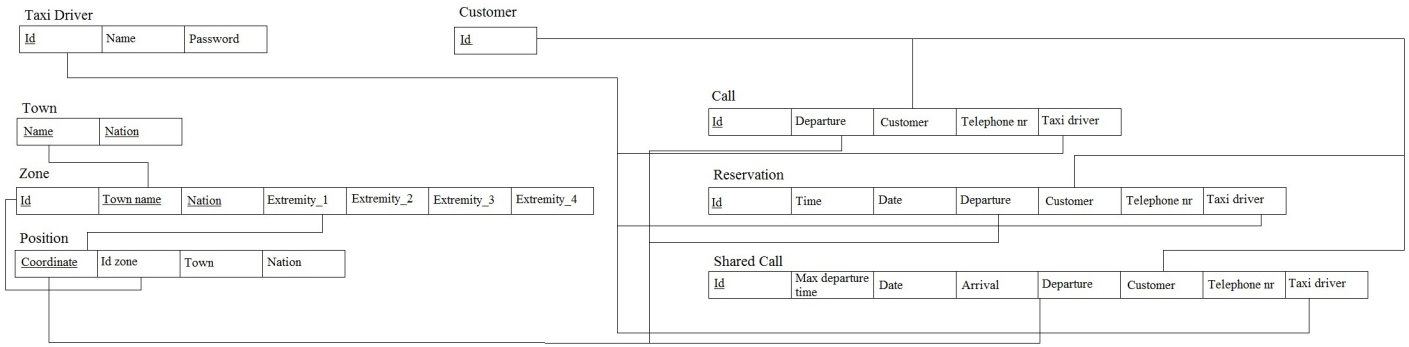

U_Id
USER
Password
Name
TAXI DRIVER
CUSTOMER
(0, N)
(0, N)
Telephone number
answers
demands
R_Id
(0, 1)
REQUEST
(1, 1)
(1, 1)
Time
Date
Date
Max departure time
REGULAR CALL
RESERVATION
SHARED CALL
(1, 1)
departure
arrival
Coordinate
(0, N)
(0, N)
(0, 1)
POSITION
(1, 1)
perimeter
in
(4, 4)
(1, N)
Id
ZONE
(1, 1)
is part of
(1, N)
TOWN
Name
Nation

2.9.2 Logical Design

The logical schema represents the problem domain in terms of either relational tables and columns. The model is generated from the conceptual ER diagram. That means that the entities are translated into tables, the attributes become columns of the tables, the N:N relations become tables in the LD diagram and the 1:N relations are modelled by using foreign keys.

Hierarchies are eliminated and each child-table acquires the attributes of the father. Each table has a primary key (that can be simple or composed) and that is unique for each tuple.

The keys of the tables TAXI DRIVER, CUSTOMER, REGULAR CALL, RESERVATION and SHARED CALL are of type integer and are auto-increment columns. As a consequence when a new tuple is inserted into the table it is not necessary to take care about the value of the key column, because it is assigned automatically by the DBMS.


Taxi Driver
Id
Name
Password
Customer
Id
Town
Name
Nation
Call
Id
Departure
Customer
Telephone nr
Taxi driver
Zone
Id
Town name
Nation
Extremity_1
Extremity_2
Extremity_3
Extremity_4
Reservation
Id
Time
Date
Departure
Customer
Telephone nr
Taxi driver
Position
Coordinate
Id zone
Town
Nation
Shared Call
Id
Max departure time
Date
Arrival
Departure
Customer
Telephone nr
Taxi driver

2.10 Other Design Decisions

As established in the section 3.2.2 Design Constraints, the application integrates an already existing GPS system that can keep track of the position of any registered vehicle at any moment when the corresponding taxi driver is set on available. The connection between the two systems happens through a dedicated API by the GPS that provides to the business logic the methods to access to the location informations for each taxi driver. Furthermore, myTaxiService acquires the current position of a customer when he/she wants to set it as the starting position of his call. In this case that is made possible thanks to the interaction of the client tier with another system, in particular the GPS integrated in the mobile phone. As above, the connection between the two applications is made possible thanks to a dedicated API by the mobile

3 Algorithm Design

This subsection contains the definition of the algorithms that according to us are the most relevant to describe for your system.

3.1 lookForTaxi()

This method belongs to the class Request of the model package and returns an instance of the class TaxiDriver to which send a request. The class request has as attribute an instance of the class Position. At first the method obtains the zone corresponding to that position using the method getZone() of Position and then gets the corresponding queue using the method getQueue() of the class Zone. The class Queue contains a data structure in which the TaxiDriver entities in the collection are kept in order and the principal operations on the collection are the addition of entities to the rear terminal position, known as enqueue, and removal of entities from the front terminal position, known as dequeue. This makes the queue a First-In-First-Out (FIFO) data structure. lookForTaxi() returns the TaxiDriver instance on the top of the queue without deleting it from the data structure.

3.2 getTaxi()

This method belongs to the class Request of the model package and returns the instance of the class TaxiDriver corresponding to the taxi driver who has already accepted that request. This method works analogously to the previous one, but when it returns it delete the chosen TaxiDriver instance from the queue.

3.3 updateQueue()

This method belongs to the class Queue of the model package and does not return anything. It performs on its data structure dequeuing the top entity and enqueuing it.

3.4 arrangeRoute()

This method belongs to the class RequestManager of the controller package and returns an array of objects Position belonging to the corresponding class of the model package, sorted in such a way as to have the position of the first passenger to get off in the first position of the array and gradually move forward. First of all it calls the method startingPoint(), a method of the same class, that sets the origin of the ride selecting one of the fixed starting points specified as an attribute of the class. This method selects a trade-off over all the starting points of the zone getting for each of them the distance between all the origins of the ride specified in each request, summing them, getting a sort of total distance and finally choosing the position having the minimum total distance. The method continues creating an array of objects Position that lists all the destinations belonging to all the shared calls that have been selected. Now, the array is sorted through a private method sortArrayByDistance() that binds each position to a distance got from the getDistance() method of the class SharedCallRequest and then incrementally orders it thorough these distances. The method finally set the route attribute using this local array.

3.5 arrangeCost()

This method belongs to the class RequestManager of the controller package and sets for ea each request an integer that represents an approximate estimation of the cost of the trip in euro. The method begins creating a local copy of the attribute sharedCalls and sorting it over the positions defined in the attribute route. At this point it creates a queue and, from position 0 to the array size -1 position, all the shared calls are putted in to the queue. This data structure has a FIFO logic and so the top element will always be the next customer to get off. A local variable origin is created and initialized to the attribute startPosition.

First loop led by the condition $queuesize >= 1$ pathCost equal to cost (starting, queue_value()) (a method of Request, where queue_value() gets the top element of the queue without deleting it) second loop led by all the shared calls in the local array sharedCalls

calls the specific shared call the method elaborateCost(pathCost, queue_size()) (the method sets an estimated shared cost for the shared call for each step and for a specified number of customers participating it)

if statement led by position on top of the queue equal to the sharedCall destination

delete the sharedCall in the local array (the passenger gets off)

end if

end second loop

starting equal to queue_get() (where queue_get() gets the top element deleting it)

end first loop

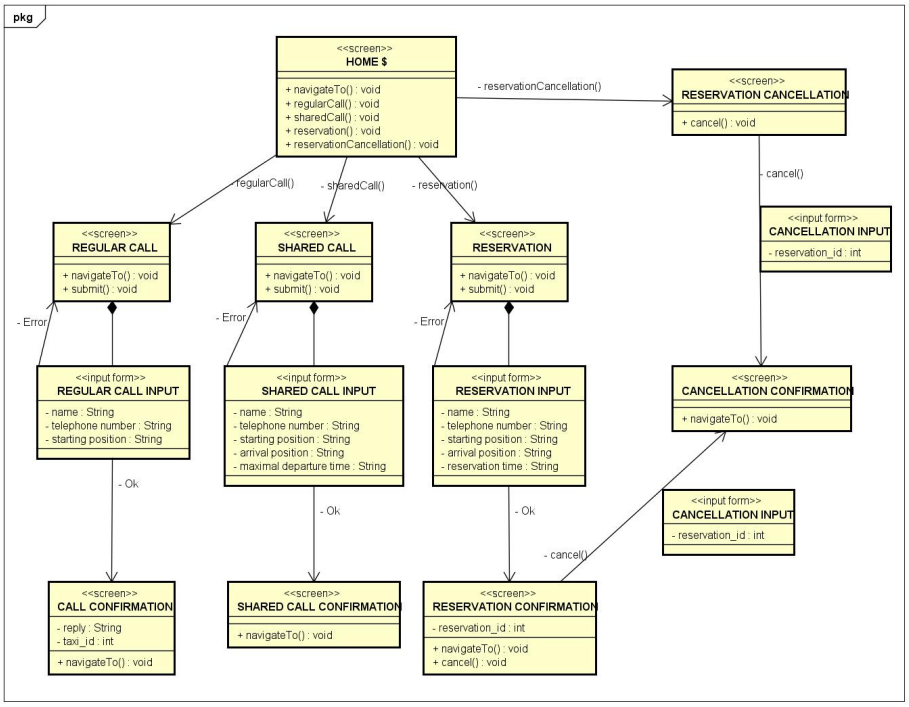
Now all the requests have an estimated cost of its personal route.

4 User eXperience Design

The User eXperience diagram is used to describe the interaction between the user and the system as a finite number of possibly step. In particular, it shows the resources available to the user (either the taxi driver or the customer), what contains the screen in front of him/her, the choices he/she could take, his/her possible activities and which are the possible change of state (and screen).

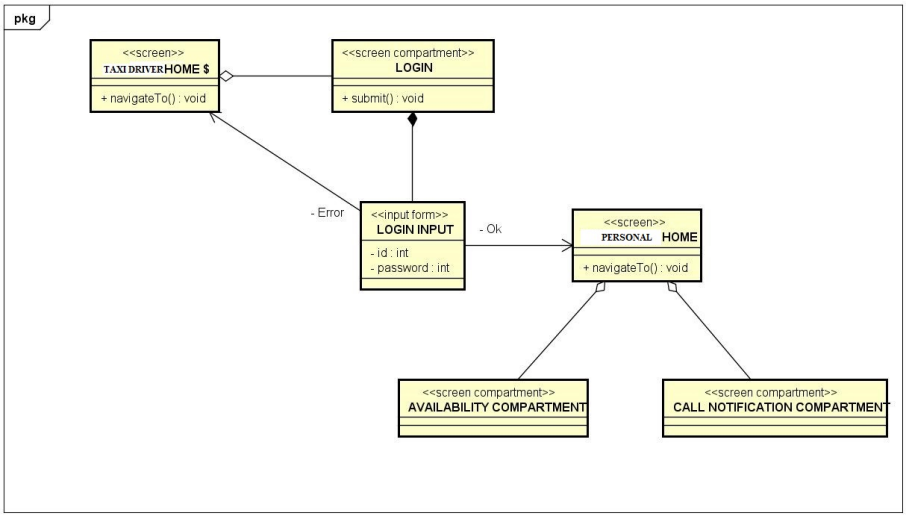
4.1 Customer Side UX

From home page a customer can choose whether to make a regular call, a shared call or a reservation. In each of these cases, a new screen is loaded and he is invited to fill the corresponding form. When the inputs are not correct the screen is reloaded. When the form is correctly submitted, the system elaborates the request and as soon as possible gives an answer. Only a reservation request can be cancelled (only until two hours before the departure time). That can be done inserting the reservation id both from the reservation confirmation page and from the home page. The home page is reachable from each other screen to accelerate and simplify the user's surfing.



4.2 Taxi Driver Side UX

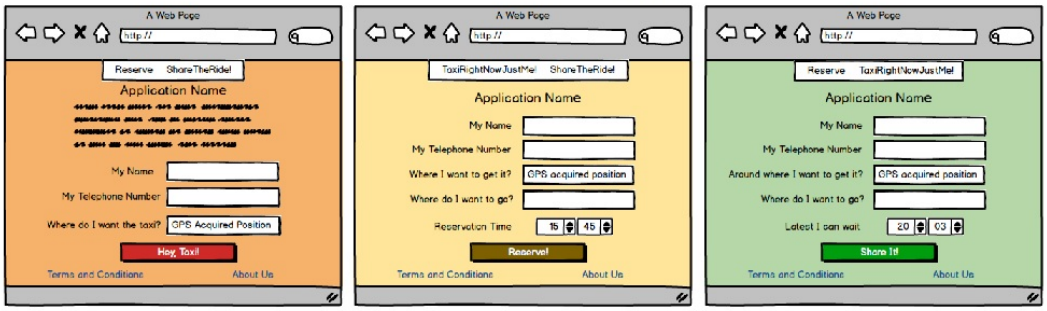
The taxi driver is invited to log in from his home page. Log in is possible through a dedicated screen compartment. When the inputs are wrong the taxi driver is re-conducted to his home page. Otherwise he/she is directed to his/her personal page from which he/she can modify his availability and accept and refuse the requests.



4.3 User Interface Design

The mock-ups proposed in the RASD in chapter 3.1 User Interface Requirements are really simple. They lend to the application a great usability and a friendly and young appearance. Below there is a recap of those mock-ups.

Regular call for mobile | Reservation for mobile | Shared call for mobile | Taxi driver personal page for mobile



Regular call for the web | Reservation for the web | Shared call for the web

5 Requirements Traceability

The choices made and explained in this document allow to satisfy all the functional and non functional requirements imposed by the Requirements Analysis and Specification Document. For each section of the supporting material and for each decision taken have already been illustrated the reasons and the consequences.

In particular, the client-server style contributes to the organization of the required distributed system and the MVC pattern manages to the order of the code.

Software interfaces satisfy the requirement to facilitate the development and the integration in the future of additional services on top of the basic one.

Furthermore, the use of the server as a service by Google consents to achieve an high-level standard regarding performances, maintainability, availability and security, while the usability of the application is guaranteed by the simplicity of the user interface.

6 RASD Modifications

According to the functional requirement R2 from the RASD the system has to collect and store the telephonic numbers that customers will provide at their first access as contacts for their following requests.

Later we thought that it's better to not couple necessarily each customer with a phone number, but to associate instead each request with a telephonic contact. In this way when a customer has low-battery can still access the service from other platforms (for example, from a cellphone lent by a friend, having provided his number in the first moment).

7 Appendix

7.1 Software Employed

The following is a list of the softwares used to redact the present document:

- *MiKTeX 2.9*: main LaTeX environment.
- *TeXstudio* : LaTeX editor used to redact the document.
- *BalsamiqMockups 3*: to create interface mockups.
- *Astah Professional*: to create the different kinds of diagrams included.
- *starUML*: to create the UML-like diagrams.

7.2 Working Hours

Fabio Catania: ca. 30 hours.

Matteo Di Napoli: ca. 30 hours.

Nicola Di Nardo: ca. 30 hours.