Politecnico di Milano
A.A. 2015-2016

Software Engineering 2: myTaxiService

Design **D**ocument

Matteo Bulloni (852676), Marco Cannici (852527)

4 December 2015

Contents

1 Introduction

1.1 Purpose

This document is intended to be a detailed design supplement to the already provided RASD, aiming to offer a specific evidence on how myTaxiService application shall be concretely designed and implemented. The focus of this document is placed on the presentation of the architectural structure of the system, which dives from its high-level anatomy to the detailed descriptions of the single components. The document also aims to provide an insight on which specific patterns, algorithms and architectural styles were chosen in designing the system, and why.

1.2 Scope

In describing the architectural structure of myTaxiService application, we will offer a perspective on the design of the whole system, thus including both the front and back-end of the application. The documentation provided will therefore start with the definition of the n-tiered structure in which the system has been chosen to be divided into, and then cover an analysis of the structure of all the system's tiers, dealing with each in the way that was thought to be the most appropriate to describe that specific part of the system.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

Availability Status: represents the current status of a logged in driver user: "available" to be forwarded and handle new requests, or "unavailable" if already busy or just not currently able to deal with new ride or reservation requests.

Availability Timestamp: attribute of the Driver objects stored in the Drivers Model, it represents the timestamp of the last status switch of that driver user from "unavailable" to "available". It is written by the system itself every time a driver changes his availability status to "available", and it is used to deal in a clever way with the possibility of an available driver moving from one city zone to another (see section 3.1.3 for a detailed explanation of this issue).

Busyness flag: attribute of the Driver objects stored in the Drivers Model, it is a boolean value which can be set on "busy" (true) or "not busy" (false). This flag is set on "busy" whenever a driver user is forwarded a ride or reservation request from the system, and it remains like that for all the time the driver is dealing with answering the request, deciding if to accept or refuse it (when allowed). When the driver answers (or if the time to answer runs out), the flag is immediately set back on "not busy", meaning that the driver could be ready to handle new requests (if his

status is still on "available"). This attribute is used to avoid forwarding multiple requests to a single driver, ensuring that any available driver who is forwarded a request has to answer that one before possibly being able to be forwarded a new one.

Driver application: myTaxiService application (mobile only) for driver users. Not publicly available, installed only on taxi terminals, allows drivers to inform the system about their state, to receive and handle ride and reserved ride requests, and to communicate the system when and how a ride started due to a request ends.

Notification: a generic informative message sent by the system to a driver or passenger user. Notifications are shown through popup windows in myTaxiService applications, and are shown only if the user is online when they are received. They can be divided into purely informative (I) and acceptable/refusable (A/R) notifications. Here we list the main situations for which we use the term "notification" sent to a

1. Passenger User:
   (a) to confirm acceptance or refusal of a reservation request (I);
   (b) to remind a reserved ride that has to be performed in the next minutes (I);
   (c) to tell that a sent ride request has been accepted or queued (I);
   (d) to ask if the user wants to confirm or cancel a ride request after he has been informed that the request has been queued (A/R);
2. Driver User:
   (a) to assign a ride request to be performed immediately (A/R);
   (b) to assign a reserved ride, to be performed in the next few minutes (A only);

Passenger application: myTaxiService application (either web or mobile) for passenger users. Publicly available for download on mobile devices and accessible by web, allows the registered users to send ride and reservation requests, and to check and manage their reservation history.

Reservation request: a request that can be sent by a passenger user to book a taxi ride in the future. To send the request, the user is needed to specify a starting point, which is the point where the client wants to be picked up by the taxi, a destination for the ride, and a meeting time and date, which has to be at least 2 hours later than when the request is sent for it to be accepted. This kind of request can be cancelled under certain conditions, and must be accepted by the assigned driver.

Reserved ride: a taxi ride that has been booked by sending a reservation request through myTaxiService application.

Ride request: a request that can be sent by a passenger user for a ride immediately needed, it is myTaxiService's application equivalent of actually calling a taxi. To send the request, the user is only needed to specify a starting point, which is the point where the client wants to be picked up by the taxi, since the meeting time is supposed to be as soon as possible. This kind of requests are forwarded to drivers, and can either be accepted or refused by them.

1.3.2 Abbreviations

- **MVC**: Model View Controller
- **RASD**: Requirement Analysis and Specification Document
- **UX Diagram**: User Experience Diagram
- **ER Diagram**: Entity-Relationship Diagramd
- **DBMS**: Database Management System
- **FIFO**: First In First Out

1.4 Reference Documents

This document refers to:

- The previous 'Requirement Analysis and Specification Document' delivered document
- The provided pattern 'Structure for the design document'
- Lecture slides

1.5 Document Structure

The document will be divided into the following main sections:

- **Architectural Design**: here we will present the n-tier division of the system and their deployment among physical devices. We will then give an insight on the single components of each tier, showing how they interface and interact, providing also examples and diagrams to better understand the components' behavior at runtime. Finally, we will discuss and explain the architectural styles and patterns chosen in designing the system;
- **Algorithm Design**: in this section we will present the most meaningful and less trivial algorithms that are used by the system to perform some of its duties, to clarify in detail how the single components behave in handling those specific tasks;

- **User Interface Design**: here we will provide a further insight on how the user interface of the application is designed, for both client and driver users of myTaxiService app, presenting UX diagrams that go along with the user interface mockups already presented in the RASD;
- **Requirements Traceability**: in this part of the document we will show how the functional requirements presented in the RASD are satisfied by the system, mapping each system component with the specific requirements it is supposed to fulfill;

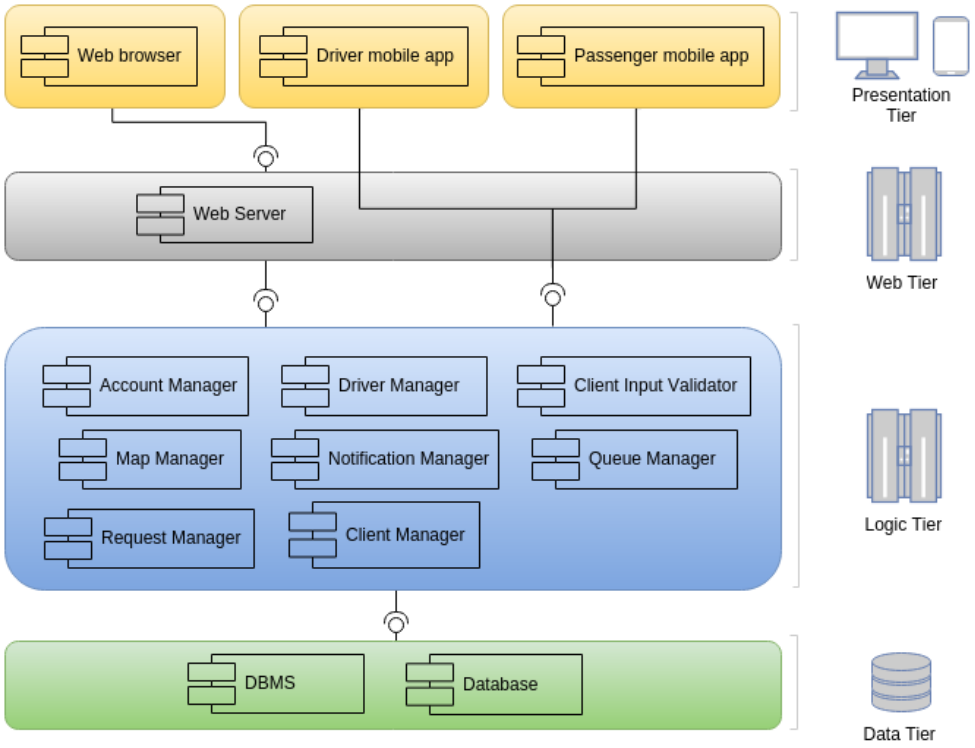
2 Architectural Design

2.1 Overview

In this part of the document we are going to present the architectural design choices we have made towards the development of myTaxiService application. We are going to explain first in how many logical tiers we divided the system's architecture in and why, showing how these tiers are intended to be deployed among physical devices, and then to break each tier up to offer an insight on the single different components that make those tiers up. We will thus explain how and why these various components interface and interact with each other, pointing out specifically which styles and patterns we chose to adopt to build the whole system up in this way, and why we did so. We are then going to provide some detailed examples on how the system behaves at runtime in accomplishing the tasks it has to face, presenting appropriate diagrams that aim to show the sequence of actions performed by the different system components in handling the tasks.

2.2 High level components and their interaction

In the following diagram we present the basic 3-tier structure the system has been divided into, with each tier also giving a first insight on his internal composition:



As shown by the diagram, the system is divided into 4 logical tiers:

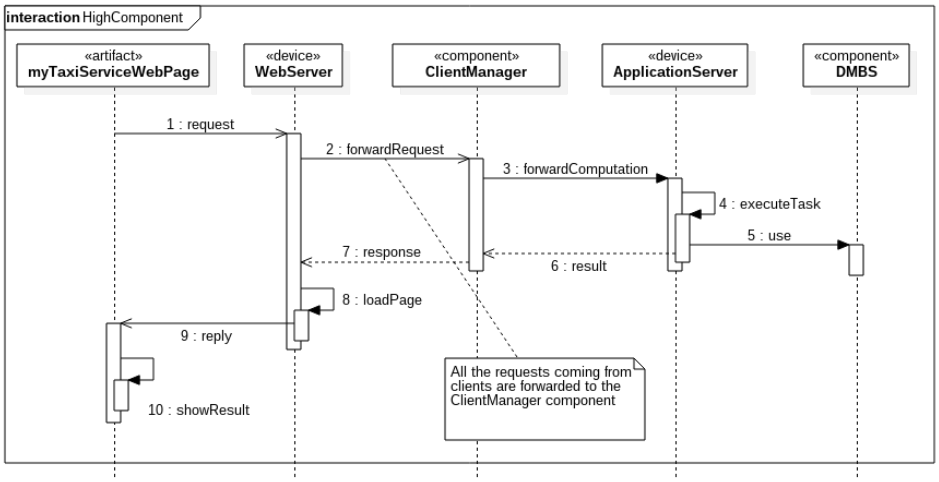
Presentation tier (Client): it is basically the client's tier, for each different kind of client the system has to deal with (web browser or the two different mobile applications). For all the three kinds of client, this tier has been designed to be very thin, leaving all the computations in charge to the Logic tier; it includes in fact just the application's user interface, with the basic input modules and functions better shown in section 6. The choice of making this tier very thin comes mainly from the perspective on how and in which situations myTaxiService application is thought to be used on average, from both passenger and driver users: they need to request or tell something, they need to receive an answer, and they need to do it fast, immediately, in a very simple way. This is why the application - and therefore the logical tier - they use has indeed to be reliable, very quick to respond and to provide feedback for interactions, and very simple. Thus follows that leaving this Presentation tier thin is clearly the best option to choose to achieve these goals.

Web and Logic tiers (Web Server and Server): the Logic tier includes all the elements of the system's server, excluding the central database it relies on. It interfaces with the clients that are using myTaxiService app from web browser through a Web Server, which performs the function of generating and retrieving the requested web pages to the clients, asking

the actual server to compute and provide the needed data. The clients using the mobile app (both the driver and passenger ones) instead, directly interface with the server, invoking some of the Client Manager's exposed functions to send requests to the server. The Logic tier is the one that takes care of all the data computation: as can be seen from the diagram it is divided into several components, acting together in the organization and computation of the data, and relying on a central database (located in the data tier) accessed through a DBMS. At the same time however, a relatively small portion of dynamic (i.e. frequently accessed) data is stored and kept available in a RAM-like memory inside the server itself. Since myTaxiService will initially be available only in one city, the Logic tier has been thought to be deployed on a unique server farm, still powerful enough to ensure reliability and constant availability of an amount of computational power sufficient to face a huge number of simultaneous requests. In case of possible future extensions of myTaxiService services to other cities, it will be considered to organize the different city's servers into a cloud-like system, to ensure a better management of the available computational power. For a detailed description of the different server components please refer to section 2.3.1 "Component View - Logic Tier".

Data tier (Database): this tier includes the central database in which all the persistent data of the system are stored, like user accounts, taxicabs informations, reservations, completed ride requests and so on. It can be accessed through a DBMS, that handles queries and data taking care of security issues and permissions checking. Please refer to section 2.3.2 "Component View - Data Tier" for an ER diagram about myTaxiService's central database.

Here below we present a very simple sequence diagram that provides an intuitive explanation of how the interaction between the 4 tiers happens at high level (specifically in the case of a client using myTaxiService app from web browser, but it's the same for mobile apps, just without the Web Server step):



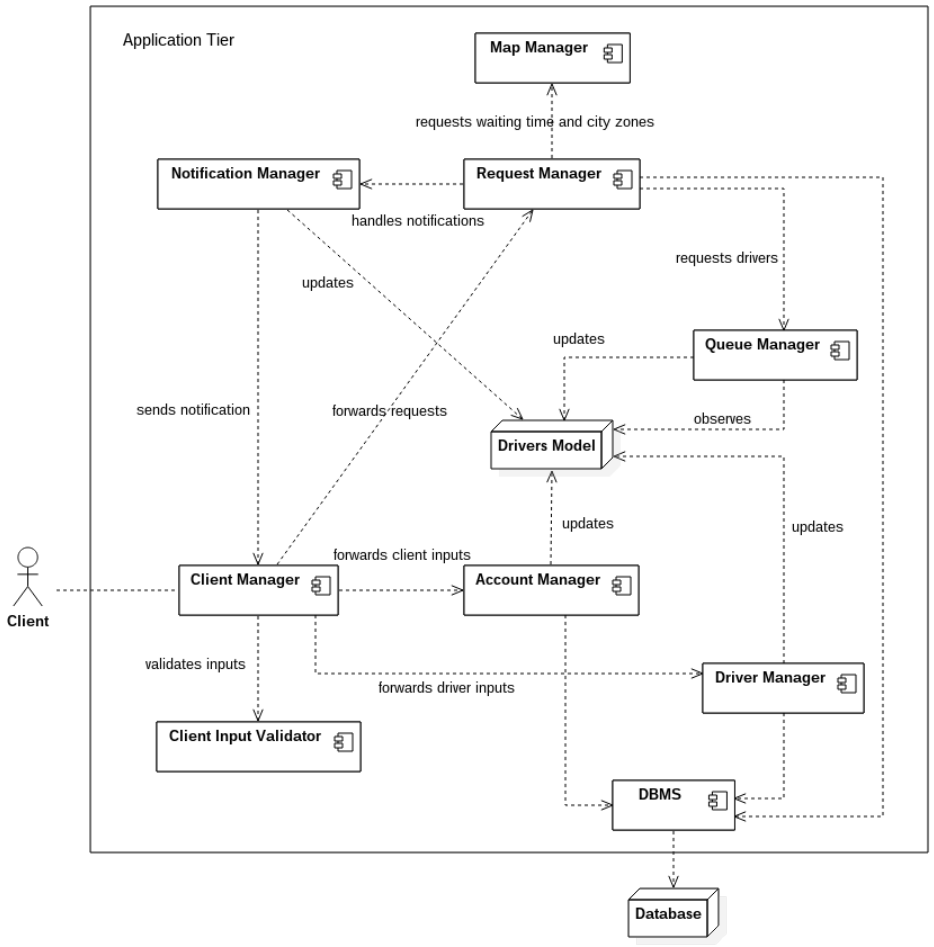
2.3 Component View

In this section we are going to decompose the high level structural elements we presented in the previous section, to present in a more detailed way the actual structure of the different tiers composing the whole system. Hence, we are going to explain how each tier's components work, and how they interface and interact with each towards the fulfilling of the application's goals. Keep in mind that, since this section is mainly focused on giving a representation of the static structural aspects of the system components, more precise and readable informations about the flow of interactions between the different components in handling the system's tasks will be given in section 2.5 "Runtime View", where some meaningful sequence diagrams will be presented to give an insight on the flow of events inside the system in the most common working scenarios. Keep also present that more specific informations about the interfaces the components expose to each other will be given in section 2.6 "Component Interfaces", and that the mapping of components' functions into the corresponding fulfilled functional requirements, among the ones expressed in the RASD, will be presented in section 5 "Requirements Traceability

2.3.1 Logic Tier

2.3.1.1 Components Overview

Here we present a diagram that aims to provide an overview of the structure of the system's logic tier, i.e. the system's server, showing his components and their interactions. Below the diagram, a brief explanation of the server's behavior is given, while further on is given a detailed description of all the single components.



2.3.1.1.1 Clarifications on "Drivers Model"

To immediately clarify, the Drivers Model isn't really a "component" of our server, given that it doesn't actively perform any action (the name in fact refers to the MVC pattern: it's not a controller, it's part of the model). However, it has been included in this diagram and described in the next pages as like the actual components because of its fundamental role it plays in the system. It is in fact the local representation of the currently operating drivers, which some components keep updated according to what the drivers are currently doing, while some others react, as observers, when some change is detected in the driver objects due to those updates. Given that not many informations about the driver users are actually needed in performing the server's routine tasks, that these informations have to be accessed and modified very frequently

and that the number of active drivers in each moment is quite manageable, it has been chosen to keep this data structure available at a higher and more quickly accessible level with respect to the system's central database, which instead contains all the informations of all the users in a more detailed way, as displayed in the brief class diagram provided in the RASD and later in section 2.4 "Deployment View".

2.3.1.1.2 General Description

The server tier of the application is structured as shown in the diagram above. The interactions with the clients all occur through the Client Manager, which manages the mapping of the exposed component interfaces between outside and inside the server itself, and filters any contents coming from the clients, supported by the Client Input Validator, to ensure that only safe and well structured data reach the inner components. Of these inner components, we have the Account Manager responsible of dealing with any request concerning user accounts' attributes and login/logout operations (and thus involving interactions with the central database through the DBMS), and of creating and destroying objects in the Drivers Model. The Driver Manager will then keep their position and status updated, receiving informations from each driver's application, and will also store any ride a driver reports as completed in the central database. The Queue Manager is responsible for all the operations concerning the driver queues associated to each city zone, from keeping them constantly updated, strongly relying on the observer pattern applied to the Drivers Model, to retrieving pointers to the next available drivers when requested by the Request Manager. This Request Manager is the core component of the server, since it manages and coordinates all the processing of incoming ride and reservation request: it takes care of organizing them in proper queues, of handling parallelism and busy situations in which there aren't available drivers, of finding a driver to take care of each request asking the Queue Manager, of ensuring that drivers are forwarded and associated the requests they are assigned to and users are notified quickly about any update of the situation thanks to the Notification Manager. Thus, the Request Manager is indeed the core of the application's server, coordinating the other main inner components. Last, but not least, there's also a Map Manager, that interacts with Google Maps' API, and mainly serves the function of computing the estimated waiting time to be communicated to passenger users when a taxicab is heading to pick them up. Furthermore, the Map Manager stores informations about myTaxiService's availability boundaries (since the application shall only serve clients within the city's boundaries) and city zones' boundaries, helping the other components when they need to convert addresses to GPS coordinates or vice versa, and to map GPS coordinates into the corresponding city zone. Given this general perspective on the system's composition and behavior, here follows the detailed description of each component.

2.3.1.2 Specific Components Descriptions

2.3.1.2.1 Drivers Model

Drivers Model	
Purpose	Store informations about currently operating (logged in) driver users, in a RAM-like memory
Tasks Performed	Keep one instance of "Driver" object for each currently logged in driver user, each containing the following informations about the driver: • Driver ID • Taxicab ID • Current location (city zone and coordinates) • Current ride • Availability Status • Availability timestamp • Busyness flag
\Interacts With	(in a passive way of course) Account Manager, Driver Manager, Queue Manager, Notification Manager

2.3.1.3 Client Manager

Client Manager	
Purpose	Actually interface the clients with the server
Tasks Performed	• Provides a series of interfaces for the different kinds of clients (passenger application, both web and mobile, driver mobile application) exposing the functions and services that can be used by that certain category of clients, mapping them into the actual methods of the different server components in charge to provide those services • Handles the communication from server to clients, retrieving the informations asked by clients and provided by the server components, and taking charge of delivering notifications and retrieving answers, forwarding them to the proper server component • Checks every client's request and provided data through the Client Input Validator, verifying their correctness and harmlessness before forwarding them to the server's internal components
Interacts With	Clients, Client Input Validator, Account Manager, Request Manager, Notification Manager

2.3.1.4 Client Input Validator

Client Input Validator	
Purpose	Check the validity of the inputs provided by clients in any situation
Tasks Performed	• Checks consistency and validity of user registration informations • Checks consistency and validity of any account information editing performed by users or admins • Checks consistency and validity of ride and reservation requests informations (valid date and time, valid location, ...) • Checks consistency and validity of reservation deletion attempts done by passenger users
Interacts With	Account Manager, Request Manager, Notification Manager

2.3.1.5 Account Manager

Account Manager	
Purpose	Handle user accounts registrations, logins and logouts and users' information retrieval and update
Tasks Performed	• Handles user registration requests, interacting with the DBMS • Handles login and logout requests from clients, interacting with the DBMS • Updates the local Drivers Model creating an instance of "driver user" object when a driver user logs in, and removing it when he logs out • Handles requests to view and operate on personal reservation history coming from passenger users • Handles requests to view and edit any account informations
Interacts With	Client Input Validator, DBMS, Drivers Model

2.3.1.6 Driver Manager

Driver Manager	
Purpose	Keep updated the Drivers Model updating location and state of driver users as they change
Tasks Performed	• Periodically checks and updates the location of all the currently available drivers, using informations coming from the GPS modules present on taxicabs and elaborating them through the Map Manager • When a driver object is set on "busy", immediately retrieves and updates his location, to allow a more precise waiting time estimation Keeps updated each driver's availability status in the model, also writing the Availability Timestamp when a driver's status is switched on "available" • Whenever a driver reports he has completed a ride, retrieves the pointer to the ride from the Drivers Model, updates the ride's state (depending on if the driver communicated "ride terminated" or "missing client") and takes care of storing it into the central database, associated to the driver that performed it. Then, empties the Current Ride field of the driver in the Drivers Model and sets back its status on "available", writing the Availability Timestamp.
Interacts With	Driver Application Clients, Drivers Model, DBMS

2.3.1.7 Queue Manager

Queue Manager	
Purpose	Keep updated the taxi queues of the different city zones, and retrieve available taxis when requested
Tasks Performed	• Keeps an available drivers queue for each city zone, in which only currently available drivers who are currently in that zone are present • Retrieves the first available driver from the target zone's queue when requested by the Request Manager, setting it on "busy" and removing it from the queue • Insert drivers at the bottom of the corresponding city zone's queue when they become both "available" and "not busy" (when one of the two values changes, checks also the other one) • Removes drivers who switch to "unavailable" from the queue they were currently in • Moves drivers who move from one city zone to another (while being on "available" status) from that zone's queue to the new one, inserting them not at the bottom, but basing on their Availability Timestamp
Interacts With	Request Manager, Drivers Model

2.3.1.8 Request Manager

Request Manager	
Purpose	Handle incoming ride and reservation requests
Tasks Performed	• Keeps a queue for the incoming ride requests, processed as a FIFO • Keeps a reservation requests queue, ordered from the closest to the farthest to be completed • Keeps a reservation waiting queue and a ride waiting queue for each city zone, in which requests are placed when there aren't available drivers in the zone the request comes from, so that they are re-processed as soon as a new driver becomes available in that zone (queues are monitored using an observer pattern). In case of both reservation and ride requests queuing for available drivers in the same city zone, the reservation requests queue has to be emptied first • Processes ride requests in parallel (creating a thread for each one in the queue); for each request, asks the Queue Manager for an available driver and forwards him the request through the Notification Manager, repeating the process until a driver accepts the request; • Processes reservation requests, first storing them when they are accepted, and then using a scheduler: the scheduler checks every hour which reservations have to be performed from 60 to 120 minutes from now, and creates a thread for each one of them, that wakes up 10 minutes before the reservation time, asking the Queue Manager to retrieve an available driver to forward the request to. While asking for drivers, reservation request threads are given higher priority with respect to ride requests in retrieving an available driver. Moreover, they ask for available drivers also in contiguous city zones if they can't find one in the zone the request is coming from. This is done to ensure that reservation requests are always processed successfully and fastly, and virtually never put in waiting queues

	• After a ride or reservation request is accepted by a driver, sends through the Notification Manager a confirmation to the user who had made the request, telling the incoming taxi code and the estimated waiting time (calculated through the Map Manager) for ride requests, and a reservation reminder with the incoming taxi code for reservation requests
Interacts With	Queue Manager, Notification Manager, Map Manager, DBMS

2.3.1.9 Map Manager

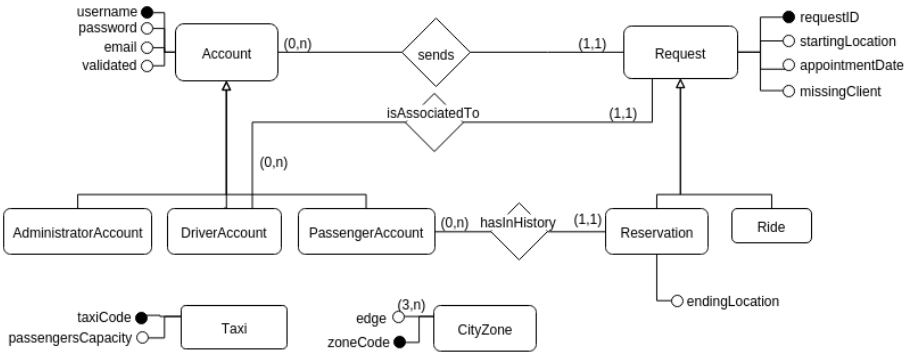
Map Manager	
Purpose	Interact with Google Maps' API in order to operate with locations, coordinates and estimate waiting times
Tasks Performed	• Called by the Request Manager, interfaces with Google Maps' API to calculate the approximate waiting time for a passenger waiting for his taxi, providing the driver's current location as a starting point and the ride's starting point as destination point, and returning the travel time suggested by the API • Called by the Client Input Validator, verifies the integrity and allowed or unallowed placement of the given address or coordinate, operating the needed conversions using Google Maps' API • Stores informations about the virtual boundaries of the city zones and of the area in which myTaxiService can be used • Called by Driver Manager and Request Manager, maps a given GPS coordinate into the corresponding city zone, or a given string address into a GPS coordinate
Interacts With	Request Manager, Client Input Validator, Driver Manager, Google Maps' API

2.3.1.10 Notification Manager

Notification Manager	
Purpose	Sending notifications to users and forward proper informations to the other components
Tasks Performed	• Called by the Request Manager, creates threads to send notifications to driver users about ride requests, and waits for their answers (for 10 seconds). In case of acceptance, sets that driver's status in the Drivers Model on "unavailable", and then sets him as "not busy"; in case of refusal or in case the driver doesn't answer, just sets the driver on "not busy". In other words, it takes care of telling the system when a driver is no more in the task of answering a ride request, telling if he accepted it and is going to take care of it or if he refused it and thus has to be put back at the bottom of some city zone's queue by the Queue Manager • Called by the Request Manager, sends to a driver user a notification that he has been assigned to some reservation, updating the Drivers Model to set his status on "unavailable" and putting him back to "not busy" • Called by the Request Manager, sends to passenger users ride confirmation notifications, reservation reminder notifications and queued request notifications, waiting in this last case for the user to answer (for 30 seconds), and forwarding the answer to the Request Manager
Interacts With	Client Manager, Request Manager, Drivers Model

2.3.2 Data Tier

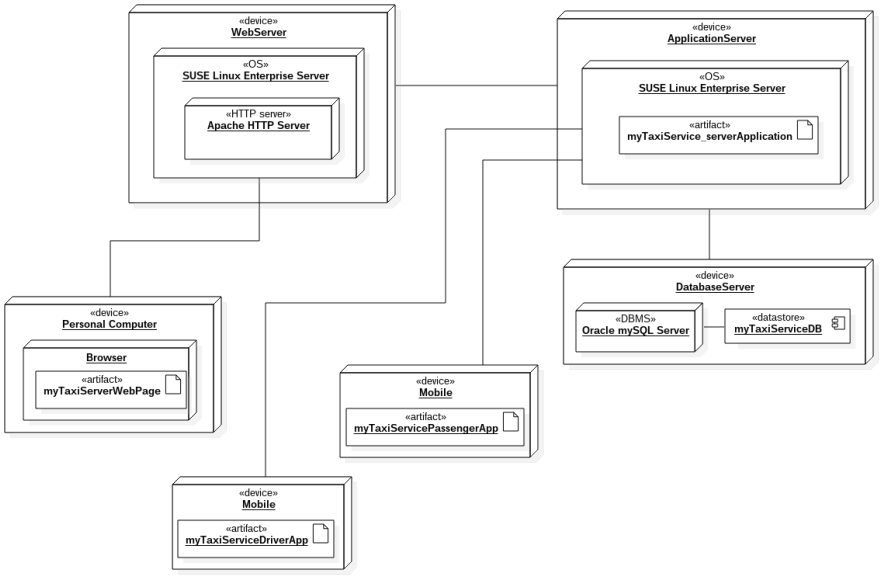
The Data tier of our system is composed by the DBMS and the database. The behavior of the DBMS is standard, it just interfaces the components of the Logic tier with the database, allowing correct creation, manipulation and querying of the data. To describe how the data are supposed to be organized in the database, we propose an ER diagram describing the most meaningful entities that are going to be stored in it:



The entities are basically the same that were in the brief class diagram presented in the RASD of course; but differently from there, the Taxi and CityZone entities are not involved in any relation with other entities. This is because in the central database we store tables representing associations between entities only if they are kind of permanent: since a driver may drive a different taxicab every day, and since each driver is constantly moving from one city zone to another, it doesn't make sense to keep track of these temporary associations in the central database. It would be a useless waste of time to keep the tables updated every time that, for example, a driver moves from one city zone to another. So, as already described in the Logic Tier paragraph of this section, to keep track of these dynamic Driver-Taxicab-CityZone associations we use instead a series of Driver objects of the Driver Model, which are stored directly on the server and refer only to currently logged in driver users. Therefore, these temporary relations are kept updated on those Driver objects only, since they are needed by the server to properly perform his tasks, while in the central database we just store CityZones and Taxis as independent data, to be read or written if needed, but much less frequently.

2.4 Deployment view

Here we present a deployment diagram of the system, showing how the different logical tiers are supposed to be distributed among physical devices.

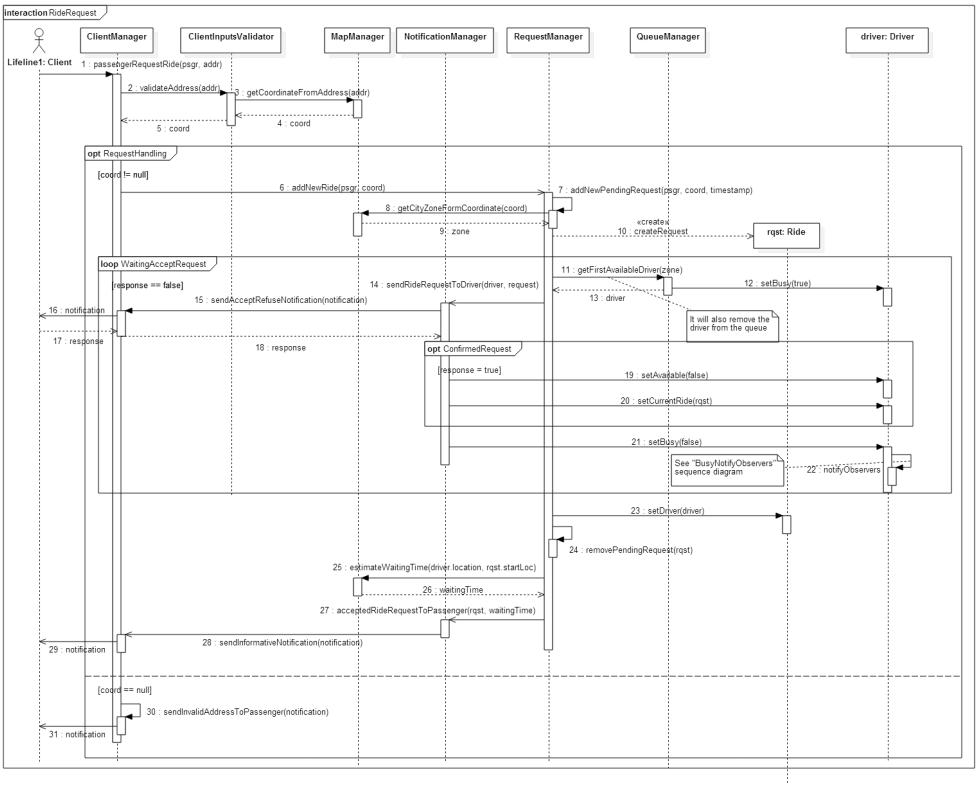


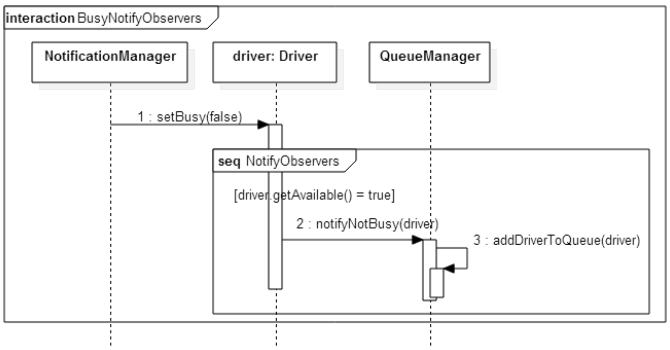
2.5 Runtime view

In this section we provide a series of sequence diagrams showing how the system handles the most common tasks it has to face at runtime, which allow to understand in detail how the interaction between the different tiers (and especially of the server's components) occurs.

2.5.1 Ride request handling

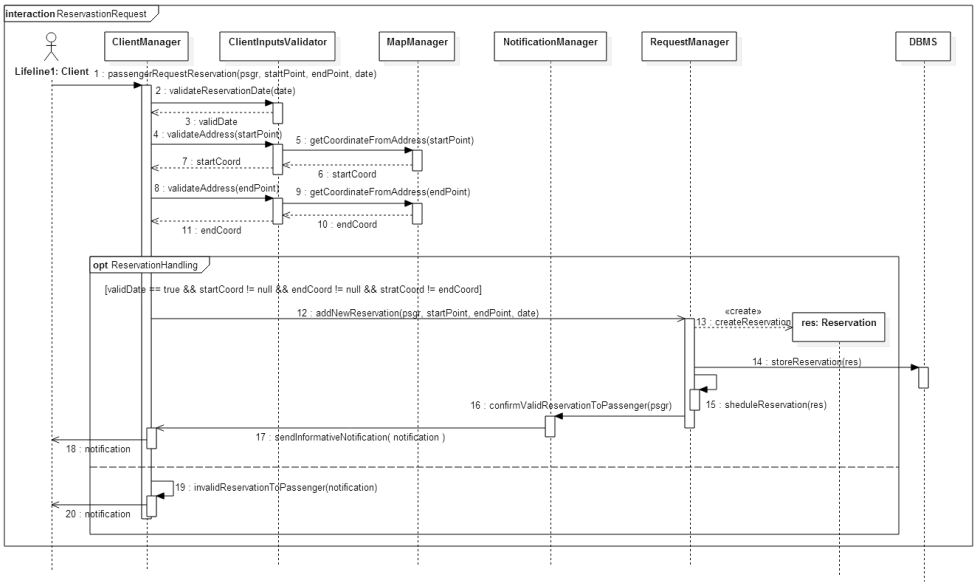
The following diagram shows how ride requests are handled in the most common case, i.e. we don't consider the case in which the ride request is queued because there was no available driver in that city zone, since it would be quite complex to represent in a unique sequence diagram; however, how that possibility is handled will be shown in section 3 "Algorithm Design".


interaction RideRequest
ClientManager
ClientInputsValidator
MapManager
NotificationManager
RequestManager
QueueManager
driver: Driver
Lifeline1: Client
1 : passengerRequestRide(psgr, addr)
2 : validateAddress(addr)
3 : getCoordinateFromAddress(addr)
4 : coord
5 : coord
opt RequestHandling
[coord != null]
6 : addNewRide(psgr, coord)
7 : addNewPendingRequest(psgr, coord, timestamp)
8 : getCityZoneFormCoordinate(coord)
9 : zone
«create»
10 : createRequest
rqst: Ride
loop WaitingAcceptRequest
11 : getFirstAvailableDriver(zone)
12 : setBusy(true)
13 : driver
[response == false]
14 : sendRideRequestToDriver(driver, request)
15 : sendAcceptRefuseNotification(notification)
16 : notification
17 : response
18 : response
It will also remove the driver from the queue
opt ConfirmedRequest
[response = true]
19 : setAvailable(false)
20 : setCurrentRide(rqst)
21 : setBusy(false)
See "BusyNotifyObservers" sequence diagram
22 : notifyObservers
23 : setDriver(driver)
24 : removePendingRequest(rqst)
25 : estimateWaitingTime(driver.location, rqst.startLoc)
26 : waitingTime
27 : acceptedRideRequestToPassenger(rqst, waitingTime)
28 : sendInformativeNotification(notification)
29 : notification
[coord == null]
30 : sendInvalidAddressToPassenger(notification)
31 : notification

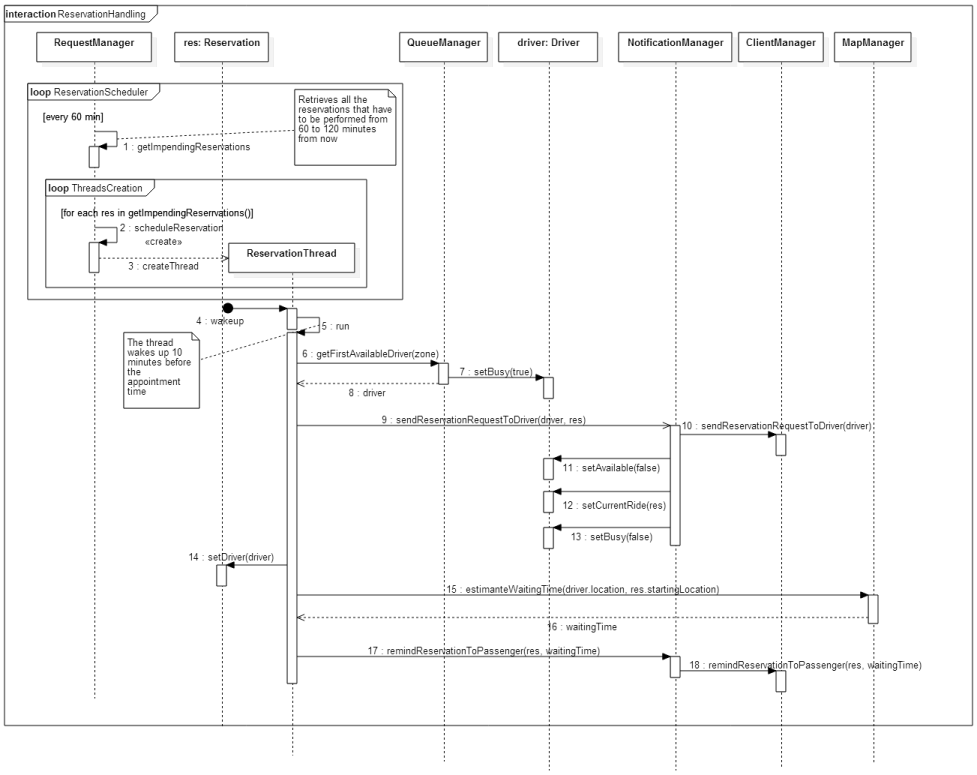

interaction BusyNotifyObservers
NotificationManager
driver: Driver
QueueManager
1 : setBusy(false)
seq NotifyObservers
[driver.getAvailable() = true]
2 : notifyNotBusy(driver)
3 : addDriverToQueue(driver)

2.5.2 Reservation request handling

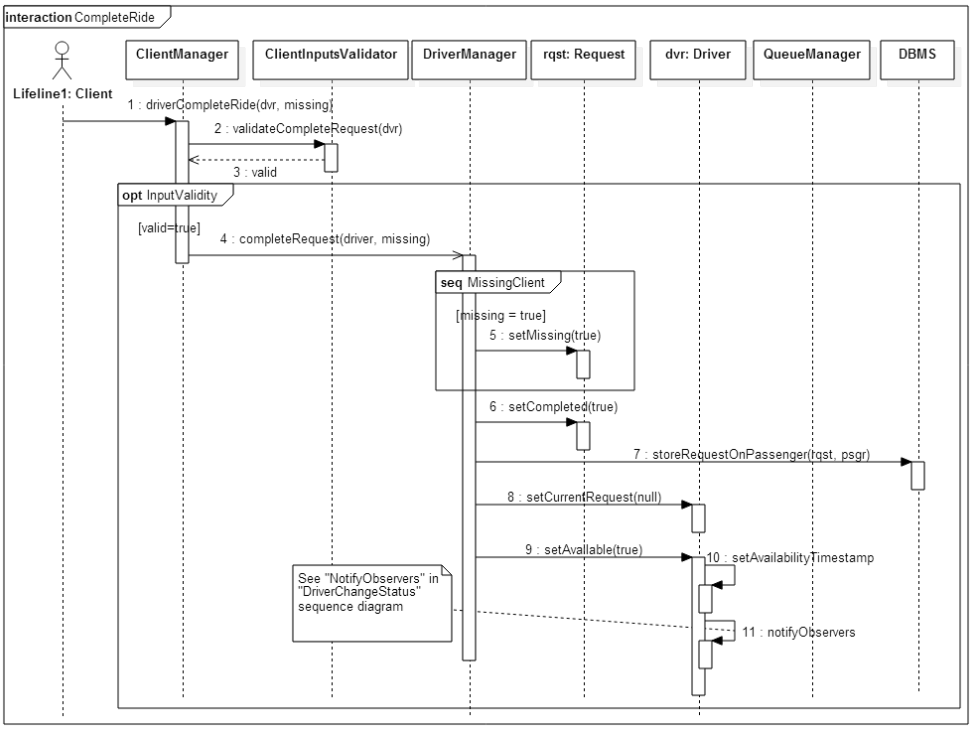
2.5.2.1 Reservation request acceptance



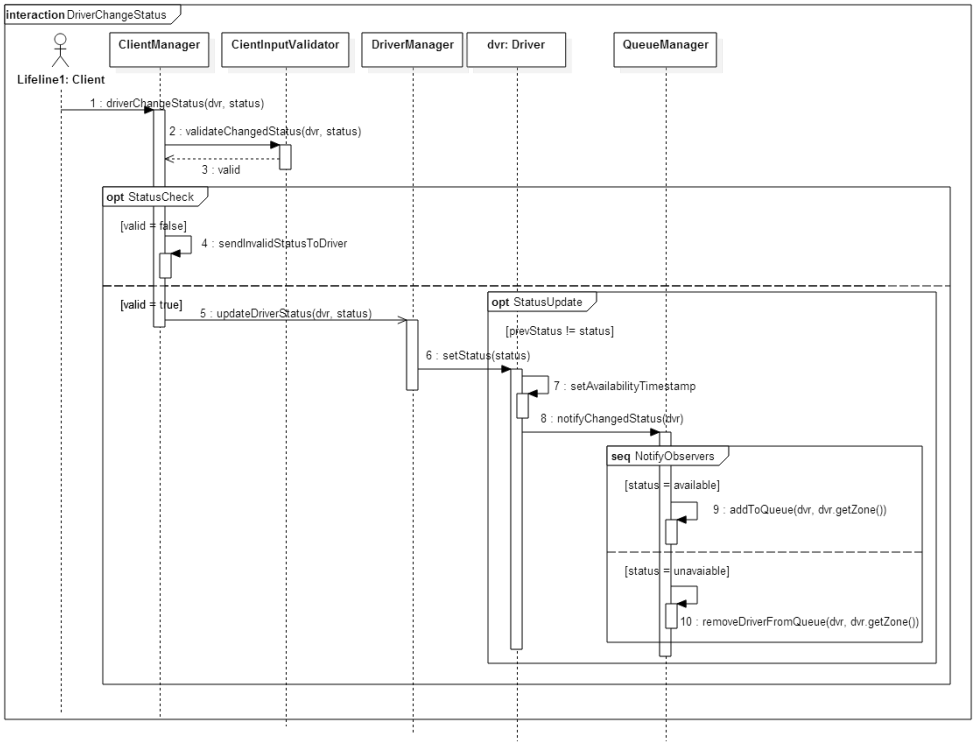
2.5.2.2 Scheduled reservation handling



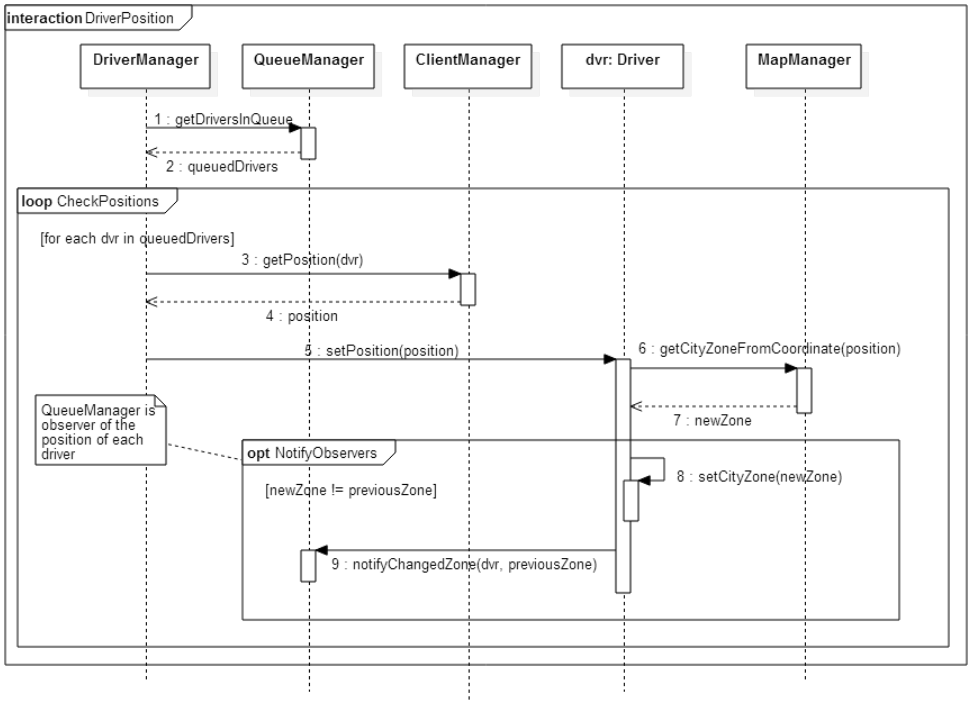
2.5.3 Ride completion



2.5.4 Driver's status change



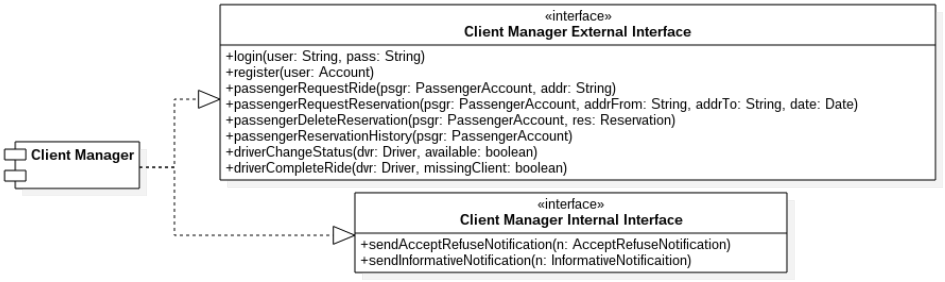
2.5.5 Available driver changing his position



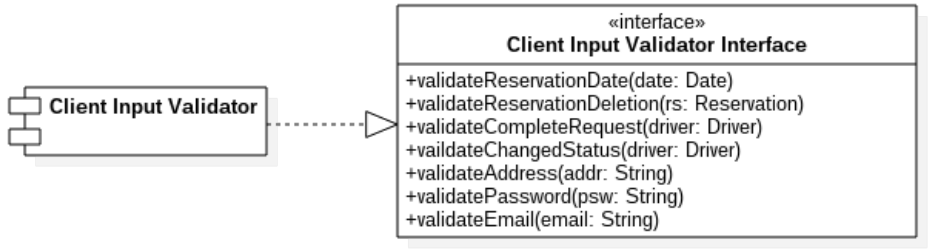
2.6 Component Interfaces

In this section we present the interfaces that are exposed by each of the server components, containing the functions that can be invoked by other components. The Client Manager exposes two different kinds of interface: one gathering the functions exposed internally, which can thus be invoked by the other server components, and the other one which is the one publicly exposed to the clients, so that they can send requests asking to perform the actions they need.

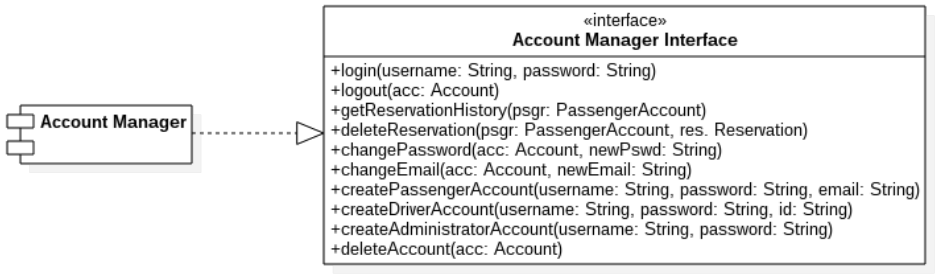
2.6.1 Client Manager interface



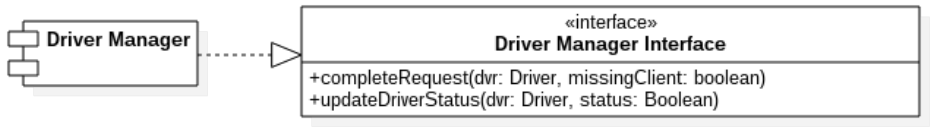
2.6.2 Client Input Validator interface



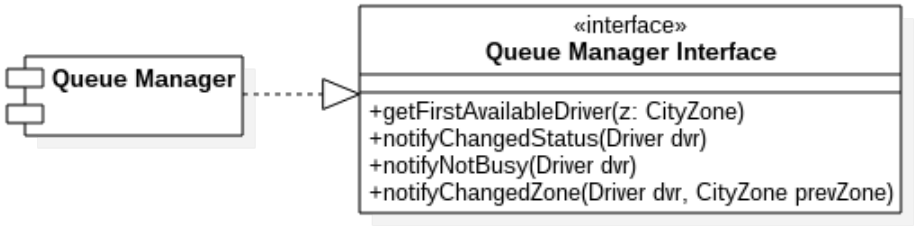
2.6.3 Account Manager interface



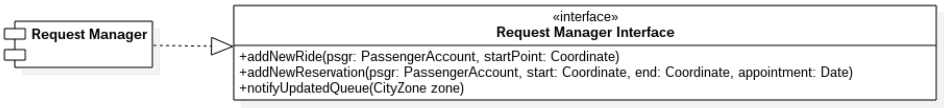
2.6.4 Driver Manager interface



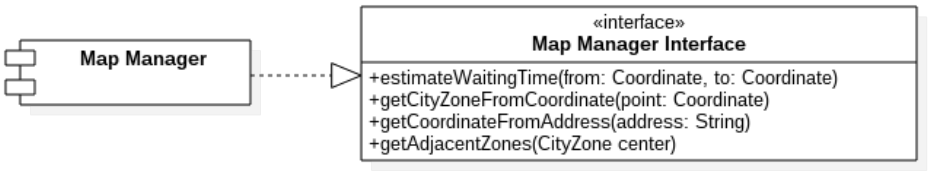
2.6.5 Queue Manager interface



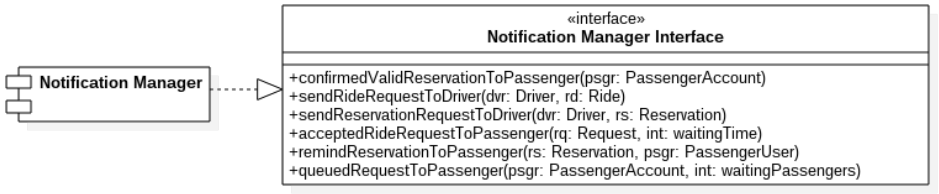
2.6.6 Request Manager interface



2.6.7 Map Manager Interface



2.6.8 Notification Manager interface



2.7 Selected architectural styles and patterns

Here we present a description of the most meaningful architectural styles and patterns we selected in designing the system, providing an explanation about why we chose such styles or patterns. Keep present that we are not going to list every single low level software pattern (like singleton, factory, ...) that shall

be used to implement the application: those are patterns that would obviously be used in any good software development environment, and therefore their description is not relevant in achieving the purpose of this document; so they won't be mentioned here, exception made for the ones that we consider particularly meaningful in understanding some design choices that have been made.

2.7.1 Client / Server style

In designing the high components of our system, we have decided to connect them using the client/server style, as it comes obvious by thinking at the intrinsic purpose of myTaxiService application. This style has been used to connect the web application with the Web Server and the mobile applications directly with the Application Server. The same architectural style has been used to connect the other high components of the system: the Web Server with the Application Server, in order to forward the requests and retrieve the responses, and the Application Server with the Database in order to store the persistent informations every time it is needed.

2.7.2 Model View Controller Pattern

In designing both mobile and web application we have decided to use the MVC (Model View Controller) pattern. The client will only have to render the View, allowing the user to interact with the system and showing informations provided by the Controller. The server acts as the Controller, receiving the requests from the clients and processing them, eventually updating the Model, and forwarding the response to the client. All the logic of the application is embedded in the server, so the client (thin client) will not perform any computation related to the functioning of the service, allowing myTaxiService to be available also on devices that don't have high computational power. We have decided to use this pattern in order to isolate each component of the system in his own independent domain, which makes the system much easier to maintain and debug. The service can be updated and modified without necessarily requiring changes to the end-user application.

2.7.3 Observer Pattern

To handle interactions among the internal components of the server, and in particular to easily spread any update that involves a change in the Drivers Model, we have decided to heavily rely on the Observer Pattern. In this way, it is easier to keep separate the roles and responsibilities of each component of the system, and to allow a correct application of the MVC pattern principles. To give an example, the Driver Manager component will only deal with keeping updated in the Drivers Model the position and the availability status of each driver, without having to directly notify the other components that something has been changed; being observers of the Driver objects in fact, they will automatically be notified about the update, and react consequently. In the same

way the Notification Manager component, after receiving by the driver the response to a proposed ride request, will only have to update the "busyness flag" of the driver without having to directly communicate it to other components. In general, in all the cases in which the Drivers Model gets updated, it will be the objects of the model itself to notify the Queue Manager (and possibly other components) about the change, and then the Queue Manager will react to the notified change keeping updated and consistent the drivers queue of each city zone, for example by moving the driver from a zone to another if the value of his 'CityZone' field has been changed, or adding or removing drivers from queues depending on their availability and busyness status.

3 Algorithm Design

In this section we are going to present the most meaningful algorithms that are used by some of the server components to handle the most tricky and nontrivial tasks involved in performing his duties. The algorithms will be presented in the form of pieces of java-like code, properly commented, to be easily understood and at the same time being quite precise, to leave no room for doubts on their behavior. Be aware that, in their actual implementations, the algorithms presented in this section will be implemented in a thread-safe way, while here this aspect of the implementation is not explicitly taken into account.

3.1 Drivers Queues Management

In this subsection we show pieces of code representing the algorithms used to manage the queue of drivers associated to each city zone. All the functions shown here belong to the Queue Manager component, which is the one that handles the task of managing those queues. To understand how they work, keep present that The Queue Manager component observes the Drivers Model using an Observer pattern (see section 2.7 for more detailed informations) and gets notified every time either the 'availability' or 'busyness' status of a driver is modified, and whenever the driver moves from a city zone to another. So, in general, the following algorithms show how the Queue Manager reacts when a Driver object of the Drivers Model notifies that one of his fields has been changed. Be also aware that, in the Queue Manager, the drivers queue of each city zone is supposed to be stored in a data structure that allows to be accessed also in arbitrary positions, so not only by pushing at the bottom and popping from the top of the queue; this is done to avoid time wasting in some specific situations, that will be explained further on.

3.1.1 The driver has changed his 'availability' status

This function is called by the Drivers Model every time the availability status of a driver changes. If the driver becomes "available", he has to be inserted in the queue of the city zone he is currently in; otherwise, if he has just switched to "unavailable", he has to be removed from that queue .

```
notifyChangedStatus( Driver driver ) {

    //Retrieves the queue in which the driver is currently in
    cityZoneQueue = getQueueFromCityZone( driver.getZone() );

    //Case 1: the driver has changed his status from unavailable to
        available.
    //He has to be put at the end of the queue of the city zone he is
        currently in
    if ( driver.getAvailable() == true) {
        cityZoneQueue.add( driver );
    }
    //Case 2: the driver has changed his status to unavailable.
    //He has to be removed from the queue of the city zone in which he is
        currently in
    else{
        cityZoneQueue.remove( driver );
    }
}
```

3.1.2 The driver has changed his 'busyness' status

This function is called when the 'busyness flag' of a Driver is set on 'false'. This means that the driver has just finished answering to a proposed ride request and so, if his status is still on "available" (which in fact means that he has refused the request), he has to be inserted back in the queue of the city zone he is currently in, from which he was popped out (and set on "busy") when he was forwarded the request.

```
notifyNotBusy( Driver driver ) {

    if( driver.getAvailable() = true && driver.getBusy() == false ) {
        //Retrieves the queue of the zone the driver is currently in
        cityZoneQueue = getQueueFromCityZone( driver.getZone() );

        //Adds the driver at the end of the queue
        cityZoneQueue.add( driver );
    }
}
```

3.1.3 The driver has moved from a city zone to another

This function is called by the Drivers Model every time in which, while updating the position of a driver, it detects that his current city zone has changed. The algorithm is designed to move the driver from the old city zone's queue to the new one, but only if he is currently "available", because otherwise the driver won't be present in any of the drivers queues, since they store only currently available drivers. Furthermore, to avoid a driver who had been "available" for a long time (and was now in the top positions of his zone's queue) to be put at the bottom of the new queue just because he moved to another zone, and thus to make him start back waiting to climb the queue for no reason, the algorithm is designed to insert him in the new city zone's queue not at the bottom, but according to his "availability timestamp", which tells us when was the last time he switched his status on "available"; in this way we avoid time wasting, inserting the driver in the queue as he had always been in that city zone since his last status switch to "available".

```
notifyChangedZone( Driver driver, CityZone previousZone ) {

    if( driver.getAvailabe() == true ) {
        //Removes the driver from the queue of the previous city zone
        previousQueue = getQueueFromCityZone( previousZone );
        previousQueue.remove( driver );

        newQueue = getQueueFromCityZone( driver.getZone() );
        //Finds the position 'i where the driver has to be placed
             according to the availability timestamp of the drivers
             already inside the queue, starting from the top of the queue
        int i = 0;
        while( i < newQueue.size() && newQueue.get(i).
             getAvailabilityTimestamp() < driver.getAvailabilityTimestamp
             () ) {
            i++;
        }

        //Adds the driver at the position 'i inside the queue
        newQueue.add( i, driver );
    }
}
```

3.2 Requests Management

In this subsection we show the algorithms used by the Request Manager component to handle ride and reservation requests, working together with the Notification Manager to forward those requests and their responses to the drivers and passengers involved.

3.2.1 Ride requests management

This function is called every time a new ride request is received by the system's server. The algorithm is designed to forward the request to the available drivers of the city zone the request is coming from, starting from the one on the top of the queue, until one accepts it. When this happens, the passenger's waiting time is estimated and a notification containing also the taxicab ID is sent to him. Otherwise, if there are no available drivers in the city zone where the request comes from, the request is enqueued and waits to be reprocessed as soon as a driver becomes available (in case of queues however, higher priority is given to reprocess enqueued reservations, if there are any (more details in subsection 3.2.3)).

```
handleRideRequest( Ride ride ){

    Driver driver;
    boolean driverResponse = false;

    while( driverResponse == false ) {
        //Retrieves the first available driver in the given zone
        driver = QueueManager.getFirstAvailableDriver( ride.getZone() );

        //If there are no available drivers, the function will return null
        if( driver == null )
            break;

        //Sends to the driver the request and waits for his response
```

```
        driverResponse = NotificationManager.sendRideRequestToDriver(
            driver, ride );
    }

    //Case 1: no available drivers were found
    if( driverResponse == false ){
        //Retrieves the number of passengers waiting a taxi to be
            available
        int waitingPsgrs = getWaitingPassengerCount();

      //Asks the passenger if he wants to wait until a new taxi becomes
          available, communicating the number of passenger already in
          queue
        psgrResponse = NotificationManager.queuedRequestToPassenger(ride.
            getPassenger(), waitingPsgrs);

        if( psgrResponse == true ) {
          //Enqueues the request in a waiting queue so that when the
              Request Manager is notified that a new driver has become
              available the request will be reprocessed
          addQueuedRideRequest( ride, ride.getZone() );
        }
    }

    //Case 2: a driver has accepted the request
    else{
        //Notifies the passenger
        int waitingTime = MapManager.estimateWaitingTime( driver.
            getPosition(), ride.getStartingPoint() );
        NotificationManager.acceptedRideRequestToPassenger(ride,
            waitingTime);
    }
}
```

3.2.2 Reservation requests management

This function is called on every reservation 10 minutes before that reserved ride has to be performed. The algorithm sends the request to the first available driver in the queue of the city zone where the ride should start from, and, if there are no available drivers there, to the first available driver in any of the adjacent zones. If still no available driver has been found, the reservation is enqueued and it will be reprocessed as soon as a driver becomes available (see subsection 3.2.3). However, keep present that this is very unlikely to happen.

```
handleReservationRequest( Reservation res ){

    Driver driver;
    //Retrieves the first available driver in the given zone
    driver = QueueManager.getFirstAvailableDriver( res.getZone() );

    //Case 1.a: there are no available drivers in that zone (the function
        returns null)
    if( driver == null ){
        adjacentZones = MapManager.getAdjacentZones( res.getZone() );

        for( int i = 0; driver == null && i < adjacentZones.size(); i++ ){
            driver = QueueManager.getFirstAvailableDriver( adjacentZones.
                get(i) );
        }
    }

    //Case 1.b: after checking adjacent zones, there are still no
        available drivers
```

```
    if( driver == null ){
        //Enqueues the request in a waiting queue so that when the
            RequestManager is notified that a new driver has become
            available the request will be reprocessed
        addQueuedReservationRequest( res, res.getZone() );
    }

    //Case 2: An available driver has been found
    if( driver != null ){
        //Sends to the driver the request
        NotificationManager.sendReservationRequestToDriver(driver, res);

        //Notifies the passenger
        int waitingTime = MapManager.estimateWaitingTime( driver.
            getPosition(), res.getStartingPoint() );
        NotificationManager.remindReservationToPassenger(res, waitingTime)
    }
}
```

3.2.3 Enqueued requests handling

This function is executed whenever a new driver becomes available in a certain city zone. Its role is to check and handle possible enqueued reserved rides first, or rides then, which refer to that city zone, before the new available driver is forwarded any other request. The algorithm checks, starting from the city zone in which the driver has become available and then checking the adjacent zones, if there is any enqueued reserved ride waiting for a driver to be available in those zones; if there is, it will forward the request to the this newly available driver. If there isn't any enqueued reservation, the algorithm will do the same for enqueued ride requests, but only if they refer exactly to the city zone where this new driver has just become available.

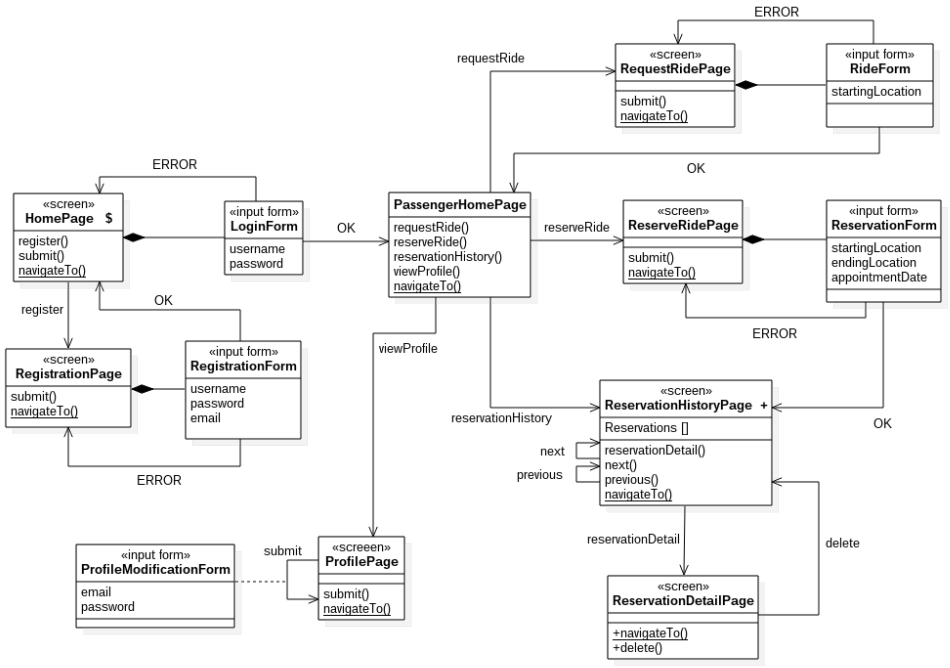
```
notifyUpdatedQueue( CityZone zone ){


    boolean driverAssigned = false;
    boolean driverResponse = false;
    adjacentZones = MapManager.getAdjacentZones( res.getZone() );

    //Preparing our city zones set, adding the zone where the driver has
        become available to the set containing also its adjacent ones
    adjacentZones.add(0, zone);

    //retrieving the newly available driver Driver
    driver = QueueManager.getFirstAvailableDriver( zone );

    //Case 1: check for enqueued reserved rides in the set of zones
    for( int i = 0; driverAssigned == false && i < adjacentZones.size();
        i++ ){
        //If there is at least one queued reservation waiting for a driver
            to be available
        if( getQueuedReservations( adjacentZones(i) ).size() > 0 ) {
            driverAssigned = true;

            Reservation res = getQueuedReservations( adjacentZones(i) ).
                getFirst();

            //Forwards to the driver the request
            NotificationManager.sendReservationRequestToDriver(driver, res)
                ;
            //Notifies the passenger
```

```
            int waitingTime = MapManager.estimateWaitingTime( driver.
                getPosition(), res.getStartingPoint() );
            NotificationManager.remindReservationToPassenger(res,
                waitingTime);
        }
    }

    //Case 2: check for enqueued ride requests in the central zone only
    //If there is at least one queued ride waiting a driver to be
        available
    if( driverAssigned == false && getQueuedRides( zone ).size() > 0 ) {
        Ride ride = getQueuedRides( zone ).getFirst();

        //Sends to the driver the request and waits for his response;
        //Same as for the usual ride request processing(this below would
            be a call to that same function, here we just duplicated the
            code to guarantee a better understanding)
        driverResponse = NotificationManager.sendRideRequestToDriver(
            driver, ride );

        //Case 1: the driver has refused the request
        if( driverResponse == false ){
            //Retrieves the number of passengers waiting a taxi to be
                available
            int waitingPsgrs = getWaitingPassengerCount();

          //Asks the passenger if he wants to wait until a new taxi
              becomes available, communicating the number of passenger
              already in queue
            psgrResponse = NotificationManager.queuedRequestToPassenger(
                ride.getPassenger(), waitingPsgrs);

            if( psgrResponse == true ) {
             //Enqueues the request in a waiting queue so that when the
                  Request Manager is notified that a new driver has become
                  available the request will be reprocessed
            addQueuedRideRequest( ride, ride.getZone() );
            }
        }
        //Case 2: the driver has accepted the request
        else{
            //Notifies the passenger
            int waitingTime = MapManager.estimateWaitingTime( driver.
                getPosition(), ride.getStartingPoint() );
            NotificationManager.acceptedRideRequestToPassenger(ride,
                waitingTime);
        }
    }
}
```

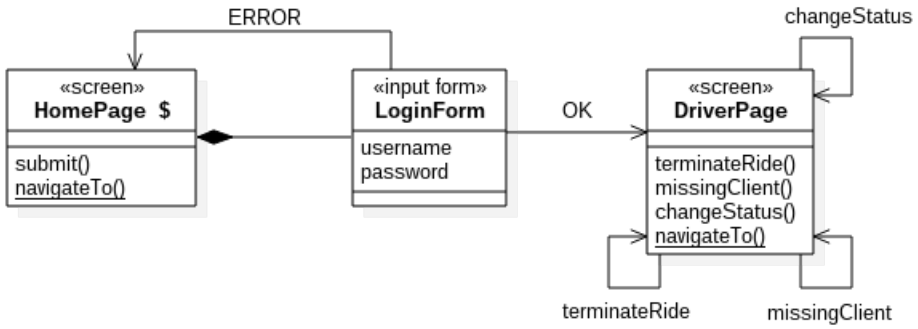
4 User Interface Design

In this section we present some User Experience diagrams that show which screens myTaxiService application should present to the user, and what kind of informations the user should provide in order to perform tasks on each of such screens. Please refer to the RASD document for a series of actual mockups of the most relevant screens of both driver and passenger applications.

4.1 Passenger user application interface



4.2 Driver user application interface



5 Requirements Traceability

Here we propose a requirements traceability matrix to show how we mapped the functional requirements listed in the RASD into the specific system components designed in this document. We chose to use this solution for the requirements traceability since it would have been chaotic and space consuming to report again all the functional requirements of the application, as we did in the RASD, and then discuss for each one of them which system components help fulfilling the requirement. Thus, we decided to just provide the traceability matrix, being aware that reading the description of the requirements in the RASD and the description of the components in this DD can easily allow to understand this matrix mapping proposed below. In reading the matrix, please refer to the legend to understand the components names' abbreviations, and please notice that for the functional requirements related to the administrator user (from [FR 17.1] to [FR 17.6]) we refer to the Passenger application (P) meaning the one accessible from web browser only, not the mobile one.

LEGEND:

P: Passenger application (either mobile or from web browser)

D: Driver application

CM: Client Manager

CIV: Client Input Validator

AM: Account Manager

RM: Request Manager

QM: Queue Manager

NM: Notification Manager

DM: Driver Manager

MM: Map Manager

DBMS: DBMS

	P	D	CM	CIV	AM	RM	QM	NM	DM	MM	DBMS
[FR 1.1]	X										
[FR 1.2]	X		X		X						
[FR 1.3]	X		X	X	X			X			X
[FR 1.4]	X				X			X			X
[FR 2.1]	X		X		X						X
[FR 2.2]	X										
[FR 2.3]	X		X		X						
[FR 2.4]	X										
[FR 3.1]	X										
[FR 3.2]	X		X	X		X	X	X	X	X	X
[FR 3.3]	X										
[FR 3.4]	X		X	X				X		X	
[FR 3.5]	X										
[FR 4.1]	X					X	X	X			
[FR 4.2]	X					X		X			
[FR 4.3]	X										
[FR 5.1]	X										
[FR 5.2]	X		X	X		X	X	X	X	X	X
[FR 5.3]	X										
[FR 5.4]	X		X	X				X		X	
[FR 6.1]	X					X		X			
[FR 7.1]	X					X	X	X	X	X	
[FR 7.2]	X										

	P	D	CM	CIV	AM	RM	QM	NM	DM	MM	DBMS
[FR 8.1]	X		X	X	X						X
[FR 8.2]	X										
[FR 8.3]	X										
[FR 8.4]	X										
[FR 9.1]	X		X	X	X	X					X
[FR 10.1]	X		X	X	X						X
[FR 10.2]	X		X	X	X						X
[FR 11.1]		X	X	X	X						X
[FR 12.1]		X	X	X					X		
[FR 12.2]		X									
[FR 13.1]		X	X	X		X		X			
[FR 14.1]		X	X	X		X		X			
[FR 14.2]		X					X	X	X		
[FR 14.3]		X				X		X			
[FR 15.1]		X				X	X	X			
[FR 15.2]		X	X			X		X			
[FR 16.1]		X									
[FR 16.2]		X	X	X			X		X		X
[FR 17.1]	X		X	X	X						X
[FR 17.2]	X		X	X	X						X
[FR 17.3]	X		X	X	X						X
[FR 17.4]	X		X	X	X						X
[FR 17.5]	X		X	X	X						X
[FR 17.6]	X		X	X	X						X

6 References

6.1 Softwares and tools used

- Google Docs (docs.google.com): to redact the document
- Lyx (lyx.org): to format the document
- StarUML (staruml.io): to create Class Diagrams, Sequence Diagrams and UX Diagrams
- Draw.io (draw.io): to create ER diagram and Tier diagram

7 Appendix

7.1 Hours of work

Here is how long it took to redact this document:

- Matteo Bulloni: ~ 28 hours
- Marco Cannici: ~ 29 hours