Cascianelli Silvia

Baldi Roberto

DESIGN DOCUMENT

V 1.1

MyTaxiService

CONTENTS

1 INTRODUCTION

1.1 PURPOSE

This is a Design Document. It aims at providing a description concerning the design of a software system in such a way that the software development team could use it as overall outlines for the architecture of the software project. Therefore, the Design Document must contain a complete functional description of the system, showing all the meaningful parts of the software and stressing how these latter ones will work and interact. In particular the document defines the major logical components of the proposed Software Architecture, shows their interactions and explores in deep all the internal and external interfaces among them; moreover it decomposes progressively the components in order to reach increasing levels of detail as far as the design choices are concerned. In this sense the Design Document makes use of all the different kind of UML diagrams to provide a full description of the system and must grant explanations regarding the adopted architectural styles and patterns. Besides of this, it offers also specifications concerning the permanent data management, the relevant algorithmic parts and it deepens some aspects of the user experience also in terms of user interfaces appearance. Finally, since it is also strictly related to the Requirement Analysis and Specification Document of the project, the Design Document must be coherent with the requirements listed on the RASD and must allow to trace them in the suggested design elements and solutions.

1.2 SCOPE

The project we will focus on is called MyTaxiService and it concerns the design of a system which is able to manage conveniently and efficiently the taxi service of a large city. Hence this project aims principally at simplifying the access of passengers to the service by exploiting the Net, but it also guarantees a fair management of taxi queues. To be more precise, MyTaxiService grants a potential passenger the possibility of requesting a taxy either through a

web application or a mobile one and it assures the applicant to be informed about the code of the incoming taxi and the waiting time, provided that the system succeeds in allocating a taxi for the request. Moreover MyTaxiService provides also the opportunity to demand for a reservation, as long as the booking occurs at least two hours before the expected ride; in this case the passenger must specify origin and destination of the ride as well as the meeting time during the booking process. In this way, the system will proceeds automatically to allocate a taxi for the reservation 10 minutes before the scheduled time; at that point, the applicant receives a confirmation with the code of the incoming taxi and any information concerning an eventual delay. However, if there are not available taxis the allocation procedure fails also for an already booked reservation; in this eventuality the negative outcome associated with the reservation is reported to the applicant through an email or a notification on the app, to apologize for the disruption. Hence, in general, MyTaxiService works in order to handle efficiently the incoming taxi requests. Presenting and supporting the design choices and solutions that allow the system to manage all these operations is the principle objective of this Design Document. In this sense it must be useful to remember also how the system should work, according with the functional requirements as well as the constraints and assumptions that have been already treated on the RASD of this project itself. In fact, as it has been already discussed on the RASD, essentially MyTaxiService arranges every available taxi in a specific queue associated with the zone of the city the taxi is located at that moment. Thus, the system can always forward an incoming request to the first taxy of the appropriate queue, according with the zone the origin of the ride belongs to and by using the special functionalities of the mobile app dedicated to drivers. Whether the chosen taxi driver does not accept the sent request MyTaxiService records a penalty for him and puts his taxi at the bottom of the local queue; at the same time the systems proceeds by forwarding the request to the following taxi, which has just became the first of the queue. If none among the first three chosen drivers accepts the request the system forces the fourth one with a mandatory service call. In fact MyTaxiService is designed to be efficient and so it cannot let a request missed unless there are not available taxi both in the local queue and in all the adjacent zones. In fact, if an incoming request refers to a zone in which the queue of available taxis is momentarily empty,

MyTaxiService provides a taxi by forwarding the request to the longer queue among the ones of the adjacent zones. To conclude this argument, it has to be noted that each queue is scrolled cyclically and so a driver could receive the same request more than once and finally should be forced to accept it. Finally, as far as taxi driver are concerned, each of them must use the mobile app, logging in at the beginning of each shift; actually through the app a driver can notify the availability every time a ride has been concluded and the system can place it in the opportune queue depending on its actual GPS position. Besides of this a taxi driver has to use the mobile app to accept an incoming call, which the system has conveniently sent to him, and also to signalize the occurrence of an urgent problem. Definitions, Acronyms and Abbreviations

1.3 DEFINITIONS AND ACRONYMS

Taxi driver: a kind of user that, through the mobile app, can access his/her private area and utilize special functionalities of MyTaxiService, usable only by this type of user.

Passenger: the other kind of user, that can utilize MyTaxiService both through the mobile app or the web application; in particular these users are interested in exploiting the application to look for a taxi easily and quickly.

Local queue: This term refers to the organization of the city in different zone to cover with the taxi service; in effect a local queue is supposed to be the ordered set of available taxi associated to a specific zone; in particular this relation is based on the GPS current position of each available taxi.

Request: This term indicates that a passenger is requiring a ride because he/she needs a lift at the current moment

Reservation: This term indicates a ride that a passenger want to book for a certain moment successive with respect to the sending time. Precisely the required time must be at least two hours later the booking.

Unsuccessful call: This terms indicates a call forwarded by the system to a chosen taxi driver, who simply does not answer and consequently does not accept the ride associated with that call.

- DD: Design Document.
- RASD: Requirements Analysis and Specification Document.
- DB: Data Base.
- DBMS: Data Base Management System.
- API: Application Programming Interface.
- JEE: Java Enterprise Edition.

1.4 REFERENCES

- Specification Document: MyTaxiService Project AA 2015-2016.pdf.
- Specification Document: Template for the Design Document AA 2015-2016.pdf.
- IEEE Std 1016tm-2009 Standard for information Tecnology-System Design-Software Design Descriptions.
- The Java EE 7 Tutorial 2013: Release 7 for Java EE Platform E39031-01

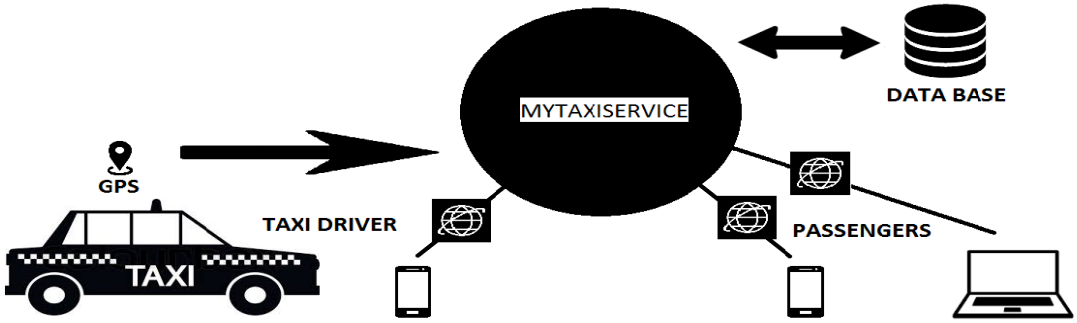
1.5 DOCUMENT STRUCTURE

The next chapter of this document, the Architectural Design section, gives a complete description of the software product by proceeding progressively from a higher level of representation of the component parts to a more detailed examination of each meaningful module. The following section, precisely, the Algorithm Design chapter is focused on dealing with all the relevant algorithms necessary for the system to work and that need to be conveniently explained. Then the User Interface Design chapter offers essentially some pictures representing the pages or the screens of the provided graphical user interface in order to show the appearance of the applications provided to users. Finally the last chapter, the Requirements Traceability section, clarifies and underlines the links among the requirements defined on the RASD and the characteristics and activities of the elements proposed in this DD.

2 ARCHITECTURAL DESIGN

This chapter will provide a series of different UML diagrams that aim at showing all the design elements of interest in our system, specifying all the interactions and relations to be highlighted and finally describing also the way components behave in order to fulfill a certain operation.

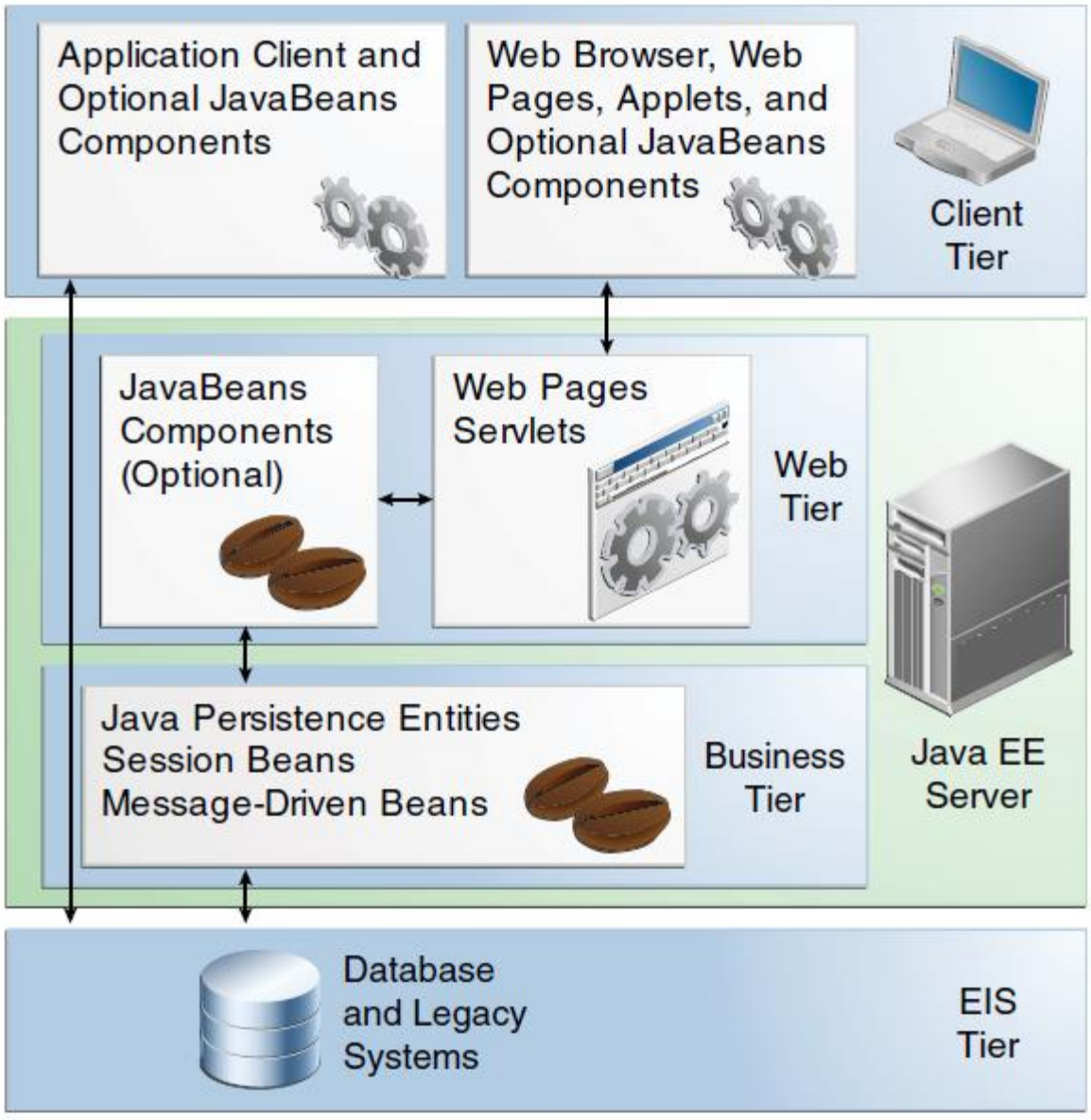
2.1 OVERVIEW



As we can clearly notice from this simple picture, the system to be designed must be able to interact through the Network with different kind of devices and users; besides of this it exploits information provided by the GPS tracker of each taxi as well as by the central Data Base containing all the meaningful data. Furthermore MyTaxiService should be designed and developed with a particular attention to maintain it open for future possible evolution or additions, either in terms of adopted solutions and dedicated hardware either as regards to the implementation of new functionalities. For all these reasons this kind of system can be conveniently realized based on a Java EE distributed multitier application model. In fact the Java EE platform for enterprise applications grants the opportunity of solving some issues of our project such as distributing service to multiple clients simultaneously, satisfying all the functional and also non-functional requirements of the system and extending the Java SE APIs

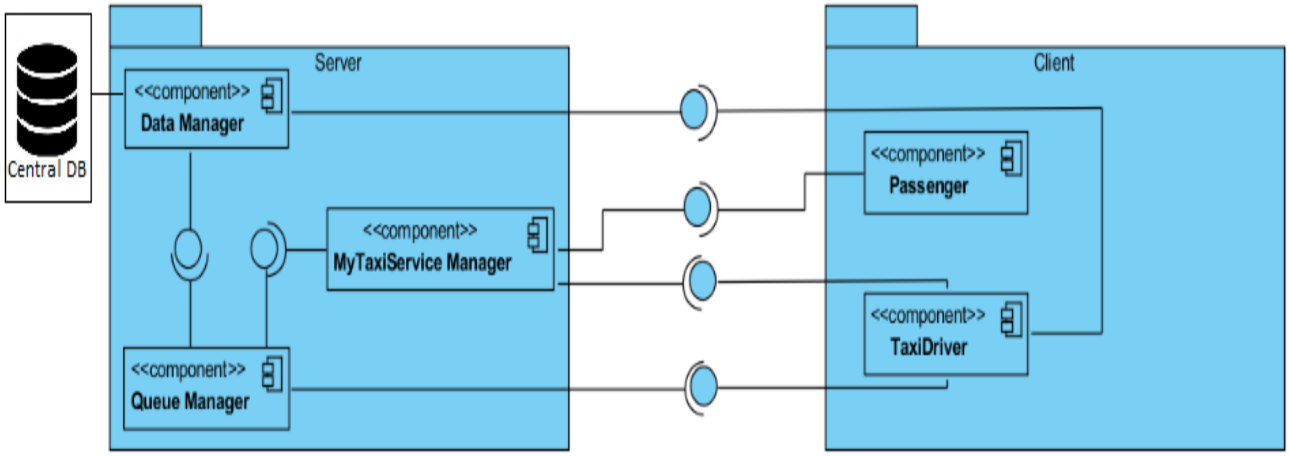
in order to reduce the development time and the application complexity as well as improving the portability, extensibility and interoperability of the software product.

The following picture shows the multi-tier architectural model of a general Java EE application:



The Client Tier can include Web Clients (also called thin clients), Application Clients and Applets (more complex clients) and also Web Browsers (which render the pages received from the Server). In general the clients communicate with the business tier running on the Java EE server either directly or, as in the case of a client running in a browser, by going through web pages or servlets running in the web tier. Anyway, in order to improve the reactivity of the system from the user point of view, the server and client tiers might also include Java Bean components that manage the data flow between the application clients or applets and the components on the Java EE server or eventually the flows from and towards the DB. Therefore, as we will see, we have intensively used these elements in the design we have intended. On the other hand the Web Tier contains the Servlets and the Web Pages that needs to be elaborated. This tier, actually, receives and dynamically processes the requests coming from the client tier and generates responses; however, to succeed in doing this, it must interact with the business tier, that implements the specific functionalities of the system in consideration. The Business Tier, in effect, includes different kind of Java Beans, able to perform the tasks required by clients according with the logic of the application; this tier contains also the Java Persistence Entities, which are classes representing directly the tables in the relational DBMS, thanks to the use of Java Persistence API implementations. We will see in the next paragraph how our project has been arranged in terms of macro-components, stressing the correspondence with the tiers of the Java EE model here presented. Finally we can consider the Enterprise Information System Tier, which contains the data source, and hence, in the case of our system, corresponds with the central database in which all the relevant data exploited by the application are stored.
To conclude we can highlight that Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the Java EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client-and-server model by placing a multithreaded application server between the client application and back-end storage. For this reason in the following schemas we have arranged the components of our architecture simply dividing the Client side from the Server side and focusing the attention mainly on a logical organization of the roles.

2.2 HIGH LEVEL COMPONENTS



The component diagram here above represents all the high level logical modules of the MyTaxiService software product, as well as the mutual interactions among these constituent parts.

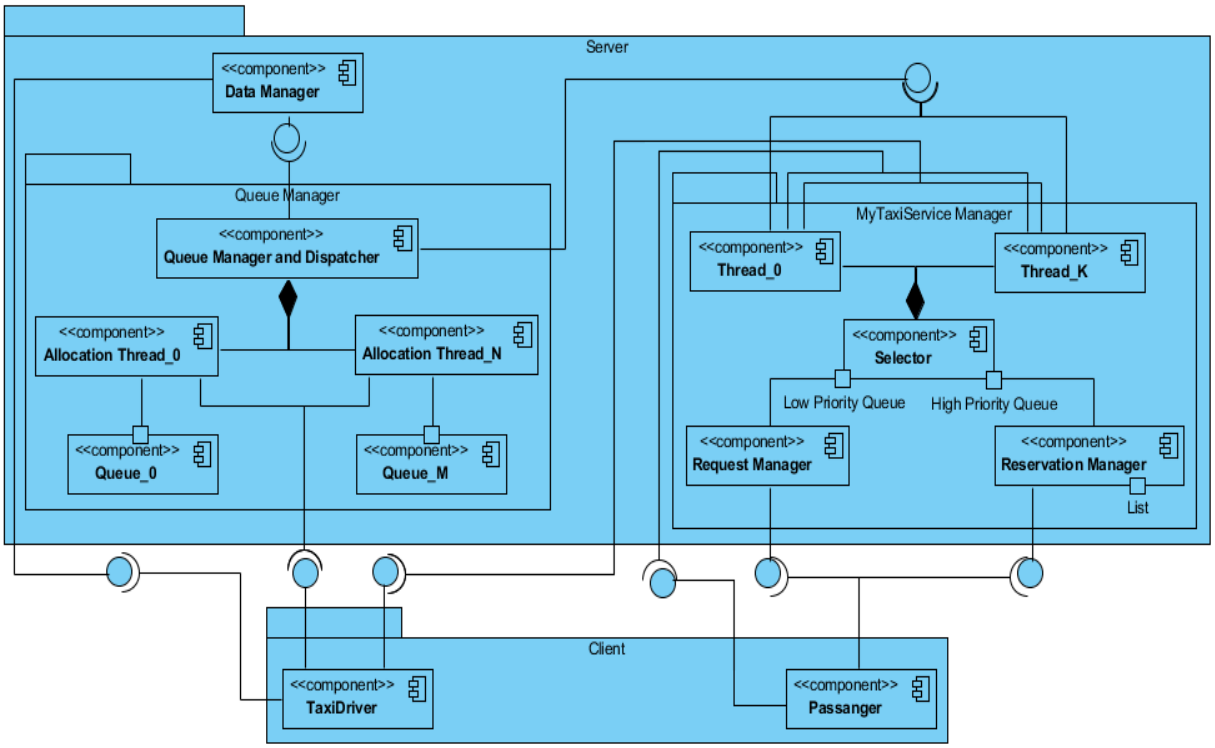
The client side groups the mobile and web applications addressed to the two different kind of users of such a system, respectively the taxi drivers and the so-called passenger. Since all the users of both categories must be granted the opportunity to exploit the special dedicated functionalities simultaneously and real time, the Server must manage all the services that the system want to provide users; it must be able to handle multiple clients and different types of operations at the same time and to succeed in providing the expected results real time and respecting the domain constraints. Hence the system has exactly all the characteristics and needs of an enterprise application and therefore, as we can easily notice from this schema, the suggested design for MyTaxiService project traces the tripartite organization of the JEE architecture model we have already discussed.

In particular, from the Client side perspective, the mobile app (addressed to both kind of users) could be an example of Application Client, with an appropriate graphical user interface; it offers

indeed a rich group of operations the client can require the system to perform and for this reason through the mobile app the user can interact usually directy with the other layers on the server side. Instead the web application, thought only for users of kind passenger, is supposed to be as a web page containing an embedded applet; in this case the link between the client and the functionalities offered on the server side is mediated by the used browser and the Java Server Faces with the related Servlets on the Web-Tier. Furthermore, since the software we want to design should guarantee the reactivity of the system from the user point of view, we have intended it uses session beans to manage the data flow between the client and the components on the Java EE server. In particular stateful session beans may be suitable to preserve an active interaction between each client, corresponding to any kind of user, and the tasks performed by components on the server side. (Please note that these stateful beans are managed by the EJB containers by alternating passivation and activation mechanisms to balance the resources but maintaining conversational states). However, at this point of our description we want to arrange the Server side from a logical point of view, by considering three macro-components called respectively MyTaxiService Manager, Queue Manager and Data Manager. The first one includes all the programs related with the creation and the supervision of taxi requests and reservations. So, in a lower level perspective, it exploits the Java Server Faces and the related Servlets that process dynamically all the operations requested by the clients, whereas from a logical point of view its behavior will be widely explained in the following paragraph. Anyway, this component offers more than an interface towards the Passenger Clients and, in addition, it interacts with the Taxi Driver Clients and with the Queue Manager in order to fulfil the management of taxi requests and reservations. On the other hand, the Queue Manager is supposed to be the handler for the allocation of all the taxis necessary to supply the incoming calls. In fact the role of this component is bivalent; it deals with the arrangement of the available cabs on the proper local queues but it is also responsible for the search of a taxi driver, who could take care of a certain ride. In this sense it is the core element of the Business Tier for what concerns the enterprise beans that implement the logic of the system and perform the required activities. So, in order to carry out its tasks, the Queue Manager uses directly an appropriate interface to the TaxiDriver Clients, in such a way that the

system could easily contact the taxi driver deemed suitable to accomplish a certain assignment. Furthermore, since our MyTaxiService Manager requires the intervention of this Queue Manager, this latter offers an interface to MyTaxiService Manager. Finally the Queue Manager needs to know all the information about the status of each taxi as well as the position of the available ones in order to update conveniently the local queues, therefore it must be related also with the Data Manager, the third macro-component of the Server side. This Data Manager is another constituent part of the Business Tier since it handles the access and the elaboration of all the meaningful data stored in the Data Base (which is our EIS-tier) and that different parts of the system needs to modify or to consult. So this last macro component includes all the Entities, actually the classes representing tables in the relational DBMS of our system. In fact we have supposed the Data Manager uses the Java Persistence API of JEE for mapping automatically the entities to relations in the relational DBMS of our central Data Base. Moreover the Data Manager is engaged also in computing all the information obtained by the GPS tracker of each cab in service, with a real-time control. Finally, besides of the link with the Queue Manager, the Data Manager offers also an interface to the TaxiDriver clients. In fact a client corresponding to a taxi driver user, (actually a driver during his working shift) can always set his state and hence modify the attribute state referring to him and stored on the DBMS; but to fulfil this change it is necessary the mediation of the Data Manager that controls and operates effectively on the data and relations of the DBMS.

2.3 COMPONENT VIEW



By analyzing in a more detailed way the component diagram of our system we can observe the role of certain sub-components and we can better understand the characteristics and the use of the interfaces. In particular in this first paragraph we will focus our attention on describing the elements that contribute on implementing the logic of such a system and we will delineate the interactions that connect the Client side to the Server one.

2.3.1 Logic of the System

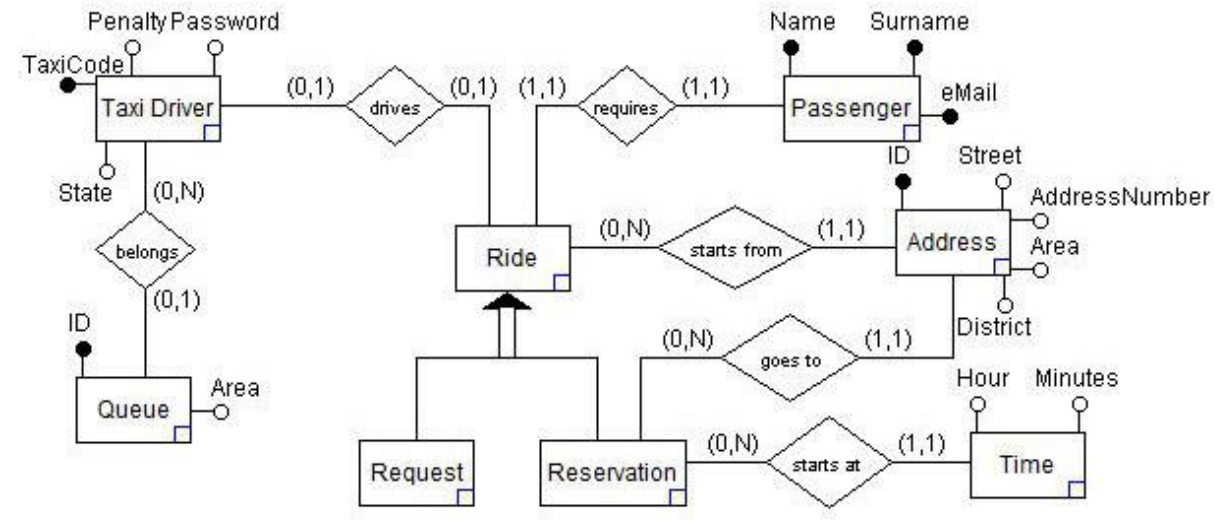
As we have already mentioned, the applications which run on the Client Side exploit some interfaces to the Server Side. In this sense we have supposed to embrace the Java EE remote connectivity model, which manages low-level communications between clients and enterprise beans in such a way that, after an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine. Therefore we have supposed to adopt some remote method invocations, although there are no limitations concerning different choices in the implementation of the connections. Anyway, the taxi driver clients requires a connection with the Server as soons as he tries to login; the JEE Server accepts the connection and if the log in concludes successfully the conversational state with the taxi driver client corresponding to a driver in service is maintened by using a stateful session bean. On the other hand, through both the web application or the mobile app, an user of kind Passenger could require the system services of reservation, request or cancellation of a ride; hence also a connection between this client and the Server Side must be opened and maintened by instantiating a stateful session bean. At this point, the sub-components of the MyTaxiService component are responsible for achieving the tasks required by the client. In particular they are the Reservation Manager and the Request Manager, that, as the names suggest, handles the creation and eventually cancellation respectively of a reservation or of a request. To be more precise, as we can see from the diagram above, the Request Manager creates a new request utilizing all the data received by the client, and push it in a so-called low priority queue. The same procedure is done by the Reservation Manager that pushes its instantiated reservation into the high-priority queue. Then, the following step is performed by the Selector component that launches a thread for each incoming ride that needs to be allocated. In this sense this component exploits a timer (with a procedure that we will discuss in the algorithmic part) in order to select also the reservations, but only at the correct allocation time, planned for 10 minutes before the arranged time for the ride. At this stage, every running thread interacts with The Queue Manager and Dispatcher Component. Thanks to the interaction with the Data Manager, the Queue Manager and Dispatcher knows exactly the state of each taxi and the current position of the available ones and so it can update conveniently the so-called local queues ,which are data

structures containing an ordered queue of taxi driver entities. More specifically, the queue manager and dispatcher instantiates and starts a new so-called allocation thread for each desired ride, in such a way that the system could improve the parallelism and also handle the concurrency. First of all the allocation thread finds the right local queue in which looking for a taxi driver, obviously based on which zone of the city the starting place of the ride belongs to. Futhermore, if the local queue of that area is temporarily empty it finds the longer local queue among the adjacent ones to fulfil the allocation of the ride; if and only if all the adjacent ones are empty the same, the thread ends informing immediately the waiting thread about the negative outcome. Hence, in all the other cases, the allocation thread tries to access the associated local queue through an exclusive lock mechanism in order to be able to implement the algorithm for searching a taxi driver. In this way the rides that refer to the same local queue can be allocated one after the other while allocations related to other local queues execute simultaneously. We will discuss in deeper how the allocation is implemented in the chapter dedicated to the algorithms. Anyway, each allocation thread uses the opened connections to the Taxi Drivers Clients, in order to contact the available ones belonging to the proper local queue and succeed in assigning the ride. As soon as the allocation thread finds a taxi driver who takes care of the ride, it returns all the meaningful information to the waiting thread which have been instantiated by the selector. Hence in the end, this thread uses the connection with the client to transfer the information about the incoming cab to the final passenger user. This latter has also the oppportunity to call off the just allocated ride as soon as he receives the confirmation of a positive outcome, exploiting the open connection and once again the thread related to that ride. In this case, the thread exploits the connection with the involved taxi driver client to inform him about the suppression of the ride. Anyway all the interactions and procedures we have just described, will be presented also using suitable sequence diagrams in the Runtime View paragraph.

2.3.2 Persistent Data Management

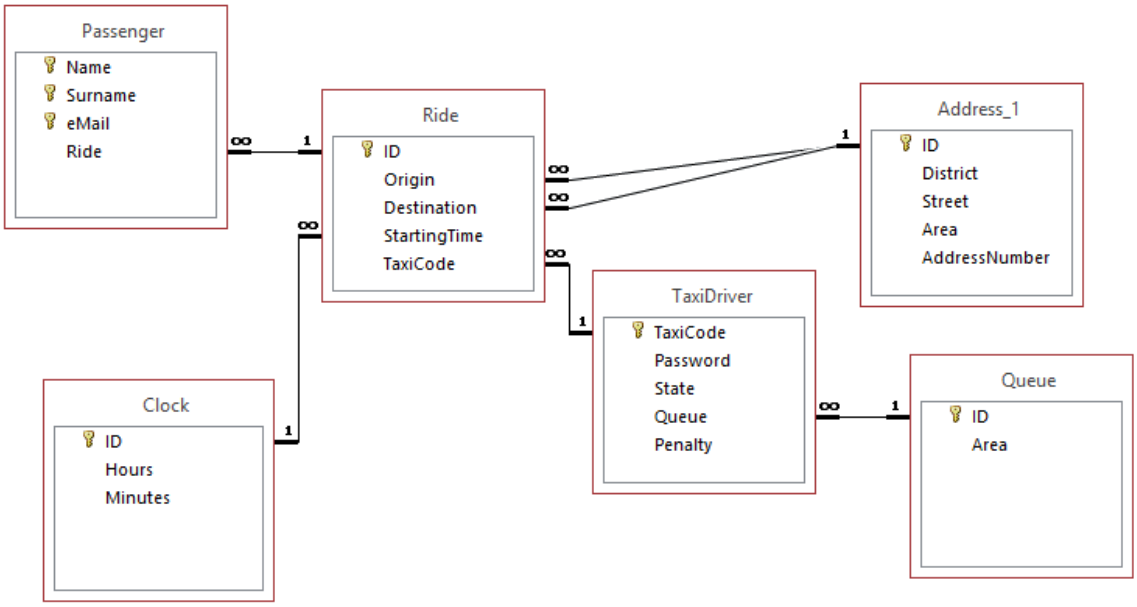
The meaningful data that our system deals with are stored into a relational database. The access and elaboration of this data is handled and mediated by the Data Manager component, which includes all the Entities of such a system. As we have already explained the relations are automatically materialized in the DBMS thanks to the Java Persistence API implementations exploited by the Data Manager and that map each entity to a table in the data base.

Therefore, in order to better understand the characteristics of all the relevant entities that the Data Manager of our system handles, in the following pictures we will present directly the corresponding design of our database in terms of Entity-Relationship Diagram and then we will provide also the derived Logical Model.



As we can see, for each relation we have conveniently reported the cardinality as well as for each Entity we have indicated the relevant attributes, stressing which are the identifiers.

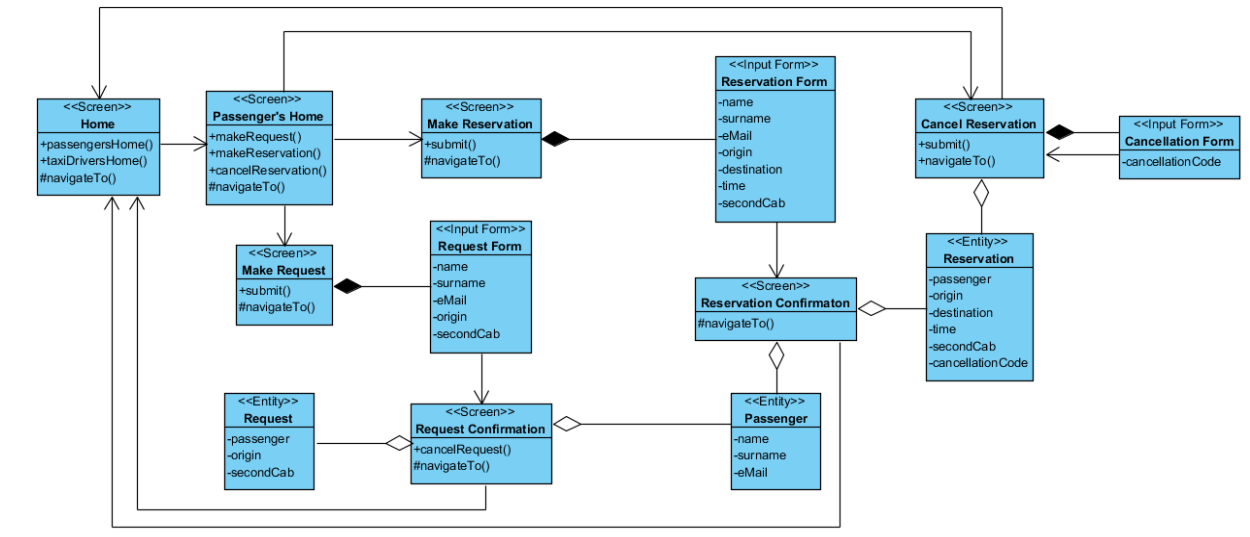
Finally the following scheme represents the relational logical design intended for the Data stored in the DB and derived from the previous conceptual model. The involved steps in such a transformation have included principally the elimination of hierarchical dependencies, the choice of primary keys for each relevant entity and the elimination of the relations by inserting foreign keys.



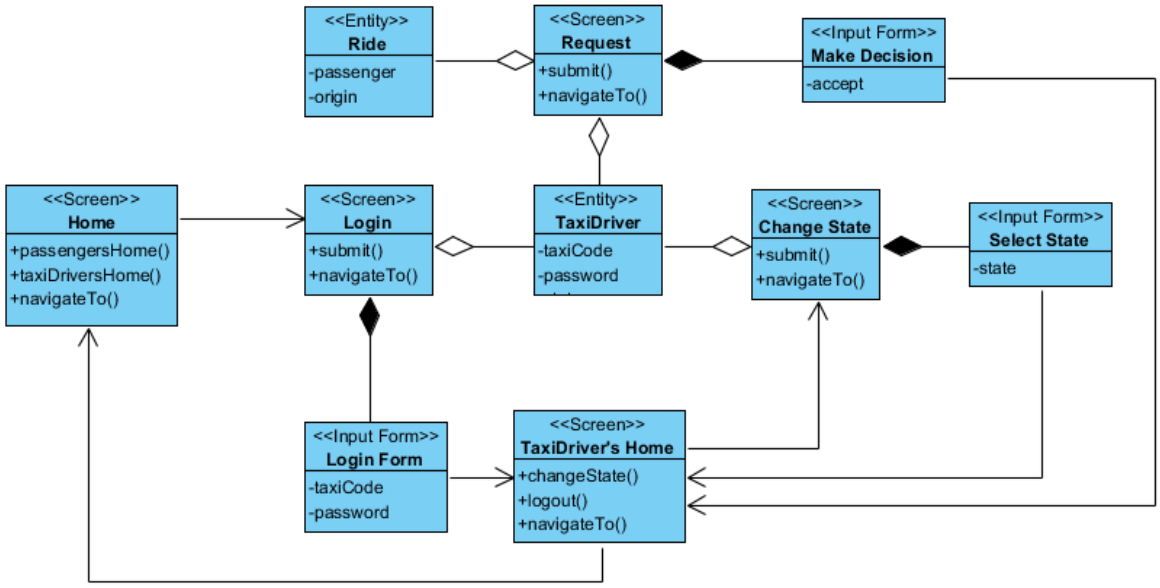
2.3.3 User Experience

In this paragraph first of all we will provide Class Diagrams to clarify how we have intended the User Experience specifically in terms of navigation experience, whereas we will focus more on the interaction between the human and the software product in the User Interface Design chapter. Hence, here we have obviously divided the navigation intended for the Taxi Driver from the one thought for the Passenger; anyway as far as these latter are concerned, we will present a general scheme referred to both the mobile and web applications. Hence, in the case of the mobile app we do not indicate specifically when the navigation could occur simply locally, since it will be quite obvious to understand at the development time. Therefore the stereotype <<screen>> has a general meaning of pages or mobile screens , whereas <<input form>> represents some input fields that can be filled out by a user (this information will be submitted to the system clicking on a button) and finally <<entity>> represents an object of a class derived from the model of the application.

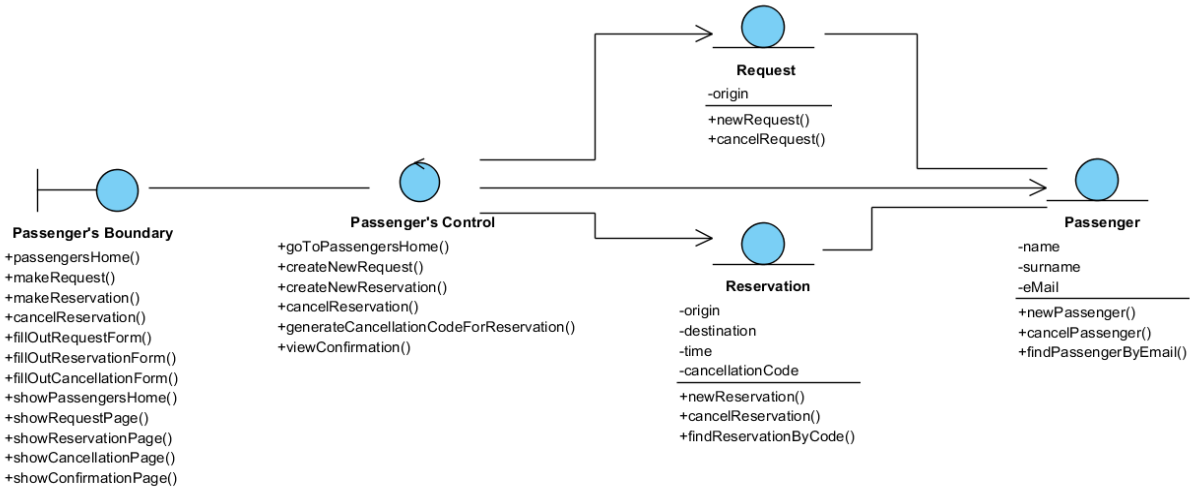
Passenger Perspective

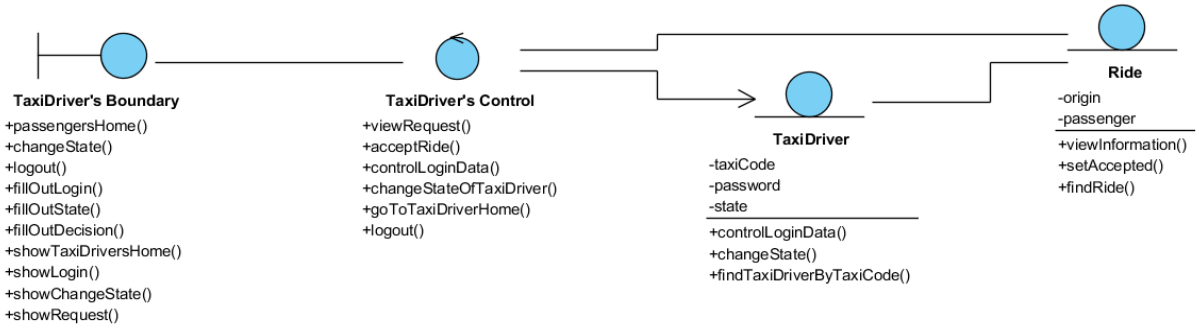


Taxi Driver Perspective



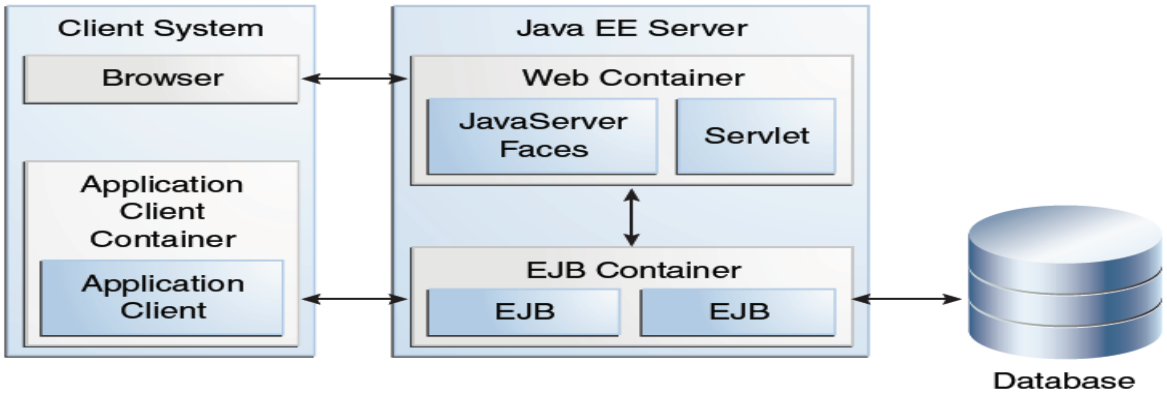
Finally, the application in its entirety must reflect the Architectural Pattern ECB: The Entity-Control-Boundary Pattern. It can be considered as an extension of the Model-View-Controller pattern because once again the Control decouples the external interfaces from the data. Anyway, although the Entity and Control parts are pretty the same as Model and Controller ones, the Boundaries are all the peripheral elements of the system or of its sub-system, not only specifically the graphic user interfaces. For instance, in the following schemes referring to the Server Side, although we have distinguished two diagrams according with the specific services and needs of the two kind of final users, the boundaries are all the external interfaces towards the client side. Anyway, obviously also the client side applications of our architecture are intended to be developed tracing the ECB architectural pattern.


Request
-origin
+newRequest()
+cancelRequest()
Passenger's Boundary
+passengersHome()
+makeRequest()
+makeReservation()
+cancelReservation()
+fillOutRequestForm()
+fillOutReservationForm()
+fillOutCancellationForm()
+showPassengersHome()
+showRequestPage()
+showReservationPage()
+showCancellationPage()
+showConfirmationPage()
Passenger's Control
+goToPassengersHome()
+createNewRequest()
+createNewReservation()
+cancelReservation()
+generateCancellationCodeForReservation()
+viewConfirmation()
Reservation
-origin
-destination
-time
-cancellationCode
+newReservation()
+cancelReservation()
+findReservationByCode()
Passenger
-name
-surname
-eMail
+newPassenger()
+cancelPassenger()
+findPassengerByEmail()


TaxiDriver's Boundary
+passengersHome()
+changeState()
+logout()
+fillOutLogin()
+fillOutState()
+fillOutDecision()
+showTaxiDriversHome()
+showLogin()
+showChangeState()
+showRequest()
TaxiDriver's Control
+viewRequest()
+acceptRide()
+controlLoginData()
+changeStateOfTaxiDriver()
+goToTaxiDriverHome()
+logout()
TaxiDriver
-taxiCode
-password
-state
+controlLoginData()
+changeState()
+findTaxiDriverByTaxiCode()
Ride
-origin
-passenger
+viewInformation()
+setAccepted()
+findRide()

2.4 DEPLOYMENT VIEW

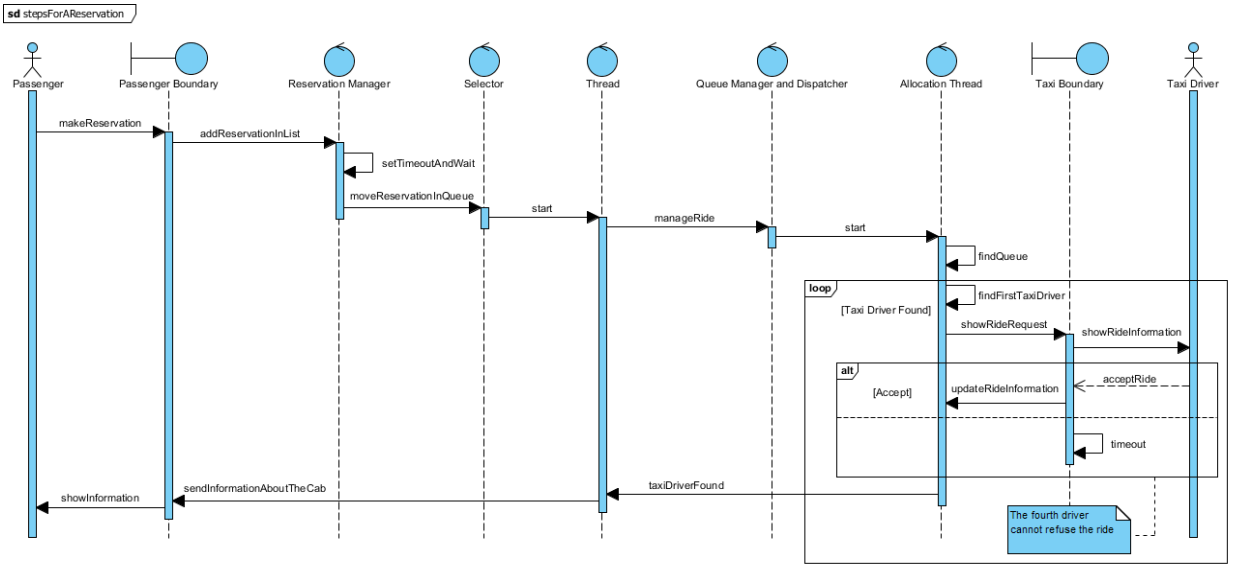
As we have specified also on the RASD, we have supposed that MyTaxiService can use Glass Fish, an open-source application server as a software framework which grants all the modules of our project the opportunity to run conveniently; actually, once this application server has been correctly started it provides all the low level functionalities that JEE offers. However, it is meaningful to take into account that all the modules of a JEE application must be encapsulated into containers that behave as interfaces in the interactions among components and between a component and low level functionalities offered by the platform. So, before it can be executed, a web, enterprise bean, or application client component must be assembled into a Java EE module and deployed into its container. Since we have already discussed the link between the elements of our design and the tiers of a JEE application we will simply offer a general picture representing the involved containers in which the modules must be deployed. Finally, we can consider that for each component the assembly process involves specifying container settings that customize the underlying support provided by the Java EE server including such services as security, remote connectivity, etc. For instance The Java EE security model lets you configure a component so that system resources are accessed only by authorized users (it could be useful to exploit for the taxi driver users) and, as we have already mentioned, the Java EE remote connectivity model allows clients to use remote method invocations. Anyway, to conclude we limit ourselves to mention that, in order to be deployed on the application server, a JEE application is conveniently distributed and packaged in an enterprise archive EAR containing all the modules with their containers and a suitable deployment descriptor.



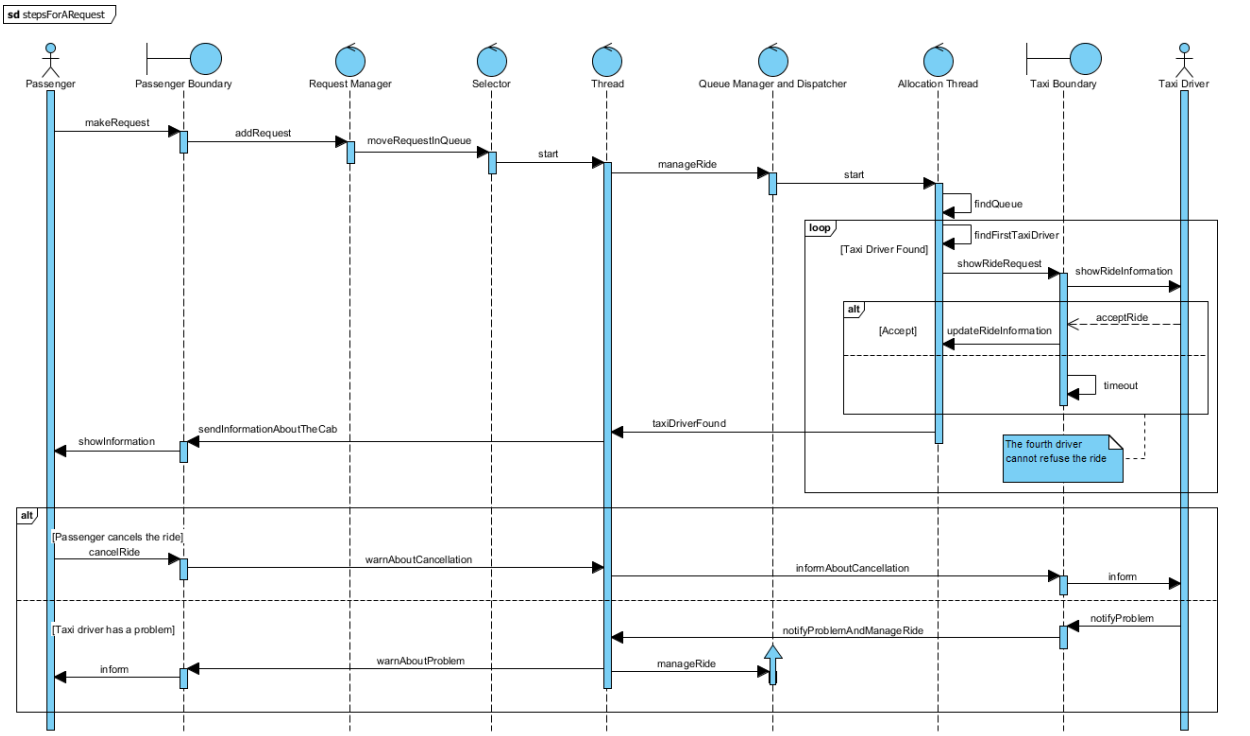
2.5 RUNTIME VIEW

In this paragraph we will provide several sequence diagrams that illustrate graphically and clearly the interactions among all the components of our architecture, and which are referred to the same relevant use cases we have already considered in the RASD. Since so far we have widely described these components and their roles, the following diagrams will be self-explanatory and certainly easily understandable. Therefore we will simply report the use case each diagram refers to.

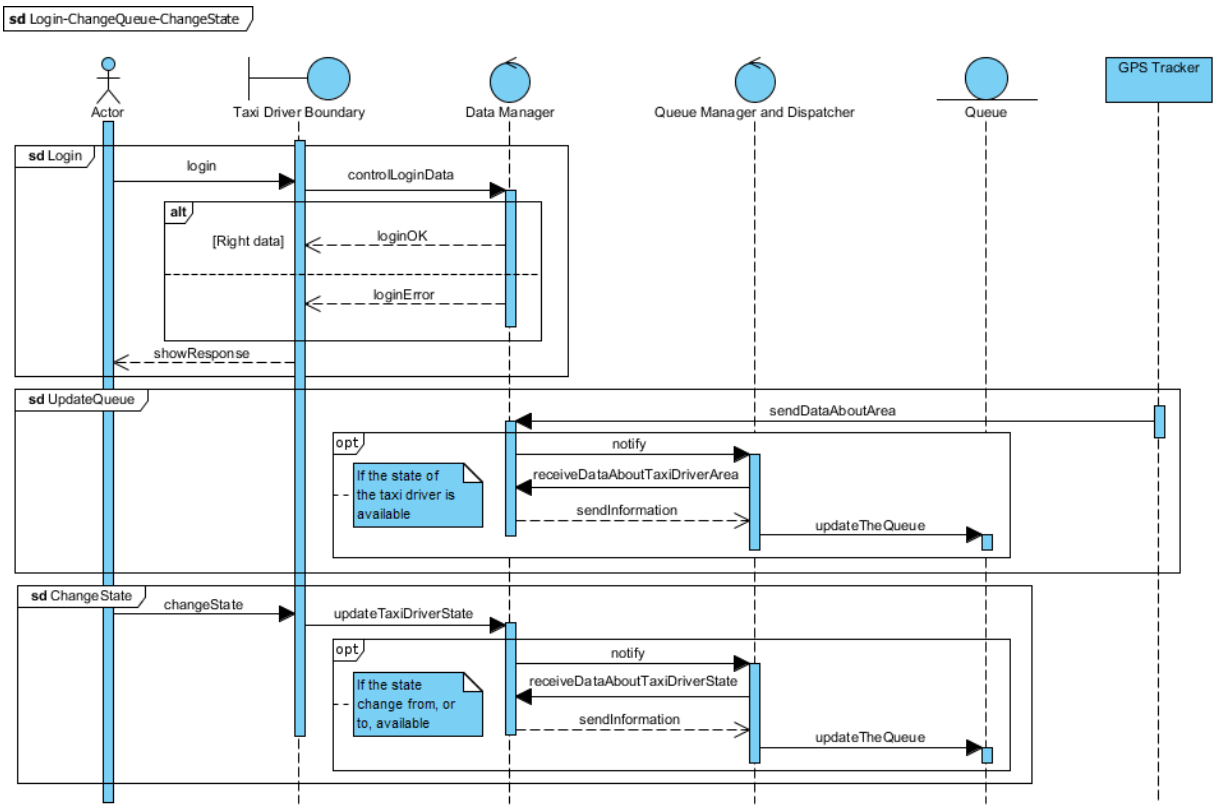
2.5.1 Passenger makes a taxi reservation



2.5.1 Passenger makes a taxi request



2.5.2 Taxi Driver performs some actions



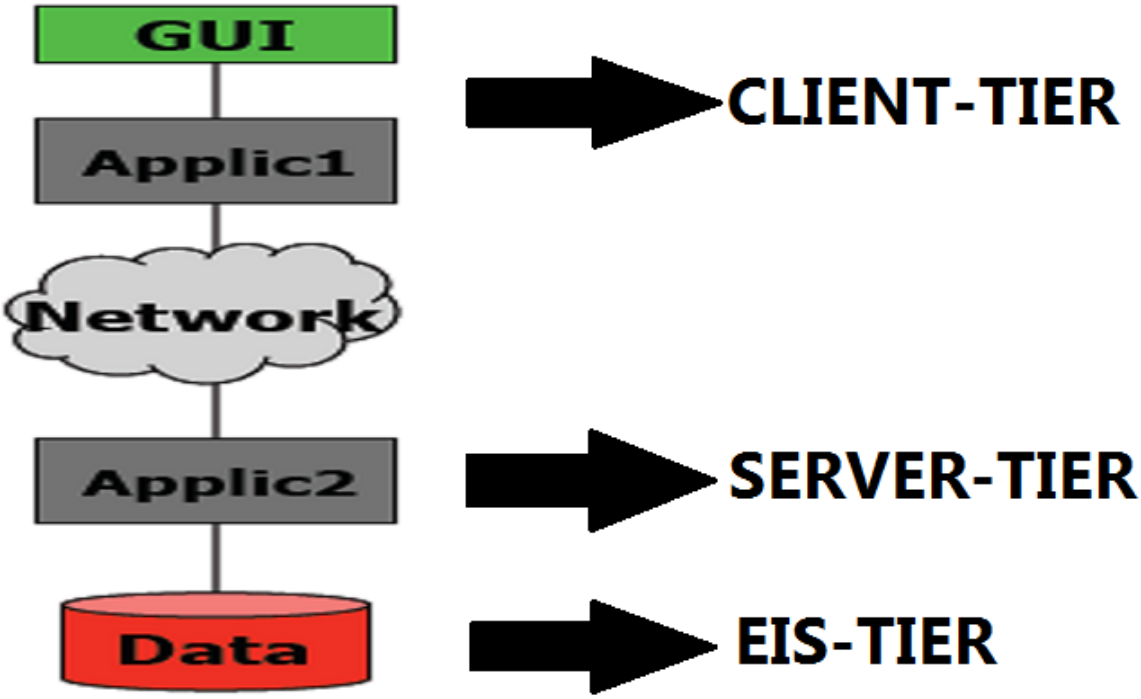
2.6 COMPONENT INTERFACES

In this paragraph we will discuss in a more detailed way how the interfaces among the components of the proposed design have been thought. First of all we consider the interfaces towards the clients of kind Passenger and offered respectively by the Request Manager and the Reservation Manager. We have supposed both are meant to be remote interfaces (implementable for instance by the class Passenger on the server side) that expose methods allowing the clients to require directly the desired services. In effect the client can forward the call of a remote method (such as makeRequest(…) makeReservation(…) cancelRequest(…) and so on) passing also all the necessary parameters (which must be Serializable) and the Request or Reservation Manager component can fulfil conveniently the required operation. A similar behavior occurs in the interactions between the Data Manager and a client corresponding to a Taxi Driver, hence, actually a mobile application which has successfully concludes the log in operation. In fact, also in this case, we have supposed the Taxi Driver-client exploits a remote interface exposed by the Data Manager for performing remote invocations such as calling a method to change the state associated with that driver. Another interface among components that runs on different devices is the interface granted by the Taxi-Driver to the allocations threads. This is responsible for forwarding an incoming call to the mobile app of a specific taxi driver, chosen for that ride, and returning the acceptance or not of the task (except for the mandatory assignments). Similarly the handler thread of a certain ride uses an interface for returning the applicant Passenger client with the outcome of the allocation procedure for that ride; in this way the Passenger has the opportunity to call of the just allocated ride or can be easily informed in case any problem occur. On the server side, instead, the interface between threads instantiated by the Selector and the Queue Manager and Dispatcher allows each thread to require the Queue Manager and Dispatcher to instantiate an allocation thread for assigning each ride. Finally both the interfaces between the Data Manager and the Queue Manager and Dispatcher as well as between the Taxi Driver client and the handler thread for a ride are interfaces that exploit the observer design pattern (although the last interaction must use also the network). To be more precise, the Taxi Driver component should consider an object

associated with the state of the driver as observable. Therefore, as soon as a driver accepts a ride the thread instantiated by the selector for handling that ride registers an observer to the state of the chosen taxi driver; if the driver uses the command Urgent problem and the state becomes suddenly “unusable” the thread is notified in such a way that it can inform the involved passenger and require a new allocation for the same ride. Instead, as far as the Data Manager is concerned, it is intended to have a Taxi Driver class which implements the observable interface. Then once again we have proper observers that are notified about any change of the state or of the geographical position of each taxi driver. Hence these observers in their turn can notify conveniently the Queue Manager and Dispatcher if there are changes involving available drivers in such a way that the local queue data structures could be always properly updated. Finally, on reverse, the Data Manager intervention can be required if it is necessary to add penalties to defaulting taxi drivers, updating the information stored on the DB.

2.7 Architectural Styles and Patterns

Since we have widely discussed the adopted patterns throughout the document, in this paragraph we will limit ourselves to offer a graphical representation that summarizes the three-tiered architecture with distributed logic that we have proposed as design choice for MyTaxiService.



3 ALGORITHM DESIGN

In this chapter we will focus on explaining some significant algorithms that our system must use to fulfil its services. In particular we will offer a brief description of each algorithm in natural language and then we will provide also pseudo-code as row version of the Java code suitable to implement the logic of that algorithm.

3.1 UPDATING THE POSITION OF A TAXI

The GPS tracker of each taxi is properly connected to the system and it can communicate the Data Manager its current position through an interface that is offered by this component. Hence, the application can control the position of each taxi and can update this information that is stored in the Database. On the other side the Queue Manager exploits the observers referring to the Taxi Driver objects in The Data Manager to be informed about the current position of each available taxi in order to append it to the proper local queue but also about any change of state that involves an available cab (either as start or ending state) in order to maintain the local queue data structures conveniently updated.

3.2 SELECTION OF THE RIDE TO MANAGE

Every time a passenger makes a request, a new request object is instantiated; this latter is directly appended at the bottom of a so called low-priority queue containing all the requests that are waiting for being forwarded by the ride-selector in order and managed by the allocation components.

Instead the reservations are processed in a different way. As soon as a passenger makes a reservation, a new reservation object is instantiated and put in an special list; actually it is an ordered list containing all the reservations that are conveniently sorted based on the time the desired ride should take place; in this way at the beginning of the list there is always the most immediate reservation (anyway, according with the constraints of our application, the reserved ride must be at least two hours later than the time the booking process occurs). Hence, thanks to this method of storage it is sufficient to maintain a timer that expires ten minutes before the arranged time for the reservation which is the head of list. Of course this timer must be correctly updated each time an incoming reservation becomes the new head of the list. With this technique, exactly ten minutes before the scheduled time, the reservation of interest is moved to a queue, in which it waits for being forwarded by the selector and then allocated; this waiting queue has a higher priority than the request waiting queue in order to fulfil the management of an already booked reservation according with the arranged time (except in the case of any eventual critical issue as for instance the lack of available taxis).

3.3 HANDLING OF A SELECTED RIDE

When a ride is selected by the ride Selector, it creates and launches a new thread to manage the ride. The thread sends all the information about the ride to the queue manager and dispatcher and then it waits for a response. The manager of the queues creates on its turn a new thread to handle all the allocation procedure for that specific ride. In fact this allocation thread chooses the right local queue in which searching a free taxi driver, according with the starting point of the ride. If the queue taken into consideration has at least a free cab the thread locks temporarily the queue and starts to forward an incoming call to the mobile app of the first taxi driver of that queue; the call is forwarded for 30 seconds and whether the driver accepts the search is concluded. Otherwise the thread puts this defaulting driver at the end of the local queue and also interacts with the Data Manager to add a penalty in that driver profile on the DB. Of course, in addition, the procedure is repeated with the new first driver, scanning

the queue cyclically. Nevertheless this iteration lasts for a maximum of 3 attempts; actually, in order to find a driver who accepts to take care of the ride, if three consecutive attempts for assigning the task fail, the fourth contacted taxi driver is forced to accept the ride with a mandatory assignment. Therefore, at the end the allocation thread returns all the information about the allocated ride to the other thread launched by the selector. In this way, it can communicate the passenger the code of the incoming cab and the waiting time; moreover if the passenger calls off the just allocated ride, the thread can notify directly the driver about the suppression of the assignment. To conclude we must also underline that the allocation method respects the outlines provided in the RASD; hence if there are no taxis in a local queue of interest to fulfil a ride, the allocation occurs referring to the longer local queue among the adjacent zones. Finally, if and only if all the adjacent zones are without taxis the outcome given back by the allocation procedure is negative.

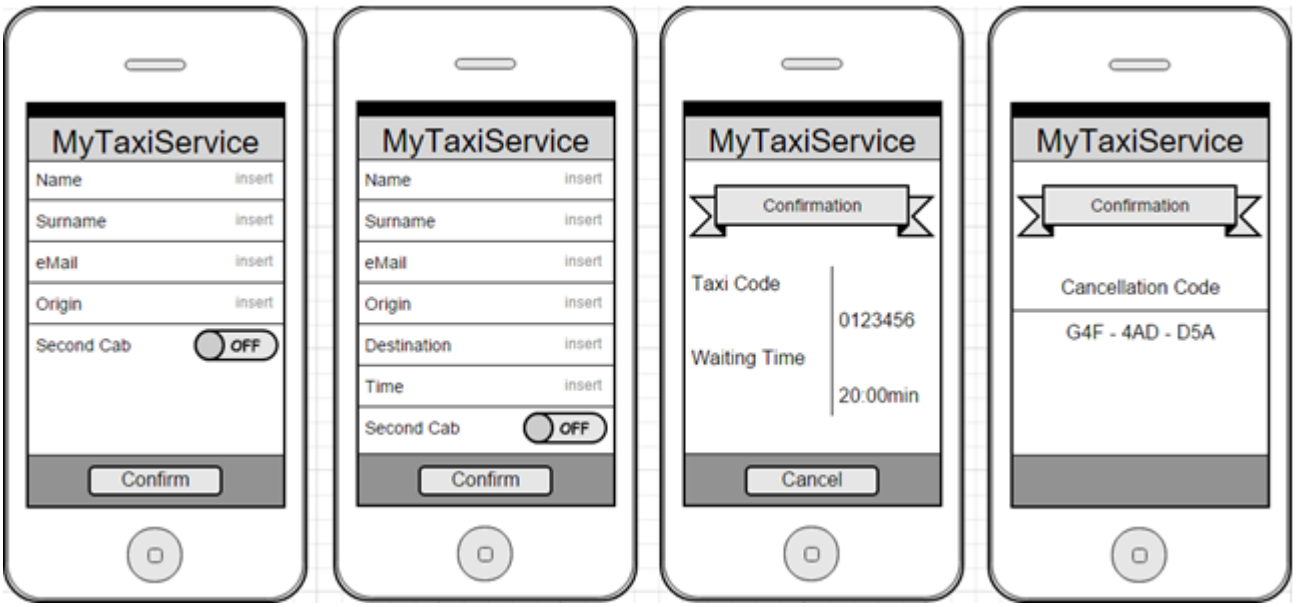
```
public class Selector {
    private QueueManagerAndDispatcher qmad;
    private Queue<Reservation> highPriority;
    private Queue<Request> lowPriority;
    private SortedList<Reservation> reservation;
    private Instant timeout = Instant.now();

    public void queuesManager() {
        while (true) {
            if (highPriority.size() == 0 && lowPriority.size() == 0)
                try {
                    this.wait(); // wait for a ride in some queue
                } catch (InterruptedException e) {
                }

            if (highPriority.size() != 0) {
                (new Thread(new ManagerOfARide(highPriority.pop()))).start();

            } else if (lowPriority.size() != 0) {
                (new Thread(new ManagerOfARide(lowPriority.pop()))).start();

            }
        }
    }

    public void listManager() {
        while (true) {

            if (timeout != null)
                if (Instant.now().isBefore(timeout) || reservation.size() == 0)
                    try {
                        this.wait(1000); // wait
                    } catch (InterruptedException e) {
                    }
                else {
                    highPriority.push(reservation.remove(0));
                    if (reservation.size() != 0)
                        timeout = reservation.get(0).getTime();
                    else
                        timeout = null;
                }

        }
    }
```

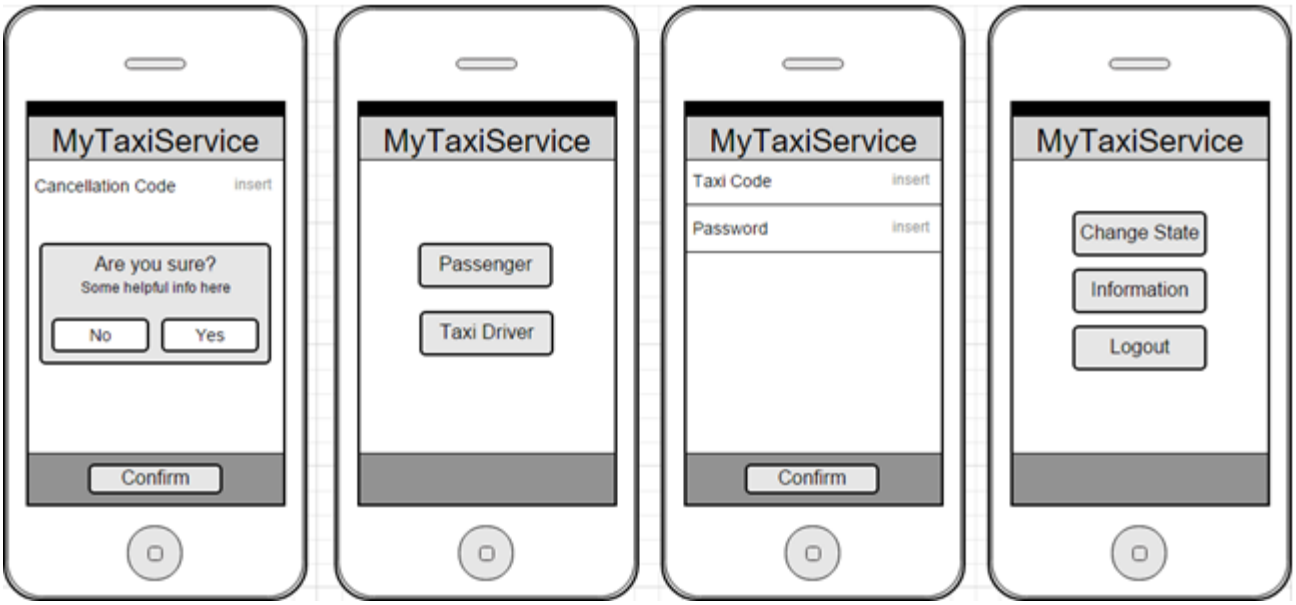
```java
public void addReservation(Reservation res) {
    reservation.sortedAdd(res);
    timeout = reservation.get(0).getTime(); // set the right timeOut
}

private class ManagerOfARide implements Runnable, Observer {
    private Ride ride;

    public ManagerOfARide(Ride r) {
        ride = r;
    }

    public void run() {
        qmad.receiveRide(ride);
        ride.getDriver().addObserver(this);
    }

    @Override
    public void update(Observable arg0, Object arg1) {
        String problem = (String) arg1;

        if (problem == "driversProblem")
            run();
    }

}
```

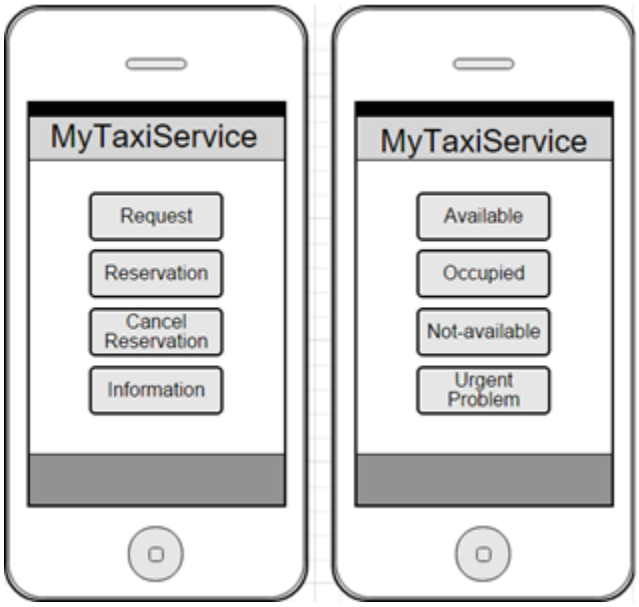
```java
public class QueueManagerAndDispatcher {


    public void receiveRide(Ride r) {
        new Thread(new RideManager(r)).start();
    }

    private class RideManager implements Runnable {
        private Ride r;
        private Queue<TaxiDriver> cabs;

        public RideManager(Ride r) {

        }

        @Override
        public void run() {

            int i = 0; // to know how many cabs have accepted the ride
            int n;
            do {
                do {
                    if (cabs != null)
                        cabs.unlock();
                    cabs = QueueManagerAndDispatcher.findQueue(r);
                    cabs.lock();
                } while (cabs.size() == 0); // to take a non-empty queue

                n = 1;
                TaxiDriver driver = cabs.pop();

                while (n < 4) {
                    if (driver.ask(r) == "accept")
                        break; // exits from this while
                    driver.addPenalty();
                    cabs.push(driver);
                    driver = cabs.pop();
                }

                driver.sendInformation(r);
                r.addDriver(driver);
                cabs.unlock();
                i++;
            } while (r.getNumberOfCabs() > i);
        }
```

4 User Interface Design

In this paragraph we want to illustrate the User Experience (UX) that our system grants its users in terms of how our graphical user interfaces will look like, respectively for the mobile app and for the web application. Therefore we simply offer some auto-explanatory mockups of the user interface appearance.




MyTaxiService
Cancellation Code
insert
Are you sure?
Some helpful info here
No
Yes
Confirm
MyTaxiService
Passenger
Taxi Driver
MyTaxiService
Taxi Code
insert
Password
insert
Confirm
MyTaxiService
Change State
Information
Logout


MyTaxiService
Request
Reservation
Cancel Reservation
Information
MyTaxiService
Available
Occupied
Not-available
Urgent Problem


Browser
MyTaxiService
Make Request
Make Reservation
Cancel Reservation


Browser
MyTaxiService
Make Request
Name
Surname
eMail
Origin
Second Cab
Go


Browser
MyTaxiService
Make Reservation
Name
Surname
eMail
Origin
Destination
Time
Second Cab
Go


Browser
MyTaxiService
Cancel Reservation
Reservation Code
Cancel

5 REQUIREMENTS TRACEABILITY

Referring to the Requirements we have presented on the RASD we can verify that the design we have proposed is coherent with these just mentioned functional and non-functional requirements.

- **[R1]** Both the web application and the mobile app must have user-friendly interfaces with intuitive commands, easily usable. Moreover, the applications aim at granting a trivial to use real-time connection between potential passengers, who want to enjoy the service, and taxi drivers, that provide it.

  [Both aspects have been conveniently fulfilled as we have extensively discussed in the Architectural Design section for what concerns the connections and in the User Interface Design chapter, where we have shown examples of easy-to-use graphical interfaces]
- **[R2]** MyTaxiService must interact with the GPS tracker of each cab in order to compute its exact position instant by instant.

  [We have supposed the Data Manager component of our Architecture is responsible for treating this kind of incoming information]
- **[R3]** MyTaxiService must interact with the DBMS of a data base containing all the relevant information about the taxis. The only mutable attributes about a taxi concern its state, its actual position and the amount of penalties; the queue are continuously updated depending on the available taxi in each corresponding zone

  [In this case we have shared the tasks between the Data Manager and the Queue Manager and Dispatcher. Since the relations in the DBMS correspond to entities of the Data Manager, this latter can access, modify or delete the data in the central DB with the mediation of the DBMS. Anyway, during this design phase, we have introduced

also the so-called Local Queues, which are meant to be as data structure managed and updated locally by the Queue Manager and Dispatcher]

- **[R4]** MyTaxiService considers the city as a grid of squared zone. Each zone encloses an area of 2km^2 and thus has a maximum of 8 adjacent zones. Every available taxi is appended by the system at the bottom of the local queue, depending on the GPS position of the taxi at that moment**.**

  [The Queue Manager and Dispatcher disposes a proper data structure for the local queue of each zone and knows the general trim of the zones and the concept of adjacency; furthermore this component exploits observers associated with the Taxi Drivers objects; so if the state of a Taxi Driver becomes available or the position of an available one changes, the Queue Manager and Dispatcher is informed and can update conveniently and efficiently its local Data Structures]
- **[R5]**. MyTaxiService must handle all the simultaneous requests and reservations of different passengers and must evaluate the addresses specified in each of them; in this way the system forwards a call to the proper cab, according with the local taxi queue suitable for accomplishing that assignment.

  [We have widely discussed about the stateful session bean associated with each client and also about how the logic of the system proceeds in order to instantiate a request or a reservation and to allocate a taxi for a ride]
- **[R6]**MyTaxiService starts a new taxi search for an already allocated call if the driver who has accepted to take care of that assignment notifies any kind of problem through the app.

  [Since in this case the state of that taxi driver becomes "unusable" the thread instantiated by the Selector specifically for that ride is informed; then it launches a new allocation thread, informs the passenger and finally provides this latter the outcome of the new allocation]
- **[R7]** MyTaxiService refuses an incoming request or fails the automatic taxi allocation associated with a previous reservation, if and only if both the local queue suitable for that ride and all the adjacent ones are empty at that moment.

[As we have already explained, if an allocation thread does not find a taxi for a ride in the corresponding area nor in all the adjacent ones, it ends returning the negative outcome to the waiting thread instantiated for that ride; this thread on its turn, exploits the connection with the client to inform this latter about the failure]

- **[R8]** Every time a taxi search regards a zone where the local queue is not empty the allocation must be achieved successfully, in the worst case through a mandatory assignment. This rule holds also if a ride has the origin that belongs to a zone where the local taxi queue is empty at the moment, as long as at least one of the adjacent zone has an available taxi. However, in this case the system forwards the call to the longer local queue among the ones of the adjacent zones.

  [A reasonable allocation time can be respected by forwarding an incoming call to a taxi driver for a maximum of 30 seconds and thanks to the parallelism and the mandatory assignment approach, which we have presented also in the algorithmic part of this document. Moreover, as we have repeatedly said, if the local queue corresponding to the area of a required ride is temporarily empty, the allocation thread can search for a taxi in the longer local queue among the adjacent ones]
- **[R9]** When the first taxi driver of a local queue does not accept a call, this attempt of taxi allocation fails; so MyTaxiService proceeds immediately by forwarding the same incoming call to the following taxi of the queue; actually for the system this latter has become the new first taxi of the queue whereas the old one is demoted at the bottom. This procedure works cyclically on the queue, so the same taxi can receive the same call more than once or even in the end it should be obliged to accept it. In fact, the system counts the times it has already forwarded the call in question and finally, if none of the first three taxis has accepted the ride, MyTaxiService sends a mandatory assignment to the following available cab in order to assure to accomplish the allocation.

  [We have described the procedure for searching a taxi driver in the Algorithm Design section and once again we have suggested simply a cyclic scanning of the Local queue]
- **[R10]** MyTaxiService cannot accept more than a request with the same credentials and the same origin address for a while of 30 minutes

[Although a client corresponding to a passenger user could forward more than a request through the opened connection with the server, the Request Manager can implement easily a mechanism to avoid useless duplications]

- **[R11]** MyTaxiService cannot accept another reservation with the same credential indicating a time that does not differ from the other one of at least one hour.
  [As we have just mentioned above, also the Reservation Manager can implement some defensive mechanisms to avoid fakes]
- **[R12]**MyTaxiService mobile app must have a specific area dedicated to the taxi drivers of the enterprise. This part of the app is accessible only through a log in operation; hence at the beginning of each shift, the driver must fill in two fields, respectively with his taxi identifier and his password. As soon as the log in concludes successfully, the system sets the state of the taxi in question automatically as available. Therefore, MyTaxiService uses the information obtained by the GPS tracker of the taxi to arrange it in the proper local queue.
  [The special area of the app dedicated to the taxi driver has been shown in the User Interface Design chapter. Furthermore, as specified on more occasions, when the taxi driver requires to log in a connection between this client and the server side is opened; so, provided that the log in operation concludes successfully the Data Manager takes care of updating the state of that driver as available]
- **[R13]** After the log in operation MyTaxiService app allows a taxi driver to enter his private area. Here there is a command that permits the driver to change the state of his cab, choosing among “not available”,”available” or “occupied”.
  [The connection between the server and each taxi driver at working allows this latter to use remote invocations for updating his state on the Data Manager]
- **[R14]** Only the system can associate a taxi also with another possible state which is “unusable”. This state is set by the system if and only if the driver notifies a glitch or any kind of problem by using the special command “URGENT PROBLEM”. This command also allows the driver to be phoned directly by a responsible for problem solving. Anyway, the cab remains in “unusable” state until the hitch is concretely worked out.

[If a taxi driver uses the command URGENT PROBLEM his state on the Data Manager becomes unusable but from that moment we can suppose that only the Data Manager can update again the state]

- **[R15]** The system tries to combine an assignment with a taxi basically by forwarding an incoming call on the MyTaxiService app of the chosen driver. This "false" call lasts 30 seconds and during this while a driver should accept simply by tapping and dragging the icon of ACCETP REQUEST right. Although this call does not open any conversation, immediately after the acceptance the driver receives a notification on the app that specifies origin of the ride and personal data of the applicant to make sure of his/her identity. The notification may inform the driver in case he is going to take care of a previously scheduled reservation.

  [The stateful session bean with a particular taxi driver client is certainly activated when an Allocation thread needs to interact with that driver to forward him an incoming call or to inform him about a just accepted ride]
- **[R16]** The state associated with a taxi changes automatically to "occupied" every time the driver accepts an incoming call from the system.

  [As soon as the taxi driver accepts a ride the app invokes automatically the remote method changing the state into occupied]
- **[R17]**Every time a passenger uses the proper command on the web page or on the app to cancel a ride that a taxi driver has already accepted, this latter is immediately informed by the system with a notification through MyTaxiService app.

  [The thread instantiated by the Selector to handle a specific ride interacts with the corresponding applicant-client; in fact it returns all the information about the incoming taxi and the waiting time to that client. Furthermore, this thread verifies also if the client calls off the just allocated ride before it terminates; so in case of cancellation, it can warn the involved driver]
- **[R18]** MyTaxiService monitors a taxi which state is not available. A taxi can use this state for a maximum of 7 minutes during a working shift. If this time has run out and

the state is still not available the system assigns a great penalty to a driver, updating the appropriate counter in the DB.

[These functionalities can be easily fulfilled by the Data Manager]

- **[R19]** MyTaxiService records every unsuccessful "call", and the taxi in question is moved back from the first position to the bottom of the local queue in which is located. Besides of this, the system punishes the taxi driver by adding a penalty on the appropriate counter in the DB.

  [These tasks are shared between the Data Manager, for what concerns updating the data on the DB, and the Queue Manager and Dispatcher that maintains an ordered view of the available cabs belonging to a local queue by updating conveniently the Local- Queue data structures]

- **[R20]**Both the web application and the mobile app of MyTaxiService provides the passenger a specific form to fill out in order to request a taxi; its fields are the origin of the ride and some personal data of the applicant, actually the name, surname, and a valid email address, all necessary to complete the request. Moreover, the passenger may tick a box to specify the need of two taxis, because of the number of actual passengers or for any other special exigence.

  [As we can observe clearly in the mockups of the User Interface Design chapter]

- **[R21]**MyTaxiService can allocate two taxis for a specific ride with respect to the specification in the request/reservation itself. Anyway the system allocates the taxis as well as for two different rides with the only constraint that the search is confirmed and successfully concluded if and only if both the taxis have been found. Hence, if only one of the two taxis is found the taxi driver who has already accepted the call is quickly informed through a notification that the ride must be deleted.

  [In this case a good solution could be that the selector starts a single thread but the allocation manager and dispatcher launches two different allocation threads; in this case the waiting thread returns the client a positive outcome if and only if the two allocations concludes successfully]

- **[R22]** As soon as a taxi has been found for a certain ride, a confirmation screen or a notification is provided to the passenger, respectively through the web application or the mobile app. This confirmation contains the code of the incoming taxi and the expected waiting time, as well as a command that enables the passenger to cancel immediately the just allocated ride; moreover, in case of web application, the information about the ride to take place are sent also by e-mail.
  [Once again the mockups in the User Interface Design section provide these screens]
- **[R23]** Both the web application and the mobile app of MyTaxiService provides the passenger a specific form to book a taxi reservation; It includes name surname, e-mail address of the applicant as well as the origin of the ride, as usual, but additionally it is necessary to indicate destination and meeting time to schedule. Furthermore MyTaxiService allows a passenger to choose a meeting time which is at least two hours later than the time he/she performs the booking. Finally the booking process ends showing a screen that confirms uniquely the receipt of that reservation and that indicates a code associated with that booking.
  [The Reservation Manager is responsible for verifying if the time required by a passenger for a reservation is acceptable according with the constraint mentioned above; furthermore, if the reservation is accepted it supplies the client with a specific code referring to that instantiated reservation]
- **[R24]**Both the web application and the mobile app have an area in which a passenger can insert a booking code to call that reservation off
  [Through the specific code, used as parameter in a remote invocation, the client can require to delete an already booked reservation. Actually, the Reservation Manager traces the reservation of interest and cancels it, provided that it has not already been selected to be allocated; in this case the cancellation fails and the client must wait to receive the confirmation for the just allocated ride to call it off]
- **[R25]** After a successful allocation of a taxi associated with a previous reservation, the passenger receives a notification on the app or an email, respectively if he/she used the mobile app or the web application for the booking process. However these messages

contain a confirmation for the ride and indicate the code of the incoming taxi as well as an eventual delay with respect to the scheduled time. Moreover, the email contains a link to cancel the ride whereas the app provides as usual a command to cancel it as well. Finally, if the allocation fails, the message via notification or email apologizes for the non-provision of the expected service.

[Starting from the allocation time, we have the same behavior for a reservation as it was an immediate request, and once again we have offered some mockups in the User Interface Design section]