Politecnico di Milano

A.A. 2015-2016

Software Engineering 2 Project: myTaxiService

Design Document

Damiano Binaghi, Giovanni Zuretti

Contents

1 Introduction

1.1 Purpose

This is the Design Document for the development of the myTaxiService application. In this document we are going to identify which are the best choices to take in order to meet all the requirements defined in the Requirement Analysis and Specification Document. Such choices are going to be architectural choices, which will involve both the software and the hardware (e.g. the physical tier division of the application) that will build the actual system which will provide the real service. However our main focus will be on software design, because this document will provide the fundamental specifications that will allow the developers to implement the application.

1.2 Scope

In order to best describe the architecture of the system and the architectural patterns that will be used in the implementation of the application, we re going to use different type of diagrams. Such diagrams should convey as much as possible the information on how the software is going to be structured. Therefore, we will present some UML diagrams (high level component view, component view, deployment view, runtime view) which will help us to define the structure and the behavior of the software and how it is mapped on the physical architecture. We will specify which architectural pattern we have chosen to use how they are mapped in our diagrams, we will give a description of the most crucial algorithms and a description of the user interface. Finally, for sake of completeness, we will show how every requirement defined in the RASD document is mapped in this document.

1.3 Definitions, Acronyms and Abbreviations

We are now going to define some useful acronyms that are going to be used in this document:

- GUI: Graphic User Interface, the graphic interface which will allow users to interact with our application
- API: Application Programmatic Interface
- DBMS: Data Base Management System
- JDBC: Java DataBase Connectivity

1.4 Reference Documents

- Specification Document: Assignment 1 and 2.pdf
- Requirement Analysis and Specification Document: RASD-binaghi-zuretti.pdf

- Specification of the structure: Structure of the design document.pdf
- ISO/IEC/IEEE 42010

1.5 Document Structure

The document will be divided in 6 sections:

1. Introduction: A brief description of the content and the purpose of the document
2. Architectural Design: This will be the main section of the document and will provide the design of the application. It will contains all the UML diagrams needed to specify how the software is going to be implemented (thus, the software design) and the architectural choices made in the design process.
3. Algorithm Design: This section will provide an insight on how the most crucial and important algorithms of our application will work.
4. User Interface Design: This section will provide a deep and satisfactory point of view over the design of the user interface. This has already been artly discussed in the RASD through some mock, however in this document will provide a more complete explanation of the user interface.
5. Requirements Traceability: In this section we will provide the mapping between the requirements identified in the RASD and our design decisions.

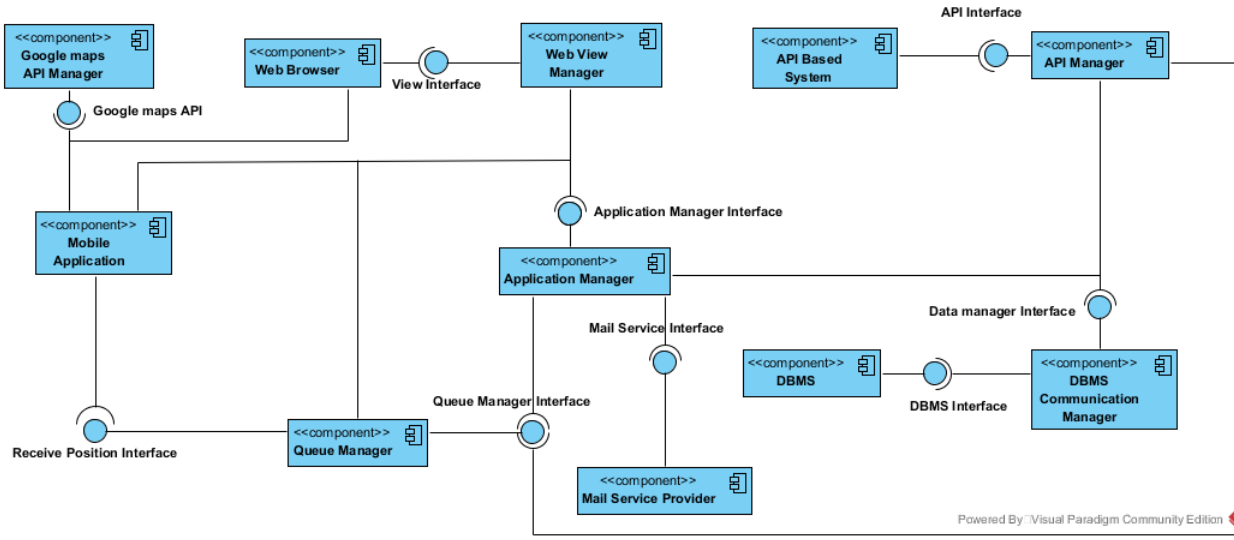
2 Architectural Design

2.1 Overview

In this section we are going to provide the architectural design of our software through some UML diagrams. We are going to show an High Level Component Diagram, which provides a description of the main software components that are going to interact in our application. Then we are going to describe in a more detailed way such components in the "Component View" subsection, in which we will explore which are the software modules that allow our components to work in the proper way. We will show also, through means of a Deployment Diagram, how the software components are going to be mapped on the physical tiers once the application will be implemented. Afterwards we are going to provide a Runtime View of our system, in which we will show how the previously defined components and modules interact among them when the system is going to actually be deployed. We will describe also how the interfaces between components work and what were the architectural tyles and pattern we used in designing the application. Finally, we are going to provide other details on design decisions we think are relevant, but which did not deserve a stand-alone subsection to be explained.

2.2 High Level Component View

We are now going to present the High Level Component View of our system:



We will now provide a brief description of the principal components of our system. We will not specify anything about those components which are not a concern for the implementation of our software, i.e. that we cannot control and that are not going to be implemented by our application (such the Google Maps API Manager, the Web Browser, the Mail Service Provider and the API Based System).

2.2.1 Mobile Application and Web View Manager

This two components are those in charge to provide the presentation layer of the application to the client. The Mobile Application component is the software that is going to be installed on the mobile device of the users and it will provide the GUI for such devices, while the Web View Manager will provide the GUI for the Web Application. Based on the kind of user (Customer, Taxi Driver, Taxi Data Manager, Application Manager) they will show different interfaces to allow to use the different functionalities each of them has access to.

2.2.2 Application Manager

This is the core component of our application and it embodies the logic behind our application. This component will provide the software modules that will allow to communicate with the clients and manage their requests.

2.2.3 Queue Manager

This component will take care of the management of the queues in each zone of the city. It is also a core component, because it will contain the most critical algorithms which will allow our application to work properly. We decided to model this component outside the Application Manager both to highlight its critical role and, as we will specify later in the deployment view, to highlight that such component is not strictly dependent from the Application Manager itself, but it can be a "stand-alone" component.

2.2.4 API Manager

This component will take care of managing the requests coming from a possible future system based on the APIs offered by our application and to communicate with such system.

2.2.5 DBMS Communication Manager

This component will provide an interface for the Application Manager and the API Manager towards the DBMS in order to allow those components to query the database in a transparent way in respect to the DBMS chosen.

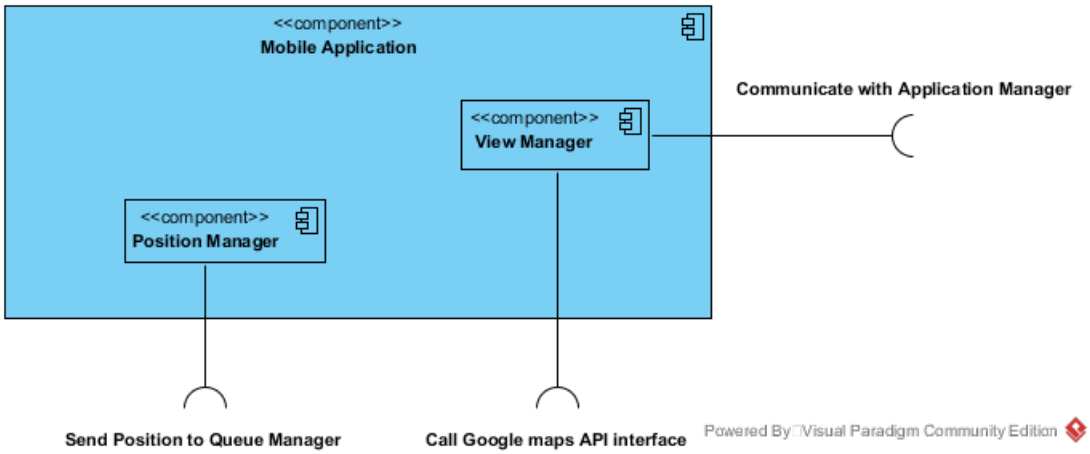
2.2.6 DBMS

This component will manage the query on the actual database. It could be one of the many DBMS that exists and are available in the world, with the constraint that it must allow to use triggers, in order to build an Active Database. Triggers are needed in order to manage in a simple way requests of advance-booking.

2.3 Component View

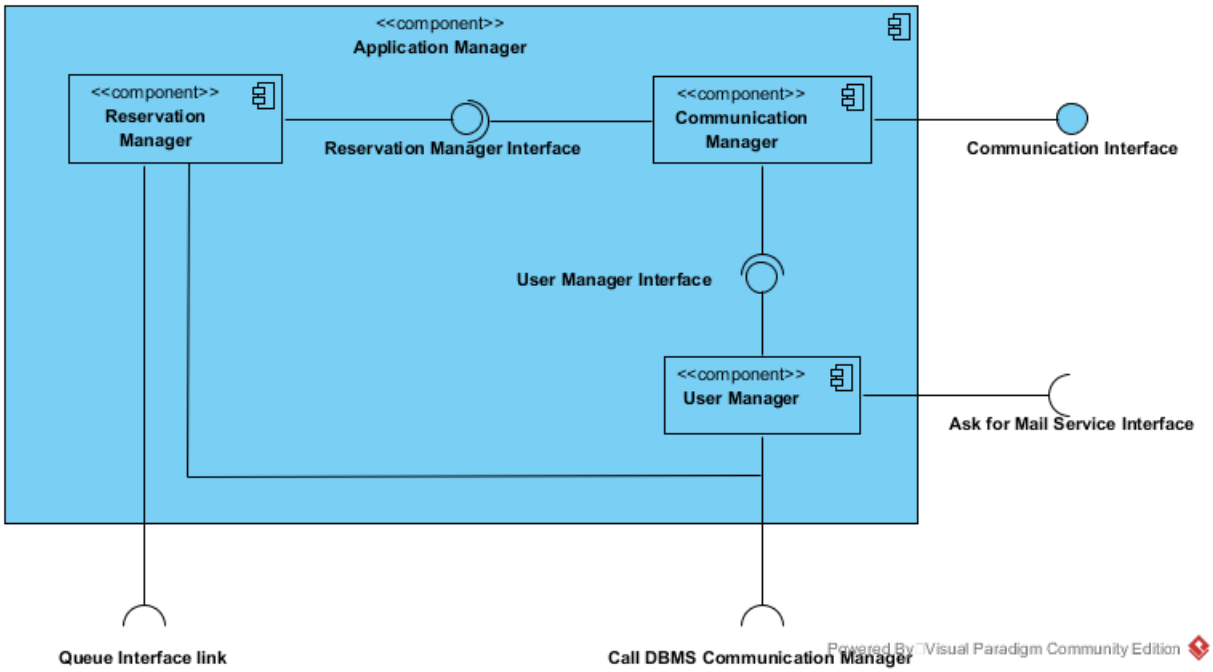
We are now going to describe in a more detailed way the components we have previously defined, also by using Component Diagrams that we will show which are the software modules contained by components. We are going to show the details only of those components in which more than one software module is actually needed to better understand how such components work. Components like the Web Presentation Layer, built by just one module we is not in charge of any particular computation, are not going to be detailed in a deeper way.

2.3.1 Mobile Application Component



This is a very simple component. Its software modules are just two: the View Manager which is in charge of showing the right view to the user, based on the type of user (Customer, Taxi Driver, Taxi Data Manager and Application Manager), and the Position Manager, which is used by the application software only if the logged user is a Taxi Driver, and keeps sending to the Queue Manager the position of the Taxi Driver thanks to the GPS on the Taxi Driver's device. Furthermore, the View Manager is the module that communicates with the Application Manager, in order to allow the application to actually provide all the requested functionalities, and with the Google Maps API Interface, when there is the need of retrieving the time a customer needs to wait for the taxi, which has been assigned to him, to pick him up.

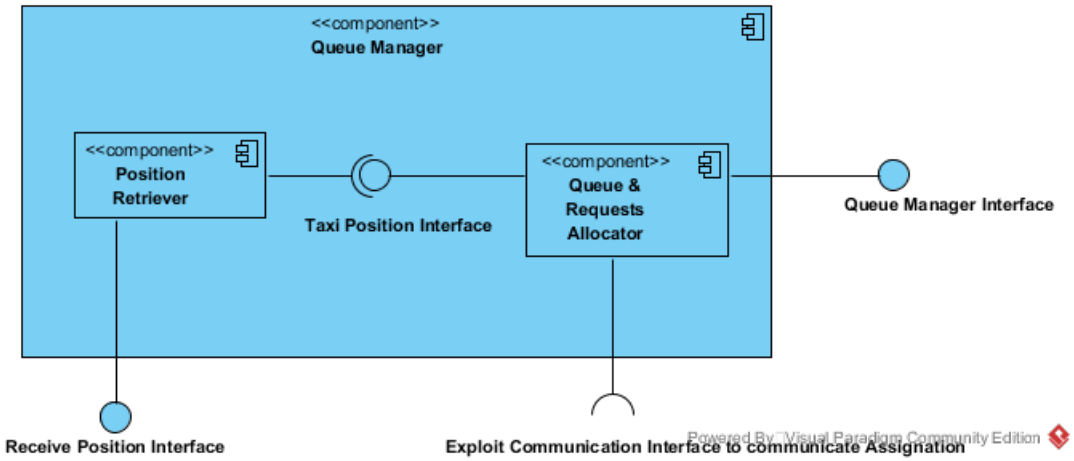
2.3.2 Application Manager



The Application Manager is divided in three principal software modules which allow it to provide all the required functionalities. The Communication Manager takes care of all the messages exchanging that happens in the application by providing a unique interface that allows the clients to communicate with others part of the system. We would like to underline that such interface is unique, and does not depend on the other component that is going to use it. What it provides is the functionality of exchanging messages from an internal-component to a client and vice versa. The messages can concern whatever the client needs to do (log-in, book a taxi, report a fake user, report an accident, etc.). The Communication Manager will dispatch the message coming from the client to the right internal component or software module and vice versa. Then there is the Reservation Manager. The tasks of these software module are to handle the incoming requests of instant-booking, by calling the Queue Manager in order to allocate a new taxi and generating the ride code which will be used for confirmation, and to call the DBMS Communication Manager to create a trigger in order to manage a new advance-booking request. Finally the User Manager module will provide all the functionalities needed for the registration of a new user profile, modification of an already existing profile and authentication. In order to work correctly, such software module shall communicate with the DBMS Communication Manager (for looking up and verifying login credentials, for modification of data, etc.) and with an e-mail service through the Mail Service Interface, for sending confirmation mail during the registration

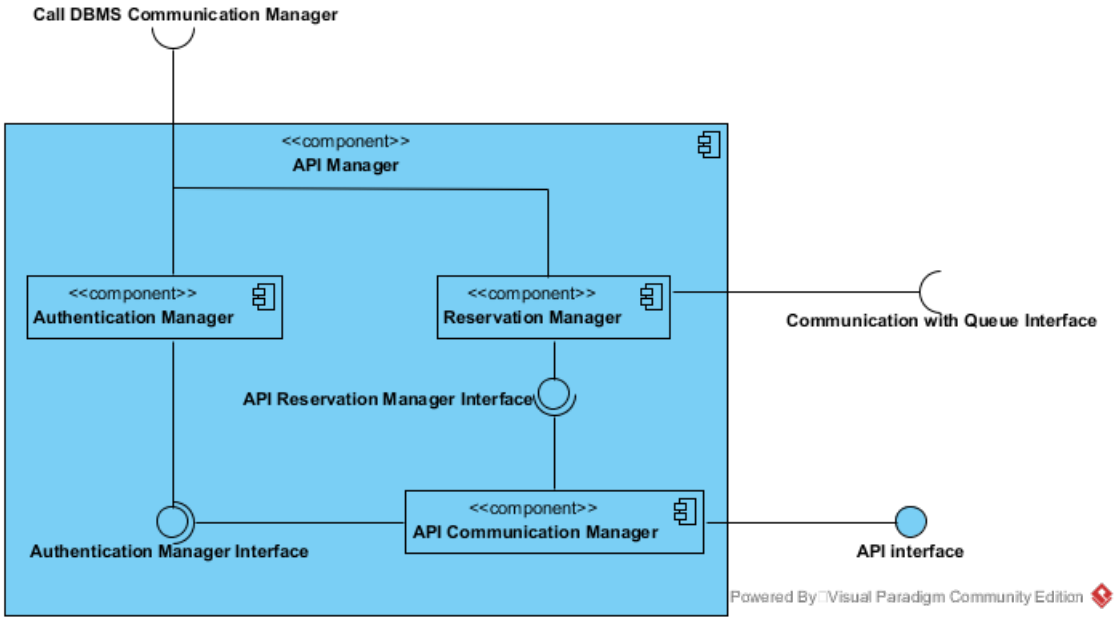
process. Such component will also register the fake user reports, associating each report to the reported user.

2.3.3 Queue Manager



This component is composed by two software modules. The Position Retriever just keeps on listening for changes in the positions of the taxi drivers and communicates such changes to the Queue & Requests Allocator. This second module is the most important part of this component, because it takes care of managing all the queues with the relative changes, moving taxi drivers between different queues, and to return a Taxi Driver to the Application Manager when such component asks to return an Assignation to a pending Request. The Queue & Requests Allocator will take care of calculating which is the nearest Taxi to the customer who has created the request and it will also contact the Taxi Driver through the Communication Interface. It will repeat this process until a Taxi Driver will confirm the assignation, and it will return the assigned Taxi to the Application Manger.

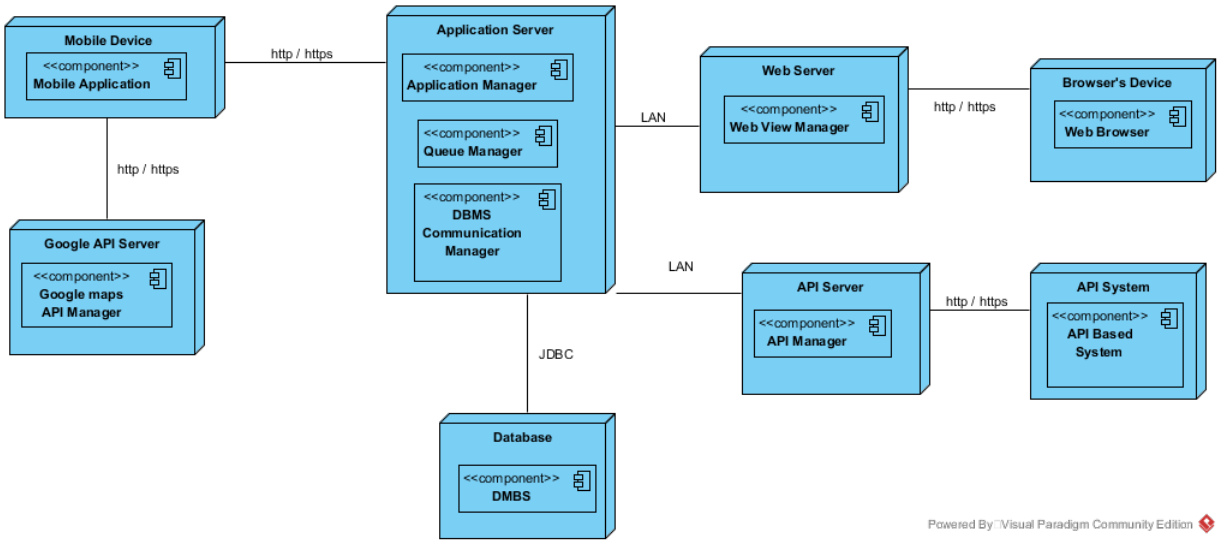
2.3.4 API Manager



The API Manager component replicates most of the functionalities of the Application Manager, which we have already discussed. The main difference is that in the API Manager does not have a User Manager module, but just an Authentication Module in order to let the user of the API system, that will use our API interface, to log in. Thus no registration or change of personal data is allowed through the API interface, while the other functionalities remain the same. We decide to divide this component from the Application Manager, even though this means replicating some functionalities, because we want to separate as much as possible the API world from the Application world.

2.4 Deployment View

We are now going to describe the Deployment View, i.e. how our software components are going to be displaced on the hardware, giving an insight on the reasoning that led us to take the choices we took.



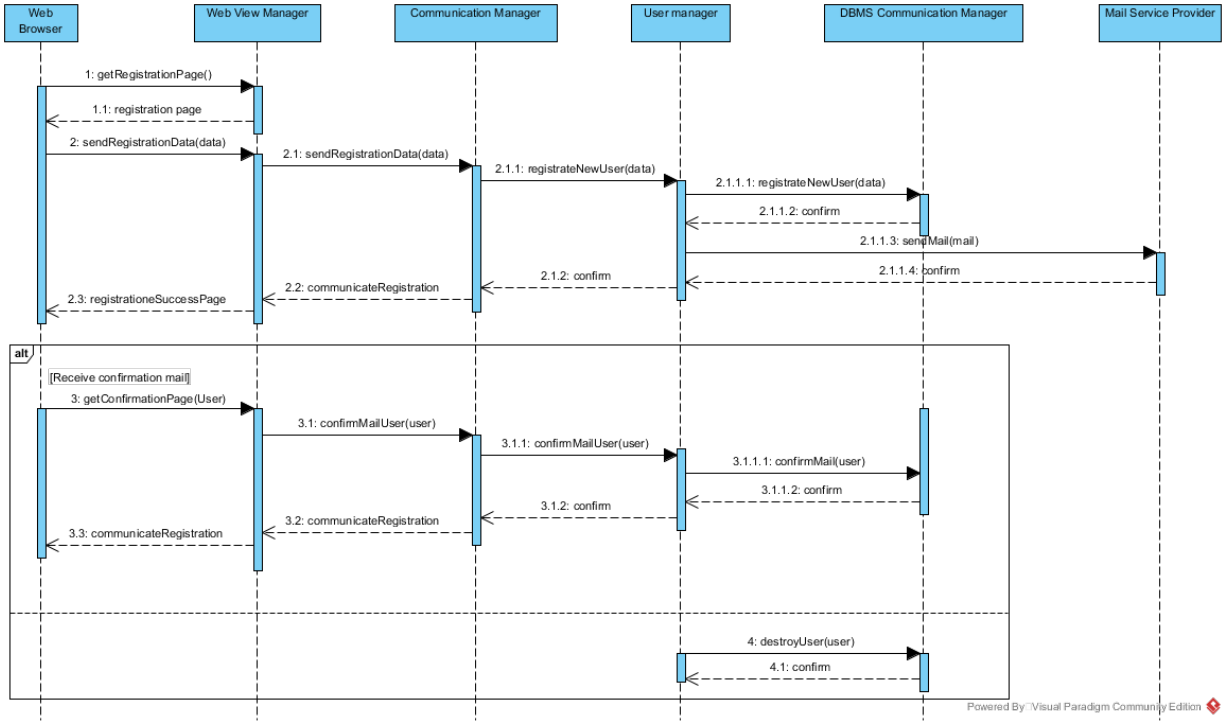
The main node is the Applicaton Server which will contain the Appllication Manager component, the Queue Manager and the DBMS Communication Manager. We felt to put this components all together, because, when we will launch our application on the market, we thin that all the traffic can be managed by just one node. Thus the application can run faster because it does not have to communicate with external nodes when it needs to use one of these three components. However, the Queue Manager and DBMS Communication manger components where designed as divided from the Application Manager, because in presence of scalability issues we could easily move them in external dedicated node. Then we have the API Server node, which is external to the Application Server because, as previously stated, we want to separate as much as possible the Application world from the world of other applications that will use our system APIs. Particularly, we don't want the traffic coming from others applications' requests overload the server of our application. For sure, the decision of actually build an API server can be taken at any time, when some applications that use our APIs actually exist, but we wanted to underline now that in such case a node dedicated entirely to manage the API manager component is probably the best choice. Then we have the Web Server node, which will contain the Web View Manager component. We choose to put into two different node the Web View Manager and the Application Manager in order to divide the presentation layer from the logic of the application. Finally, we have the Database node, which will contain the DBMS component and the data of our application. We will not explain how the other nodes work, because we cannot control them,

however we decided to show them for sake of completeness.

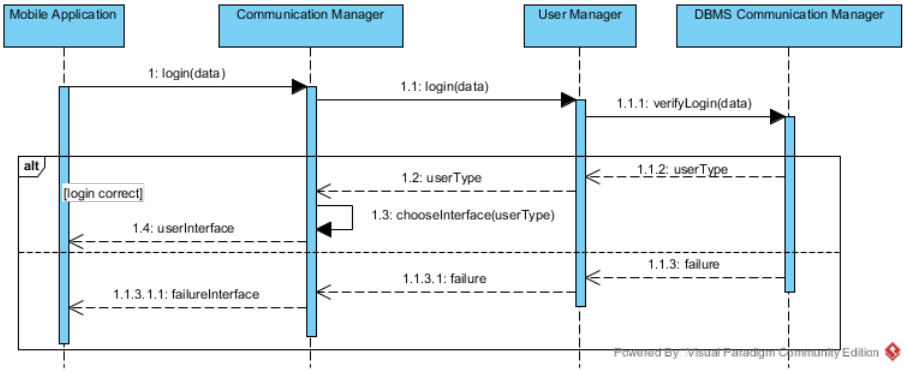
2.5 Runtime View

We are now going to present through means of some sequence diagrams the Runtime View of our system, i.e. how the different software modules we defined in the component view will interact between them once our application will be deployed. In the RASD we have presented similar diagrams where the system was a generic entity and we did not specify how the things were done inside the system. On the contrary now, we are going to see what happens inside the system at runtime.

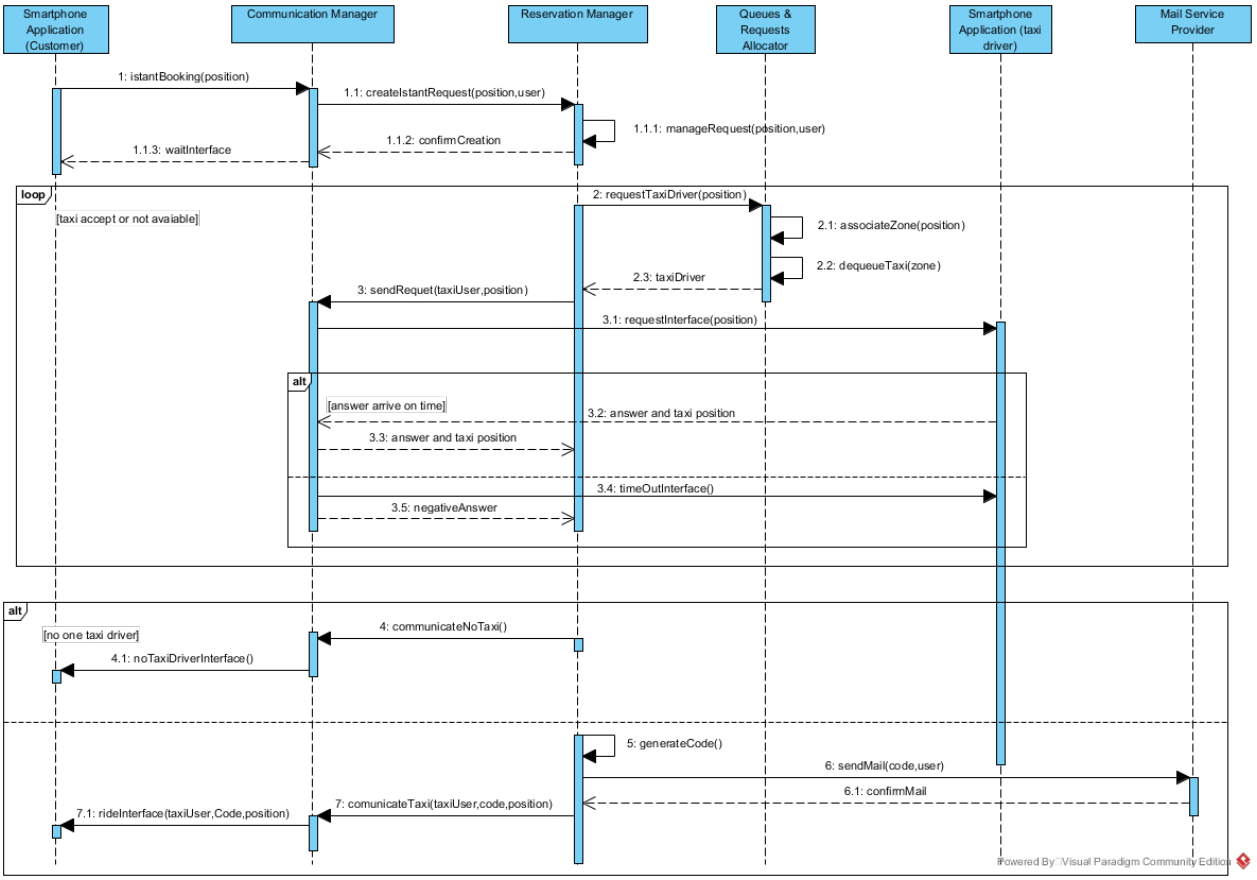
2.5.1 Registration



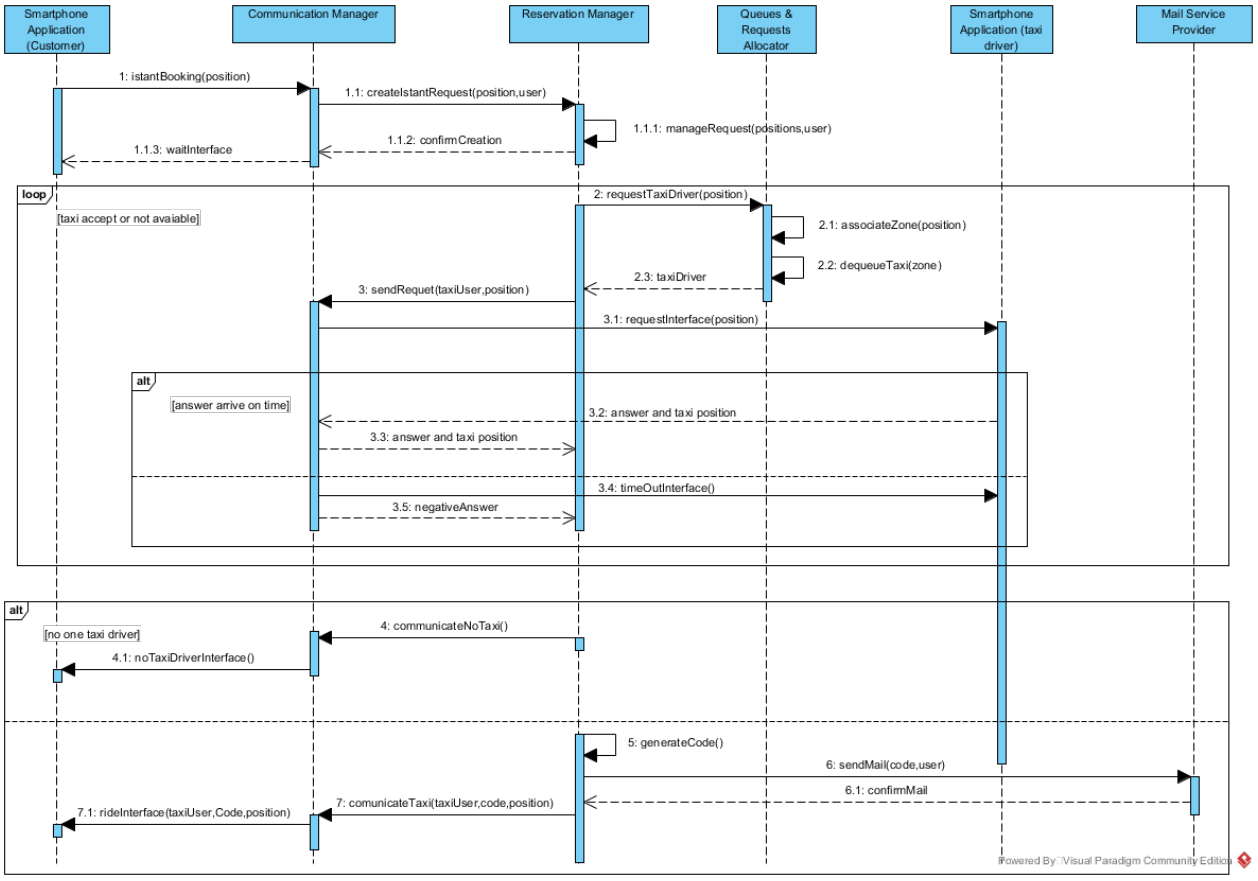
2.5.2 Log in



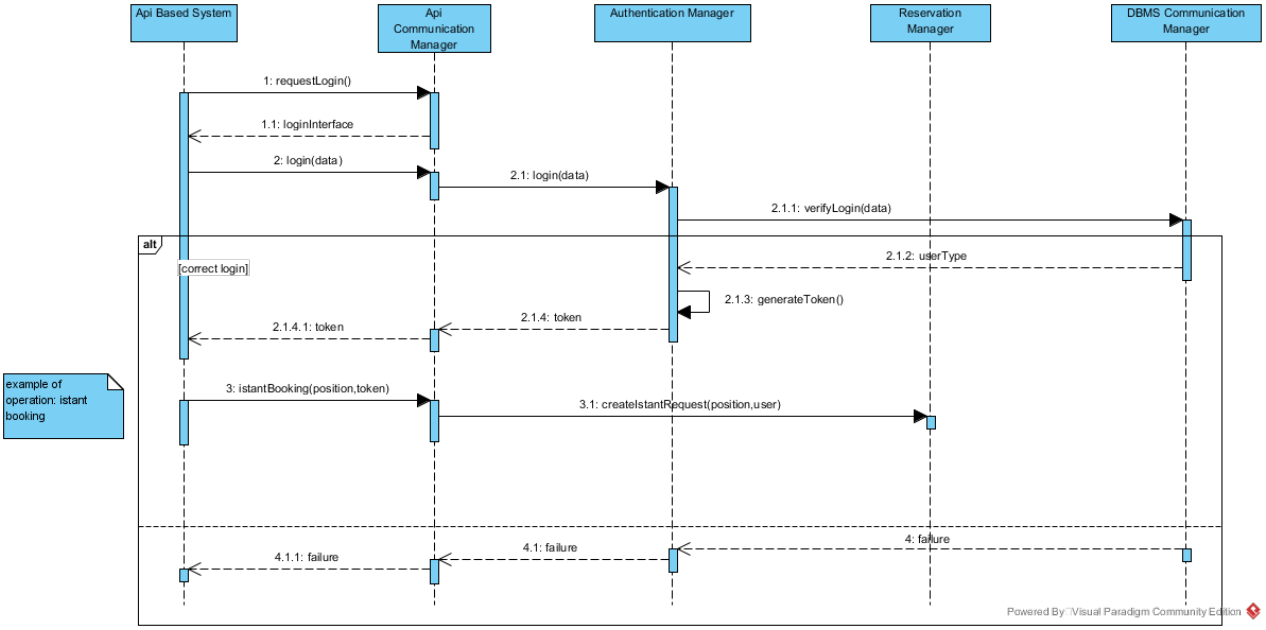
2.5.3 Real Time Booking



2.5.4 Advance Booking



2.5.5 API Request



2.6 Component Interface

We are now going to give a more detailed description of the interfaces that allow our components to communicate one with each other.

2.6.1 Communication Manager Interface

- Provided by: Communication Manager module in the Application Manager component.
- Used by the following components: Mobile Application, Web View Manager, Queue Manager.
- How it works: This interface is the point of contact between the component Application Manager and the rest of our application world. It allows the Web View Manager and the Mobile Application to communicate to the logic of the application the actions and requests done by the client (such as login, ask for a instant booking (customer-side), ask for an advance booking (customer side), accept or refuse assignation (taxi driver side), report fake user, report an accident, etc.). It allows the Queue Manager to communicate an assignation to the Taxi Driver chosen by the assignation algorithm.

2.6.2 Reservation Manager Interface

- Provided by: Reservation Manager module in the Application Manager component.
- Used by: Communication Manager module in the Application Manager component.
- How it works: This interface provides the methods for communicating to the Reservation Manager that a new booking instance needs to be handled. Such methods are called by the Communication Manager when it receives a new request of real-time booking, when a trigger, corresponding to an advance-booking request, is fired in the database and it notifies the Communication Manager that there is a new reservation to handle, or when a taxi driver reports a problem and thus there is the need of allocating a new taxi to the pending request . Through this interface the Communication Manager can, thus, communicate to the Request Manager the need of managing a new requests.

2.6.3 User Manager Interface

- Provided by: User Manager module in the Application Manager component.
- Used by: Communication Manager module in the Application Manager component.
- How it works: This interface provides the methods for calling the User Manager to handle registration, authentication and editing of the personal. When a client need to perform one of the previous listed actions the Communication Manager, which receives the request from the client, passes the request to the User Manager which will actually handle it.

2.6.4 Receive Position Interface

- Provided by: Position Retriever module in the Queue Manager component.
- Used by: Position Manager module in the Mobile Application component.
- How it works: The Position Manager use the GPS information on the device of the user to sends the position of the taxi to the queue manager. In order to do this, the Position Manager calls the Position Retriever and keeps updating the position of the taxi based on the information coming from the GPS. Thus, we can say that the device of the taxi driver will keep sending information about its position to the Queue Manager which will manage how the taxi driver is placed in the queue. The interface that allows to keep track of the position of a taxi is the Position Retriever interface, which is simply a listen-and-update module, which keeps on receiving a position and update it.

2.6.5 Queue Manager Interface

- Provided by: Queue & Requests Allocator module in the Queue Manager component.
- Used by: Reservation Manager module in the Application Manager component or in the API Manager component.
- How it works: The Queue Manager Interface provides the method to assign a taxi driver to a pending request. On a new booking instance which needs to be handled, the Reservation Manager calls the Queue Manager Interface asking to allocate a taxi to the request. The Queue & Requests Manager will start to run the assignation algorithm until a taxi driver accepts the assignation (in order to do this it will use the Communication Interface as seen previously), and then it will return the information about the taxi (taxi plate and position) to the Reservation Manager.

2.6.6 Taxi Position Interface

- Provided by: Queue & Requests Allocator module in the Queue Manager component.
- Used by: Position Retriever module in the Queue Manager component.
- How it works: The Taxi Position interface is used simply to notify the Queue & Requests Manager when a taxi changes zone in the city or when a new taxi driver logs in into the system. As already seen, the Position Retriever component keeps updating the position of taxis by listening to the GPS information sent by the application of the taxi drivers. The Position Retriever will notify the Queue & Requests Allocator by using the Taxi Position Interface only when the new acquired position comes from a different zone with respect to the last saved position or when the taxi driver has just logged into the system, thus he did not have a previous position.

2.6.7 API Interface

- Provided by: API Communication Manager module in the API Manager component.
- Used by: API based system component.
- How it works: The API Interface is used by the API Based System to call one of the functionalities that our system provide as APIs. Such functionalities are authentication, instant-booking and advance-booking. In order to provide such functionalities, the software modules of the API Manager component use very similar interfaces to the ones used in the Application Manager component, replicating the functionalities we have just listed. Such interfaces are the Authentication Manager Interface and

the API Manager Interface for which explanation of how they works would be redundant, because already seen in previous subsections. In conclusion, the API interface provides all methods that allow the API Based System to communicate requests of different types (e.g. authentication and booking) to our system.

2.6.8 DBMS Interface

- Provided by: DBMS component.
- Used by: DBMS Communication Manager component.
- How it works: The DBMS Interface is provided in the database and it can be of many types (MySQL, Oracle, SQL Server etc.). Such choice will be taken in the development phase. The DBMS Communication Manager is an abstraction of this interface, so it is not bounded to the specific type of DBMS. Therefore it must use the DBMS Interface to actually carry out the operations on the real database.

2.6.9 Mail Service Interface

- Provided by: Mail Service component.
- Used by: Application Manager component.
- How it works: This interface is used by the Application Manager when it needs to send an email to a client, in order to communicate the confirmation of the registration or the assignation of a taxi to a pending request made by the client himself. This is not an interface we can control. It depends on the service we will decide to use in the development phase.

2.6.10 Google Maps API

- Provided by: Google Maps API Manager Component
- Used by: Mobile Application component and Web Browser component.
- How it works: This interface is used by the components just listed to ask for the calculation of the time a client has to wait for the taxi, which is coming to pick him up, to actually arrive at his position. It is done by sending to the Google API both the coordinates of the client and of the taxi driver. The Google API manager will calculate the time needed to cover such distance and it will return it to the client. Again this is not an interface we can control, thus at development phase programmers must be aware of how such interface actually works. This explanation is provided at the following web address: https://developers.google.com/maps/

2.7 Selected Architectural Styles and Patterns

We are now going to show which are the architectural styles and patterns we chose in the process of designing the application. We can say our application is Service Based, Event Based and Client-Server (styles) and that it follows the Model-View-Controller (MVC) and the Multi-Tier patterns. We are going to dedicate one subsection to each style to better explain where and how we used it.

2.7.1 Service Based Application

MyTaxiService is a Service Based Application because both it uses and it provides some sort of services, but we cannot say it is service oriented because it is not true that each component of the application can be seen as a service and can be externally used as a service. The used services are Google Maps and a generic e-mail service as shown in the component diagram. The provided services are those specified in the API manager, thus the possibility of authentication in our system and the possibility to use customer functionalities such as the booking of a taxi. We decided to use such style because it is intrinsic in the specification of how the application should work the need of using some services (such as Google Maps or an E-mail Service) and the need of providing some others (in the specification document it is explicitly requested to provide APIs of our system for possible future development).

2.7.2 Event Based Application

MyTaxiService is an Event Based Application because the request of booking a new ride made by a customer is managed as an event by the system. Thus, the event is the new incoming request and the subscribers to such an event are the customer who made the request himself and the taxi drivers. Taxi drivers are notified following a queue logic, thus, from the taxi driver point of view, a priority is defined over the event. This implies that only the taxi driver with the highest priority is notified of the event. The concept of priority depends both on the zone in which the taxi driver is and the time he has been waiting for a new assignation. From the customer point of view, when his request is assigned with success to a taxi driver, the state of the event changes and he is notify of such changes. For such reasons, we feel confident in saying that this part of the system is designed according to the Event Based style. We decided to use such a style because, when facing the problem of booking taxis, it was clear that a booking instance can be seen as an event that is created in our system when a customer decide to book a taxi. From our point this is the style that best fits such feature because the client should be free to do whatever he wants once he has created a new booking instance and he must be notified only when the status of the instance has changed (thus, we have a change of an event which is notified to one of its subscriber).

2.7.3 Client-Server Application

MyTaxiService is also a Client-Server Application because the user profile and the authentication are managed in a typical Client-Server style. There is the client that sends a request to the server (e.g for the login, or for the editing of his personal information) and waits for an answer by the server. Only when the server answers, the client can proceed, e.g. only when the server has verified the correctness of the login credentials the client is actually logged into the system and can access his homepage. We decide to use such style for managing this part of our application because it is the most common style used to manage such functionalities, therefore we can reuse a widely known style which has already been implemented. We do not need to invent anything new, because in most of the applications existing nowadays such style is used for managing the same features we need to manage in our particular case.

2.7.4 MVC Pattern in our Application

MyTaxiService is based on the Model-View-Controller (MVC) pattern. We decide to use such pattern for the development of the application because we have different type of views (Mobile Application and Web Application) and the MVC pattern is typically used in such cases. Furthermore it allows to have the nice property of scalability with respect to the type of views. If in the future we want to expand our application on other kinds of devices, we will just need to make a different view for that specific device, without changing anything in the application logic or model. The components which implement the View of our system are the Web View Manager, for the web application, and the Application Manager, for the mobile application. The Model of the system are the data stored in the database, while the Controller is implemented by the components on the Application Server (Application Manager, Queue Manager and DBMS Communication Manager). We underline that the Queue Manager and the Reservation Manager, which is inside the Application Manager, are considered part of the controller but actually have also part of the model (the queues and the active booking instances).

2.7.5 Multi-Tier Pattern

As we can see in the deployment view, the configuration of our application leans toward a multi-tier architecture. This pattern was selected to divide the parts of our software which accomplish different tasks. What we tried to do is to separate as much as possible the data from the logic of the application and the logic from the user interface. Therefore we selected a 4-tier architecture for the Web Application (client, web server, application server, database), which does not require the installation of any software client side, and a 3-tier architecture for the mobile application (client, application server, database), because it requires the installation of an application file in the client's device which contains the view functionalities that in the web application are provided by the web server. Therefore, we have a thin-client configuration for both the web application and

the mobile application. In the case of the web application, the client does not hold anything, just a browser, while in the mobile application case one could comply that it is not a thin-client because a file must actually be installed on the client for the application to work. However, in such case, what it is installed on the device are just the pages that must be displayed to the user, and no computation is done by the client. All computations are left to other components (either the Application Manager or the Google maps API manager). This is why we say that also the mobile application is thin client.

2.8 Other Design Decisions

We are now going to provide an explanation of those design decisions that were not described in previous sections, but for which we think a remark and a more detailed description are important.

2.8.1 Java and JDBC

As shown in the Deployment View, our intention is to use the JDBC interface for communication between the database and our application. This means our application must be able to use such interface, therefore binding us to use Java as programming language for the implementation. We think the choice of this language is good for the implementation of the application, because Java Enterprise Edition allows and supports the development of a multi-tier architecture, which is our goal, and Java is also the standard for the development of applications on Android. (chiedi al canaglia)

2.8.2 Active Database

As we outlined in the definition of the DBMS component, our application must use an active database in order to properly work and satisfy the requests of the customer. We need an active database because we need to use triggers to correctly handle instances of advance-booking. A trigger is created in the database when a new request of advance-booking arrives in the system. The Reservation Manager calls the DBMS Communication Manager passing the data about the advance booking. The DBMS Communication Manager will call the actual methods that allows to create a trigger in the real database. The trigger must fire 10 minutes before the time specified in the reservation and it must communicate to the Reservation Manager, again this is done passing through the DBMS Communication Manager, that it needs to handle a new booking instance and it must allocate a taxi to such instance.

3 Algorithm Design

We are now going to present the most relevant algorithms of our application using some pseudo-code. The algorithms on which we focused our attention are those which manage the queues and the allocation of a taxi to an incoming request.

3.1 New incoming request handled in the Request Manager

The algorithm create a new request for the customer who booked a taxi and asks to the Queue & Reservation Manager to allocate a taxi to the request. When the allocated taxi is returned, the Request Manager communicates to the customer the confirmation code. If no taxi driver is available, temporary unavailability of taxi drivers is communicated to the customer.

```
newRequest (user, position){
    found = false
    while (!found && available)
        try
            taxiDriver = requestTaxiDriver (position)
            taxiDriver.priority = 0
            found = taxiDriver.request (user, position)
        catch unavailableException
            available = false
    if(!available)
        user.communicateUnavailable()
    else
        code = generateCode()
        user.communicateRideInfortmation (taxiDriver.position , code)
end
```

3.2 Allocating a taxi driver for a request

We will show now how the requestTaxiDriver (position) function used in the previous code should work. This function is implemented in the Queue & Request Allocator

```
requestTaxiDriver (position){
    zone= associateZone (position)
    zoneIterator = zone.near()
// find the nearest zone with a taxi driver available
    while ((zone.queue.length = 0 || zone.everyTaxiDriverRefuse) && zone.hasNext())
        zone = zoneIterator.next()
// if no taxi driver is available throw exception
    if ((zone.queue.length = 0 || zone.everyTaxiDriverRefuse))
        throw unavailableException
```

```
        taxiDriver= zone.queue.dequeue() // dequeue taxi driver
        return taxiDriver
        end
```

3.3 Updating the position of taxi drivers

We are now going to present how the method for managing drivers who are waiting in queue should work. This method is called in the Position Manager which uses the queue and remove methods of the Queue & Request Manager to update the queues when the zone of a taxi driver changes.

```
    updatePosition (user,position)
        user.setPosition(position)
        zone = associateZone(position)
        if (user.zone != zone) // if the zone changes, update the queues
            oldQueue = user.zone.queue
            oldQueue.remove(user)
            user.zone = zone
            queue = zone.queue
            queue.enqueue(user)
    end
```

3.4 The Queue Object

Finally we are going to define what a queue is and how a queue works. This is done for sake of completeness, but it should be well known to who is going to actually implement the application.

```
    enqueueUser (user) // function that manages the insertion of a taxi driver
to a queue
        i=this.length-1 //where 'this' is a queue object
        while (i>=0 && user.priority > this.get(i).priority) //find the right
position
            i–
        this.insert (user, i+1)
    end

    dequeue()
        if(this.length>0)
            user=this.get(0)
            this.remove(0)
            return user
        throw emptyQueueException
    end

    remove (user)
        i=0
        while (i<this.length && user.equals(this.get(i)))
```

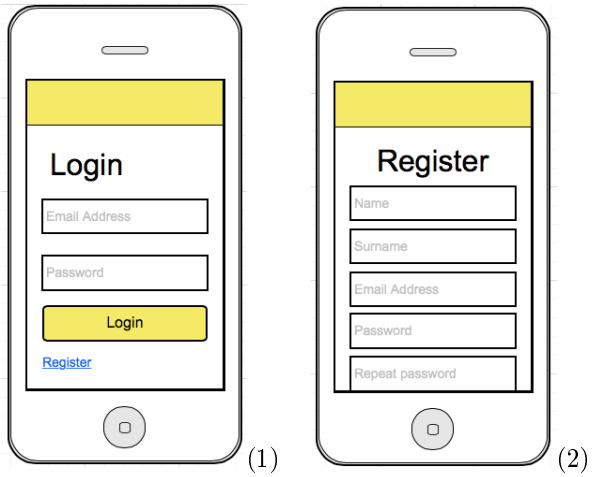
```
            i++
        if(i<this.length)
            this.remove(i)
    end
```

4 User Interface Design

In this section we are going to provide a description of how the user interface should look like. We have already partially done it in the RASD, but what we are going to present here impose a higher degree of constraint. We are going to present the user interface only for the mobile application, leaving to the development phase the task of adapting such interface to the web application. We will also try to show the flow of pages when a user interacts with the application.

4.1 Home Page, Log in and Registration

The home page (1) allows the user to login or to access the registration page (2).

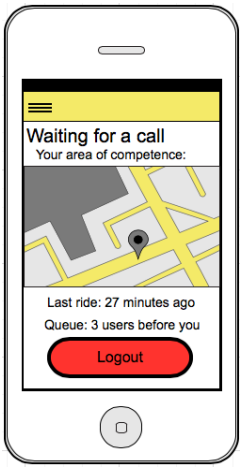


(1) (2)

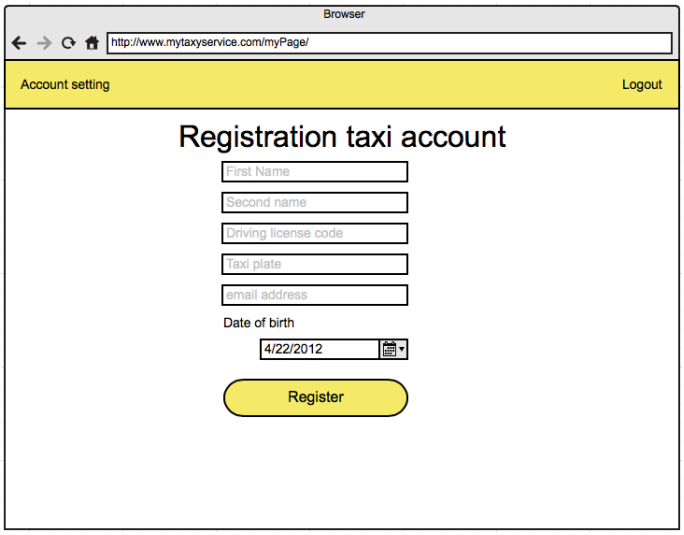
4.2 Personal Home Page

We have two types of personal home pages in the mobile application, one for the customers (3) and one for the taxi driver (4). We provide in this section also the personal homepage for the taxi-data manager user, which actually is accessible only from the web application.

(3)



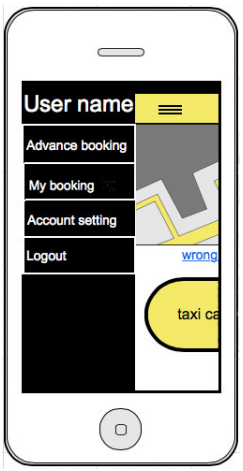
(4)



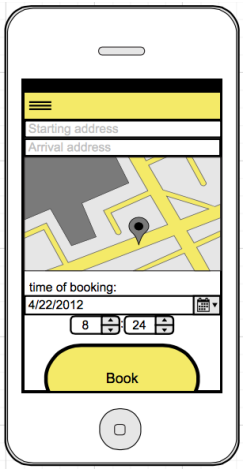
(5)

4.3 Customer Menu

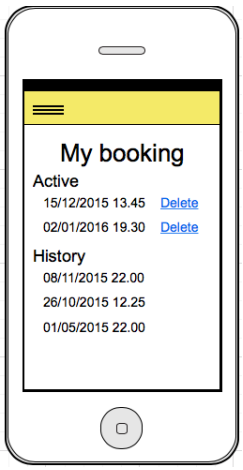
We are now going to present how the customer menu will appear (6) and the pages that are accessible from such menu ((7), (8), (9)). From such pages, once the actions a customer wants to do are finished, the custormer will return to his personal homepage.



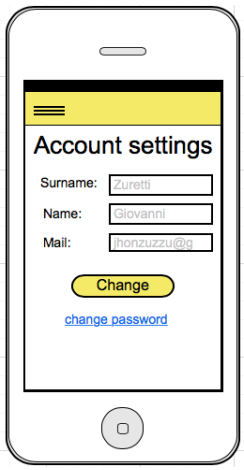
(6)



(7)



(8)

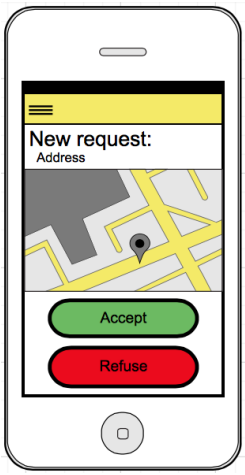


(9)

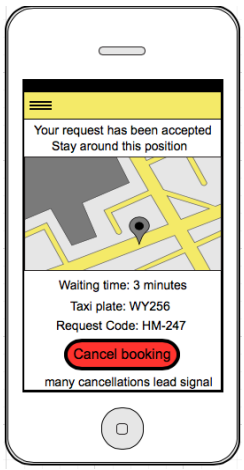
4.4 Taxi Driver & Ride GUI

We are now going to show what happens at the GUI level when a new booking instance that needs to be handled enters the system. When a customer has booked a taxi he simply return to his personal homepage. While from the taxi driver point of view, when he is assigned to a reservation, he is notified on his device and the screen (10) is shown to him. If he refuse to take the assignation he is simply redirected to his personal homepage. If he accepts the screen (11)

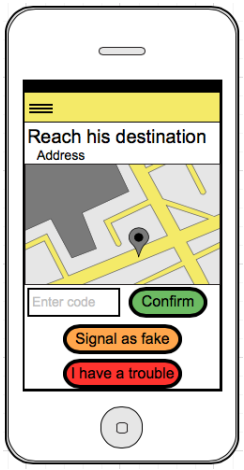
appears on his device while the customer is notified and the screen (12) is shown to him. Three scenarios can then happen: the taxi picks up the customer and the ride is correctly completed (13); the customer does not show up at the arranged place, and he is reported as a fake user (14); the taxi incurs in some sort of problem and can not complete correctly the ride (15). After one of these actions the taxi driver is redirected to his personal homepage.



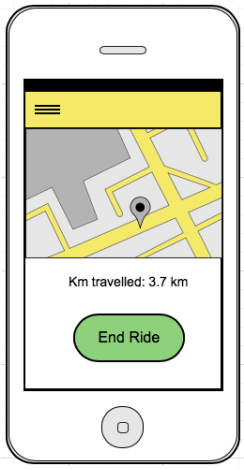
(10)



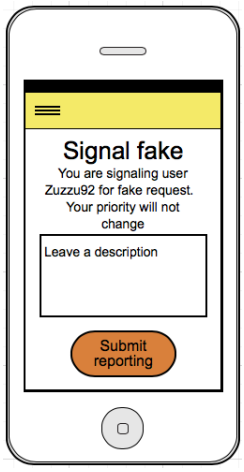
(11)



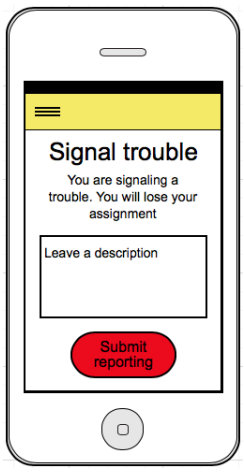
(12)



(13)



(14)



(15)

5 Requirement Traceability

We are now going to show a mapping between each requirement we defined in the RASD and the components or the design choices that allow us to satisfy them. We will have a subsection for each goal, and in each subsection there will be a table with the requirements in the first column and the respective design components and decisions in the second column.

5.1 Allow registration of a new user to the system

• The system must provide the sign up functionality	• User Manager
• The system must send an e-mail to the address specified by the registering user for confirmation.	• User Manager • Mail Service Interface
• When confirmation is received the system must add the new user to the user table in the DB.	• User Manager • DBMS Communication Manager

5.2 Allow log-in to already existing users

• The system must provide an input form in which the user can insert his login data.	• Web View Manager (for the web application) • Application Manager (for the mobile application)
• The system must check the correctness of the inserted data. (Right email matched with the right password)	• User Manager
• If log-in data are correct, the system must redirect the user to his personal page.	• Communication Manager • Web View Manager • Application Manager

5.3 Allow users to modify his personal information

• The system must provide an interface for the modification of personal data by the users.	• Web View Manager • Application Manager
• The system must update the modified information in the DB.	• User Manager • DBMS Communication Manager

5.4 Allow insertion of new taxis and taxi drivers by the Taxi-Data manager

• The system must provide an input form for the Taxi-Data manager to insert new taxis and taxi drivers.	• Web View Manager
• In case a new taxi driver is added, the system must generate a unique password and send it to the taxi driver.	• User Manager • Mail Service Interface
• The system must add the new items to the respective tables in the DB.	• User Manager • DBMS Communication Manager

5.5 Allow Taxi-Data manager to delete taxis that no longer exist or taxi drivers who no longer provide the service

• The system must provide an interface which allows the deletion of taxis and drivers.	• Web View Manager
• The system must perform the changes on the DB.	• User Manager • DBMS Communication Manager

5.6 Allow Application manager to manage all other users (insertion of new TaxiData manager, change privileges of some users, etc.)

• The system must provide an interface which allows the Application Manager to modify the data he wants and he is allowed to modify.	• Web View Manager
• The system must perform the changes on the DB.	• User Manager • DBMS Communication Manager

5.7 Allow customer to real-time book a taxi ride and notify the assigned taxi driver

• The system must provide an input form which allows the customer to insert the information about his position.	• Web View Manager • Application Manager
• The system must assign from the taxi queue of the zone in which the customer is (or, if no taxis in that zone are free, from the other closest zone) a taxi cab to the request.	• Reservation Manager • Queue & Requests Manager
• The system must send a notification to the driver of the assigned taxi cab.	• Communication Manager

5.8 Allow customer to book in advance a taxi ride (advance-booking) and notify the assigned taxi driver

• The system must provide an input form which allows the customer to insert information about the starting point, the destination point and the time at which he will need the taxi.	• Web View Manager • Application Manager
• The system must save this reservation into the advance-booking table of the DB.	• Reservation Manager • DBMS Communication Manager
• 10 minutes before the time specified in the reservation, the system must allocate a taxi to answer the request.	• Reservation Manager • DBMS Communication Manager • Active DBMS (creation of triggers) • Queue & Requests Manager
• The system must send a notification to the driver of the allocated taxi cab.	• Communication Manager • Mail Service Interface

5.9 Allow fair management of the queues

• Whenever a driver ends a ride the system must insert him as last in the queue of the zone where he is.	• Position Manager • Position Retriever • Queue & Requests Manager
• The system must place as first in the queue of each zone the driver that in that zone has the highest absolute waiting time	• Queue & Requests Manager
• When a free taxi changes zone the system must automatically find the gps position of the taxi and insert it in the queue of the respective new zone in the right position.	• Position Manager • Position Retriever • Queue & Requests Manager

5.10 Allow taxi drivers to either accept or refuse the assignation and notify customer (in case of acceptance)

• The system must provide an interface to allow the taxi drier to either accept or refuse an assignation.	• Web View Manager • Application Manager
• The system must send a notification to the customer related to the request when a taxi diver accept the assignation.	• Communication Manager
• The system must reinsert the driver as last in the queue if the request is rejected.	• Queue & Request Manager

5.11 Allow taxi drivers to refuse an accepted assignation in case of accidents, problems with the vehicle or other sort of problems, and notify customer with the new assignation

• The system must provide an interface for the taxi drivers to cancel an assignation in case of accidents or other problems.	• Web View Manager • Application Manager
• The system must automatically search for a new taxi to answer the request of the customer.	• Communication Manager • Reservation Manager
• The system must send a notification to the customer with the code of the new assigned taxi.	• Communication Manager • Mail Service Interface

5.12 Allow customer to cancel a reservation

• The system must enable customers to access their previous reservation.	• Web View Manager • Application Manager
• The system must allow to cancel previous reservation when no longer needed.	• Communication Manager • Reservation Manager

5.13 Allow taxi drivers to report fake-users in case a user who has booked a taxi and did not canceled his request does not show up

• The system must provide an interface which allows the drivers to report customers who did not cancel their reservation but they did not show up at the appointment place.	• Web View Manager • Application Manager
• The system will need a function that keep counts of how many times a user is reported as fake.	• User Manager
• After a certain fixed threshold the system must notify the application manager who will take the appropriate countermeasures.	• User Manager • Communication Manager
• The system will automatically reinsert the driver into the queue as if the assignment was never created.	• Queue & Requests Manager

5.14 Do not allow customer to take the wrong taxi or taxi driver to pick up the wrong customer

• With the notification of assignation the system must also send to the customer a unique code associated to the taxi which is going to provide the ride.	• Mail Service Interface • Reservation Manager • Communication Manager
• The system must allow taxi driver to verify the correctness of the code.	• Web View Manager • Application Manager
• The system must check that the code inserted by the taxi driver matches the code generated by the system itself for that particular ride.	• Reservation Manager

5.15 Allow the development of applications that need to interface with myTaxiService through APIs

• The system must provide programmatic interfaces to allow other application to access user data (only read, no modification) and user's functionalities	• API Manager • Reservation Manager (in the API Manager component)
• The system must provide programmatic interfaces to allow users to use the same log-in credentials in other systems which interface with our system.	• Authentication Manager
• The system must provide programmatic interfaces to allow other applications to access the list of reservation (only read, no modification)	• Reservation Manager (in the API Manager component) • DBMS Communication Manager