

**POLITECNICO DI MILANO**  
**Computer Science and Engineering**  
**Project of Software Engineering 2**



**myTaxiService**

**Design Document**  
**Ver. 1.0**

**Release date: December 4, 2015**

Authors: Simone Rosmini (853949)  
Vincenzo Viscusi (858689)  
Matteo Zambelli (776162)

Reference Professor: Mirandola Raffaella

## TABLE OF CONTENT

<b>1. INTRODUCTION.....</b>	<b>3</b>
<b>1.1 Purpose.....</b>	<b>3</b>
<b>1.2 Scope.....</b>	<b>3</b>
<b>1.3 Definitions, Acronyms, Abbreviations.....</b>	<b>3</b>
<b>1.4 References.....</b>	<b>4</b>
<b>1.5 Document Structure.....</b>	<b>4</b>
<b>2. ARCHITECTURAL DESIGN.....</b>	<b>5</b>
<b>2.1 Overview.....</b>	<b>5</b>
<b>2.2 High level components and their interaction.....</b>	<b>5</b>
<b>2.3 Component view.....</b>	<b>6</b>
<b>2.4 Deployment view.....</b>	<b>9</b>
<b>2.5 Runtime view.....</b>	<b>10</b>
<b>2.6 Component interfaces.....</b>	<b>11</b>
<b>2.7 Selected architectural styles and patterns.....</b>	<b>13</b>
<b>3. ALGORITHM DESIGN.....</b>	<b>17</b>
<b>4. USER INTERFACE DESIGN.....</b>	<b>21</b>
<b>5. REQUIREMENTS TRACEABILITY.....</b>	<b>24</b>
<b>6. USED TOOLS.....</b>	<b>29</b>

# 1. INTRODUCTION

## 1.1 Purpose

The purpose of this document is to provide documentation for the design and implementation of the myTaxiService's modules system. Both high-level requirements and implementation details are documented here to ensure successful completion of the project and continuity for future project development.

## 1.2 Scope of this document

This document is intended to be a detailed design supplement to the *Requirements Analysis and Specification Document* (RASD), and therefore does not replace it. This document will provide detailed specifications for designing, implementing, and configuring software for myTaxiService system, but will not include end-user documentation.

## 1.3 Definitions, Acronyms, Abbreviations

Term	Description
USER	For user we mean a person already registered in the system. A user has a profile that includes all these information: Name, Surname, Email, Username, Password, Payment Card And optionally a Picture. A user has the capability of doing everything that we mentioned in the goals, making exception for accept or decline the calls.
TAXI DRIVER	A taxi driver is a person that work for taxi's company and drive the taxi during his worktime. A taxi driver has the capability of accept or decline the calls.
GUEST	A guest is a person who hasn't signed up yet. Guests have less power in the system than users, they don't have the users' skills, and the only function they can use is to sign up.
CALL	Each request done by a user for a ride (real time, shared or single).
RIDE	"Ride" is the term used to indicate all the information that is sent to taxis to satisfy the call of one or more customers . One ride corresponds to a specific route

	created with the departure and destination points of each passenger associated with the ride .
ROUTE	A route is the itinerary traveled by the taxi drivers to reach the destinations indicated by passengers.
TAXI SHIFT	We call taxi shift the movement of a taxi that change its zone for compensate the movement of another taxi in the reverse path.
(PROCESS OF ) TAXI ALLOCATION	When a user call a taxi with the app the taxi driver first in the queue of his zone receive a notification and have to accept or decline the call, if accept the call finish when taxi driver bring the user at the destination, but if he declines, the call pass through the queue until one taxi driver accept it.
TAXI DRIVER'S STATES	A taxi driver with his application can set its status on three modes : 1. Available : when he is waiting a call and can accept it; 2. Busy : when he is driving for another call and can't serve another one; 3. Not available : when for personal reasons he can't serve a taxi call.

## 1.4 References

- RASD
- Specification Document: Assignments 1 and 2.pdf
- DD T TOC.pdf
- IEEE Standard for information technology-System design-Software Design Descriptions
- Software Engineering: Principles and Practice (Hans Van Vliet)

## 1.5 Document Structure

The section 2 explains the components in which the system will be divided and how they interact, the sequence diagrams of the main use cases and the type of architecture that will be used.

The section 3 presents the most significant algorithms, while in Section 4 are presented some mockups for the interface that will be seen by the taxi drivers .

In section 5 the components presented in section 2 are associated with the requirements presented in the RASD.

## 2. ARCHITECTURAL DESIGN

### 2.1 Overview

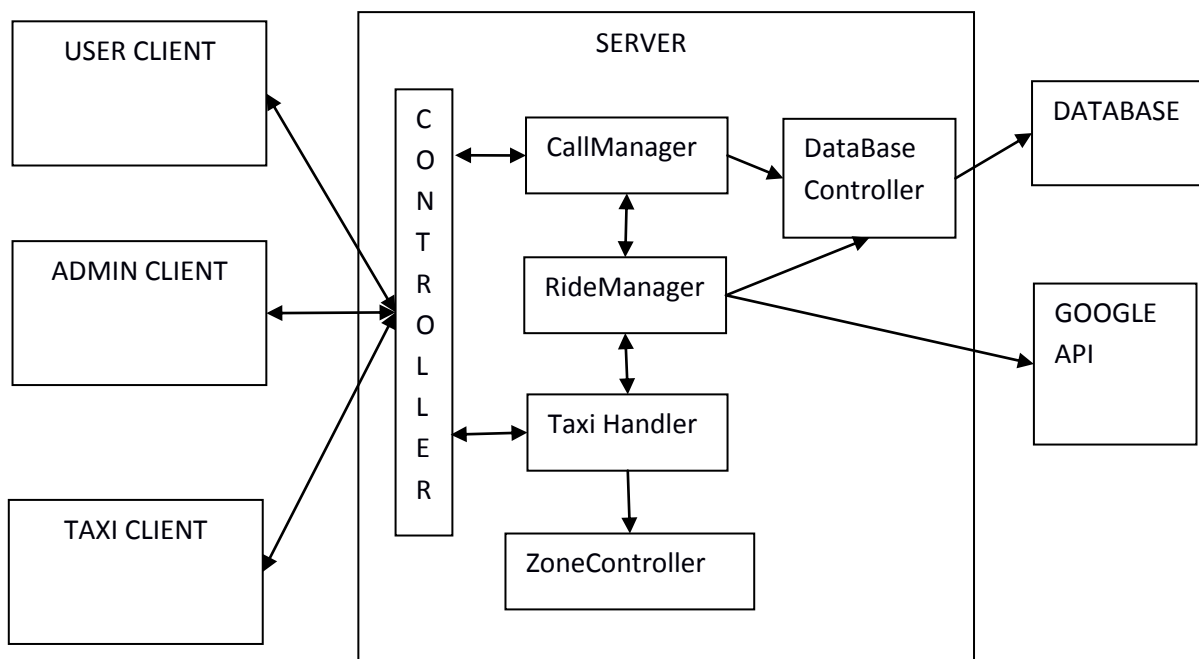
We decide to adopt a top-down approach at least at this point of the project.

Maybe, once defined the sub-systems, we will adopt a bottom-up approach in order to create more reusable components.

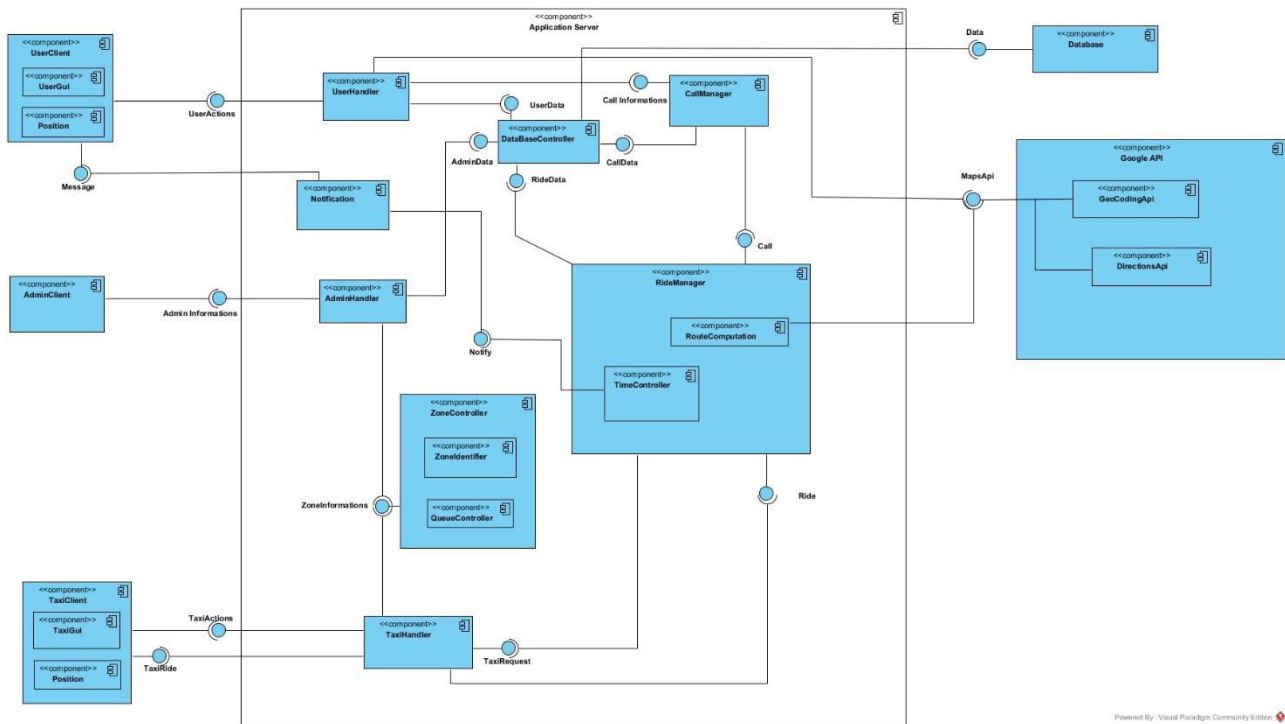
So we think it is now necessary to decompose our system into other sub-systems, in order to make it easy to understand the issues that we will find in implementing functionalities and to separate, logically, groups of functionalities and state clearer their interaction.

### 2.2 High level components and their interaction

We have decided to divide the system in the following subsystems:



## 2.3 Component view



### DESCRIPTION OF THE COMPONENTS:

- **UserHandler:**
  - **Provided Interfaces:** UserActions
  - **Required Interfaces:** UserData, CallInformations, MapsApi

Description:

It's the controller for the user clients, it provides the switching of the client requests toward the components necessary to satisfy the specific request. In particular it interacts with different components: with the **DataBaseController** component to permit the update of the user profile data, with the **CallManager** component to which it forwards the call coming from the users and with the **GoogleApis** component to obtain the information regarding the position of the client like the actual address of the client given his GPS coordinates.

- **CallManager:**
  - **Provided Interfaces:** CallInformations
  - **Required Interfaces:** CallData, Call

Description:

It takes care of calls processing and the consecutive forwarding to the **RideManager** component in order to get a Ride corresponding to each call. After that it provides saving the call information in the database and remains available for the client to send the all the call details that he requires.

- **RideManager:**
  - **Provided Interfaces:** Call, Ride,
  - **Required Interfaces:** Notify, RideData, MapsApi, TaxiRequest

Description:

It takes care of associating a Ride for each call received from the CallManager component and creating a route for that ride using the **RouteComputation** component witch is set to communicate with Google geolocation api. Its work includes fiding the best way to join different Rides to create a shared route following the time requirements imposed by the users during the reservation phase (MaxDelay). In the requirements we have some time constraints concerning the allocation of taxis (10 min before the ride) or the notifications to send to the clients, so it is necessary the **TimeController** component witch takes care of measuring the actual time and synchronizing the different actions in respect to a given timetable.

- **Notification:**
  - **Provided Interfaces:** Notify
  - **Required Interfaces:** Message

Description:

It takes care of sending the notifications to the client when asked by the TimeController component in the RideManager by using the client application or by sending emails.

- **ZoneController:**
  - **Provided Interfaces:** ZoneInformations

Description:

It provides all the necessary information about the zones like returning the zone corresponding to a given address or returning the number of taxis for each zone. To do that it needs the **QueueController** module which provides all the typical operations on the queues, in particular it provides to the TaxiHandler component the first taxi in the queue of a given zone to which will be sent the request for a ride.

- **TaxiHandler:**
  - **Provided Interfaces:** TaxiActions, TaxiRequests

- **Required Interfaces:** TaxiRide, Ride

Description:

It's the controller for the taxi clients, so it takes care of processing all the requests coming from the taxi and if necessary forward them to the RideManager component for example to modify the status of a ride or notifying the get on or the get off of a passenger. On the other side it provides allocating a taxi for the ride requests coming from the RideManager component by interacting with the ZoneController component and forwarding the ride request to the correct taxi client.

- **TaxiClient:**

- **Provided Interfaces:** TaxiRide
- **Required Interfaces:** TaxiActions

Description:

Represents the application for the Taxi clients. It needs some mandatory components to respect the requirements. First of all the **TaxiGUI** component which allows the taxi drivers to interact with the system to accomplish some basic operations like accepting or refusing the rides or to notifying the get on and get off of the passengers or changing the taxi status. It also needs the **TaxiPosition** component which gets the position information using a GPS sensor.

- **UserClient:**

- **Provided Interfaces:** UserActions
- **Required Interfaces:** Message

Description:

Represents the application for the user clients. There are different necessary component to satisfy the requirements of the taxi service: the userGUI component which allows the user to interact with the system by sending requests to the server and receiving notifications; the Position Component that takes care of providing the position information of the client using a gps sensor in case of mobile application or through network localization in case of web application.

- **AdminHandler:**

- **Provided Interfaces:** AdminInformations
- **Required Interfaces:** AdminData

Description:

It's the controller for the admin client. It interacts with the ZoneController to show the zone information to the admin and allows him to modify the zone parameters if needed. It can also access the database to see and modify other types of data (Future Implementation)

- **AdminClient:**

- **Required Interfaces:** AdminInformations

Description:



This component allows the AdminClient to view all the informations about the zones and modify the number of taxis for each zone.

- **GoogleApi:**

- **Provided Interfaces:** MapsApi

Description:

Provides all the maps processing services of Google maps, in particular contains the **DirectionsApi** component which returns the fastest or the shortest path that passes through a given set of addresses and the **GeocodingApi** component which returns the address corresponding to some GPS coordinates an vice versa.

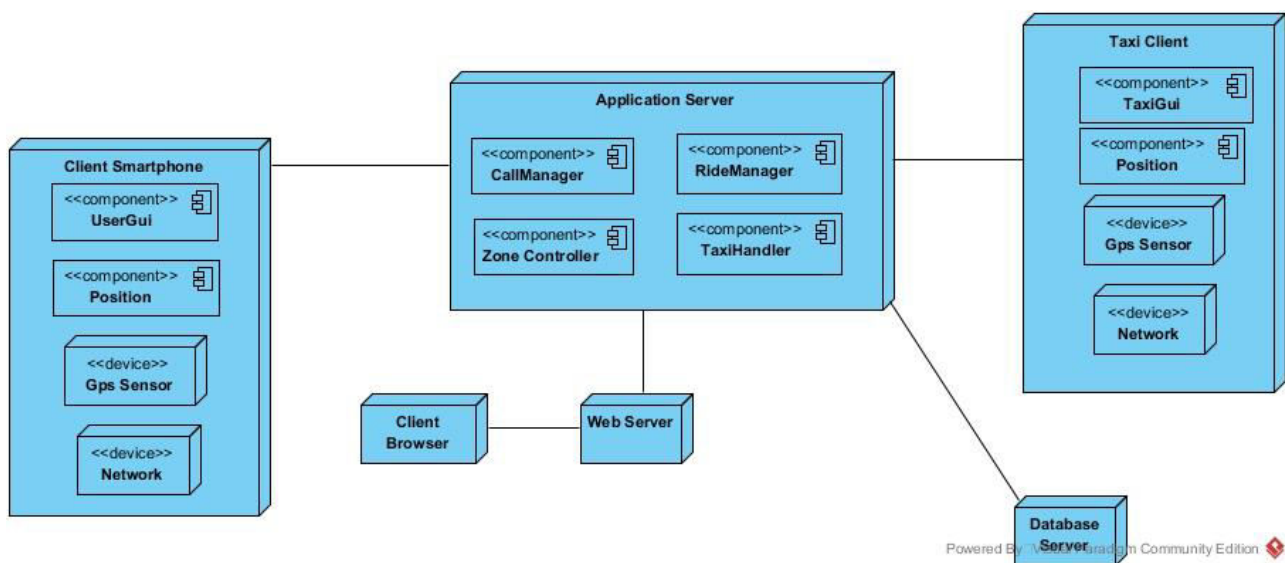
- **DataBaseController:**

- **Provided Interfaces:** RideData, CallData, RideData, AdminData
- **Required Interfaces:** Data

Description:

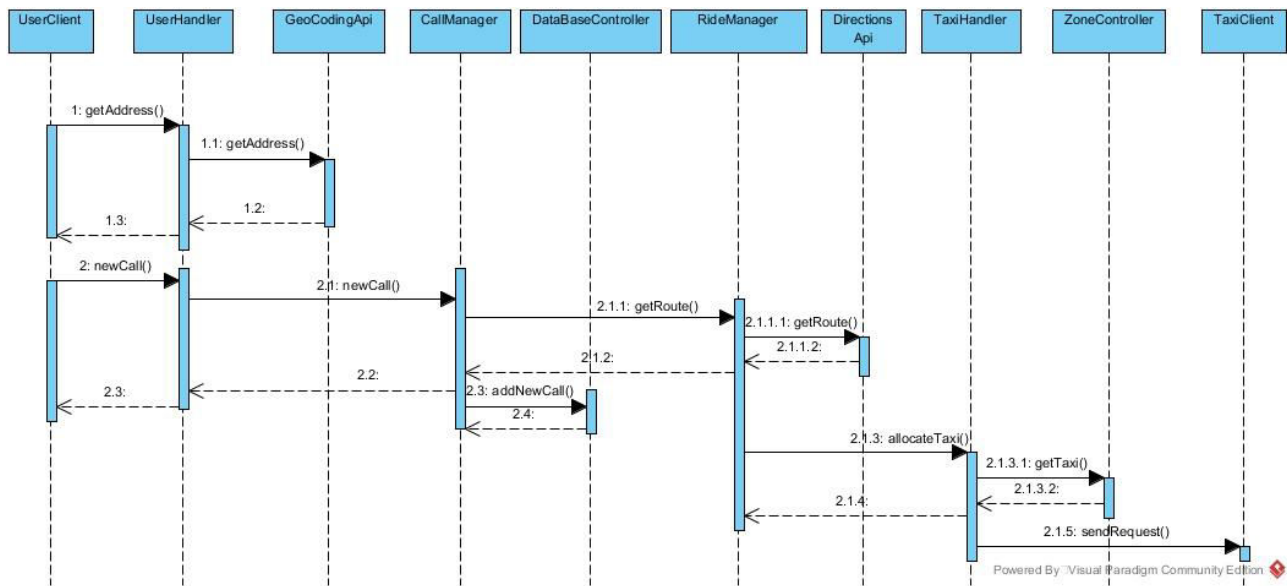
It takes care of sending all the query requests to the Database in order to provide to the other components the adding or the updating of the data.

## 2.4 Deployment view

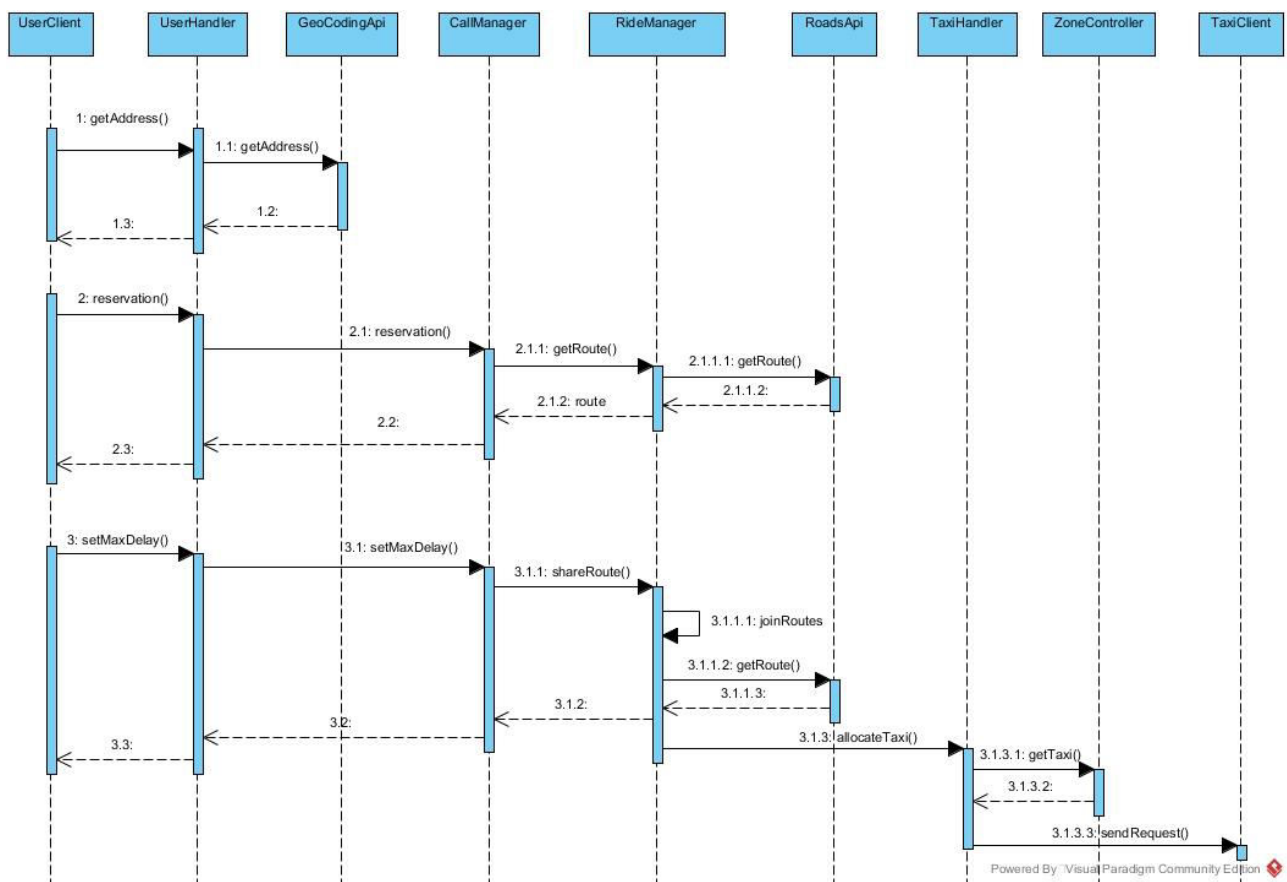


## 2.5 Runtime view

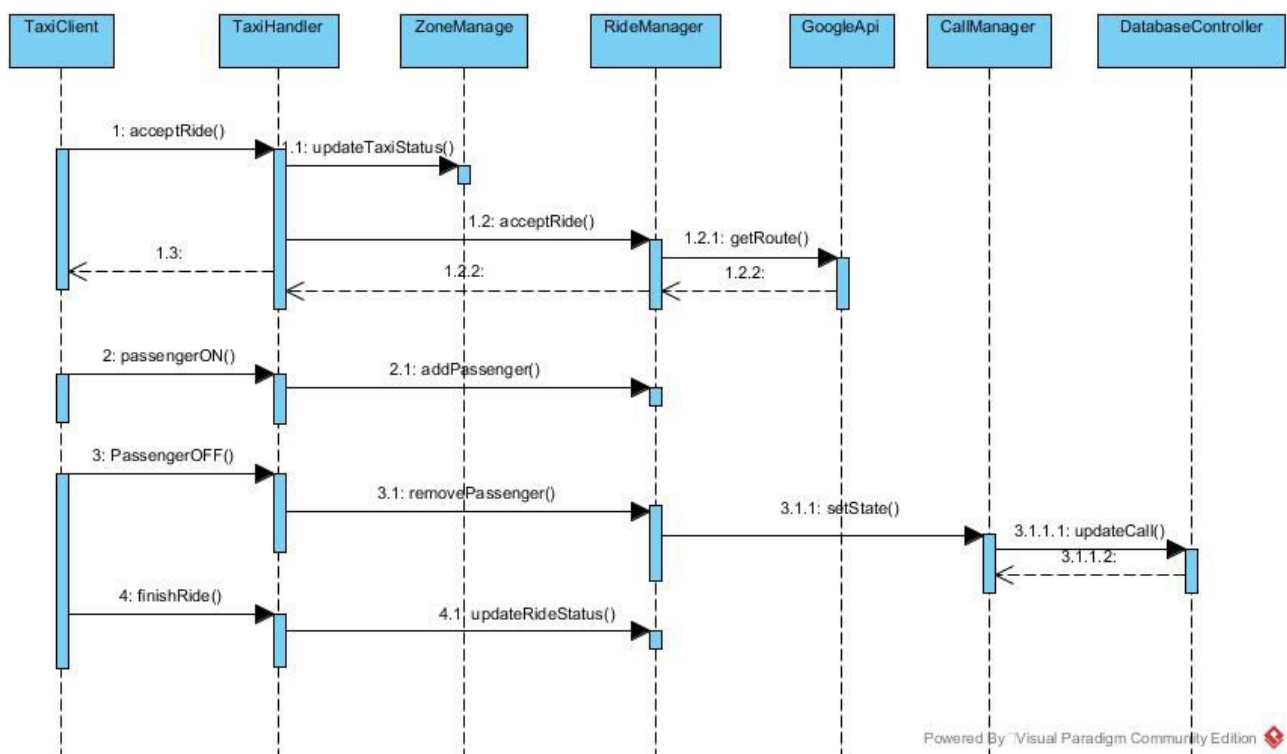
The following sequence diagram describes the elaborating process of a Real Time Call from the client request to the receiving of the ride to the taxi:



The following sequence diagram describes the elaborating process of a Reservation Call from the client request to the receiving of the ride to the taxi:



The following sequence diagram describes the process of serving a ride by a taxi from the fase of accepting the ride to the end of the ride.



## 2.6 Component interfaces

This sections includes the description of the provided interfaces of each component in a pseudolanguage.

### UserHandler:

- UserActions:
  - o Boolean login(Username, Password)
  - o Boolean registration(User Info)
  - o Route newCall(CallData)
  - o Address getLocalAddr(gps Coordinates)
  - o UserInfo getUserInfo(User ID)
  - o Boolean updateUserInfo(UserInfo)
  - o CallList viewHistory (User ID)
  - o Route getRoute(Call ID)
  - o Addr getTaxiPosition(Call ID)
  - o CallInfo getCallInfo(Call ID)
  - o Boolean removeCall(CallID)

### CallManager:

- CallInformations:
  - o CallList vewHistory(User ID)
  - o Route newCall(CallData)
  - o Route getRoute(Call ID)
  - o Addr getTaxiPosition(Call ID)
  - o CallInfo getCallInfo(Call ID)
  - o Boolean removeCall(Call ID)
  - o Void shareCall(Call ID, maxDelay)

### RideManager:

- Call:
  - o Route getRoute(CallInfo)
- Ride:
  - o Void passengerOff(User ID, Taxi ID, Ride ID)
  - o Void passengerOn(User ID, Taxi ID, Ride ID)
  - o Route acceptRide(Ride ID, Taxi ID, TaxiPosition)
  - o Void refuseRide(Ride ID, Taxi ID)
  - o Void UdateRideStatus(Ride Info)

### TaxiHandler:

- TaxiActions:
  - o Void passengerOff(User ID, Taxi ID, Ride ID, GpsCord)
  - o Void passengerOn(User ID, Taxi ID, Ride ID, GpsCord)
  - o Route acceptRide(Ride ID, Taxi ID, TaxiPosition)

- Void refuseRide(Ride ID, Taxi ID)
- Void finishRide(Ride ID, Taxi ID)
- Void setState(Taxi ID, State)
- TaxiRequest:
  - TaxiID allocateTaxi(RideInfo)
  - GpsCord getPosition(TaxiID)

#### **ZoneController:**

- Zone:
  - Zone getZone(Address)
  - ZoneInfo getZoneInfo(ZoneID)
  - Taxi getTaxi(Zone ID)
  - Void addTaxi(Zone ID)
  - Void updateTaxiStatus(Taxi ID)
  - Zone getTaxiZone(Taxi ID)

#### **Notification:**

- Void sendNotification(User ID, Notification)
- Void sendEmail(User ID, Notification)

#### **DatabaseController:**

- UserData:
  - Boolean newUser(UserInfo)
  - Boolean updateProfile(UserInfo)
  - Boolean removeProfile(User ID)
- RideData:
  - Boolean updateRideStatus(RideInfo)
  - RideList getSharedRides(Zone, TimeInterval)
- CallData:
  - Boolean addNewCall(CallInfo)
  - Boolean updateCall(CallInfo)
- AdminData:
  -

#### **GoogleApi:**

- MapsApi:
  - Addr getAddr(Gps Coordinates)
  - GpsCord getPosition(Addr)
  - Route getRoute(AddrList)

#### **UserClient:**

- InputMessages:
  - Void sendNotification(Notification)

#### TaxiClient:

- TaxiRide:
  - o Void sendRequest(Ride)
  - o GpsCoord getPosition(TaxiID)

## 2.7 Selected architectural styles and patterns

### Architecture style: 4-Tier

For our software we chose 4 different layers in which we separate the different logical parts:

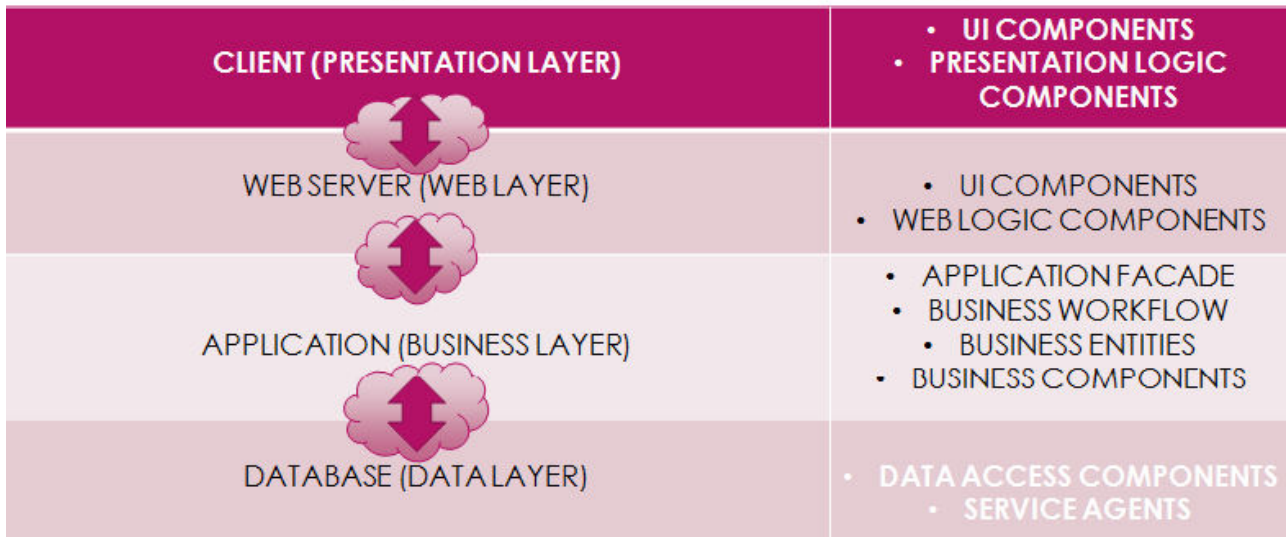
- ▶ Presentation layer
- ▶ Web layer
- ▶ Business layer
- ▶ Data layer

We have segregated functionality into separate segments with each segment being a tier located on a physically separate computer and communication between tiers is made through network.

The main benefits of the 4-tier architectural style are:

- **Maintainability**, because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole;
- **Scalability**, because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward;
- **Flexibility**, because each tier can be managed or scaled independently, flexibility is increased;
- **Availability**, applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

# ARCHITECTURE STYLE OVERVIEW



## ➤ Presentation layer:

This layer contains the user oriented functionality responsible for managing user interaction with the system, and generally consists of components that provide a common bridge into the core business logic encapsulated in the business layer passing through Web layer.

It is composed by :

- **UI-Components** that are the application's visual elements used to display information to the user and accept user input;
- **Presentation logic components** is the application code that defines the logical behavior and structure of the application in a way that is independent of any specific user interface implementation.

## ➤ Web layer

This layer defines and implements the web service, it has two main components :

- **UI components** : provide a presentation graphic for the browser requests and a small part of the web logic for the actions that a client can do (e.g. : tell at the web logic components that the user wants do a registration);
- **Web logic components** : It is a component that knows the structure of the web data and business data and matches them for the management of the requests and the replies.

#### ➤ **Business layer**

This layer implements the core functionality of the system, and encapsulates the relevant business logic.

It is composed by :

- **Application façade** provides a simplified interface to the business logic components, It reduces dependencies because external callers do not need to know details of the business components and the relationships between them;
- **Business logic** is defined as any application logic that is concerned with the retrieval, processing, transformation, and management of application data;
- **Business entities** that encapsulate the business logic and data necessary to represent real world elements within your application;
- **Business workflow** components define and coordinate long running and multistep business processes.

#### ➤ **Data layer**

This layer provides access to data hosted within the boundaries of the system, and data exposed by other networked systems; perhaps accessed through services. The data layer exposes generic interfaces that the components in the business layer can consume.

It is composed by:

- **Data access components** abstract the logic required to access the underlying data stores;
- **Service agents** : when a business component must access data provided by an external service, you might need to implement code to manage the semantics of communicating with that particular service and in our case we manage data provided by GoogleMapsApi.

### **Matching our Application with Architecture style**



► **Presentation layer** : in this layer the clients can :

- **USER** : make a real time taxi call, book a shared or single ride, view data of profile, view data of taxi race history, make a registration and do a login;
- **TAXI** : accept or decline a real time taxi request, change the taxi status, display the route of the ride, display the passengers of a ride, change the passengers status and login through a card with a barcode;
- **ADMIN** : change data of some users(USER or TAXI) or display some statistic data.

► **Web layer**: in this layer is contained the **web server**, it receive request from clients(mobile application and browser) and make a response at them through the network. It is a bridge between user and application, adapts the data structure and manages the requests and the replies between the presentation layer and business layer.

► **Business layer**: here there is the main core software application (a small example of code in algorithm part). In this layer :

- are managed all the system's requests made by clients (USER, TAXI AND ADMIN);
- are stored all the data structure (ZONE, TAXI, USER, RIDE, CALL ,...);
- are managed all database communications with a call at data layer;
- are managed the communication with external software(in our case with Google Api for the route of a ride);
- is decided the workflow of a reply or internal procedure between components.

► **Data layer** : here there are the software modules to access the physical database:

- it dialogs only with the business layer and reply only at it;
- it knows the structure of the tables in the db and match the logical data structures with them (e.g. : data fields in a logical structure with their respective fields in a certain table);
- it manages all the queries at the db and their replies for the business layer (e.g. : store/read data of a clients, data of a ride, ...).

### 3. ALGORITHM DESIGN

#### Matching between calls

```
/*
PercorsoGmaps è una classe delle api di googlemaps che permette di memorizzare
un percorso tra due punti.

PercorsoGmaps CreaPercorso(Position StartPoint, Position EndPoint) è un metodo
chiamato dalla funzione fornito dalle api di google il quale dati due punti ti
ritorna il percorso più veloce con i relativi dati tra di essi.

Assumiamo che la classe Call creata da noi abbia all'interno i campi :
StartPoint, EndPoint, StartTime, EndTime, durata, maxovertime e PercorsoGmaps.

ArrayCall è un array di call shared appartenenti alla stessa zona in cui si
cerca di unirne il più possibile.

il metodo MatchingSharedCalls viene chiamato ogni volta in cui si aggiunge una
shared call all'array.
*/

PercorsoGmaps[] MatchingSharedCalls(Call* ArrayCall)
{
    //variabile in cui vengono memorizzati tutti i percorsi delle call unite
    PercorsoGmaps[] percorsi = new PercorsoGmaps();

    //ordino le calls secondo l'ora di partenza in modo crescente

    for(int i=0; i<ArrayCall.size()-1; i++)
    {
        for(int j=i+1; j<ArrayCall.size(); j++)
        {
            if(ArrayCall[j] != NULL)
            {
                if(ArrayCall[i].StartTime >
ArrayCall[j].StartTime) //ipotizzo che il confronto delle date tenga
conto sia del giorno che dell'ora
                {
                    call* temp = ArrayCall[i];
                    ArrayCall[i] = ArrayCall[j];
                    ArrayCall[j] = temp;
                }
            }
        }
    }

    //controllo quali call sono integrabili, se una call non è integrabile
con la successiva allora non lo sarà nemmeno con tutte le altre

    int IndiceControllo = 0;

    while(i<ArrayCall.size())
    {
```

```

        int i=IndiceControllo;
        int j = i+1;
        while(j<ArrayCall.size())
        {
            PercorsoGmaps PercorsiUniti =
CreaPercorso(Arraycall[i].StartPoint,ArrayCall[j].EndPoint);
            if((Arraycall[i].durata + ArrayCall[i].maxovertime) <=
PercorsiUniti.durata) //i due percorsi sono unibili
            {
                for(int z=j+1;z<ArrayCall.size();z++) //provo ad
unire altre call
                {
                    PercorsoGmaps temp =
CreaPercorso(Arraycall[i].StartPoint,ArrayCall[z].EndPoint);
                    for(int k=0;k<z;k++)
                    {
                        Boolean compatibili = true;
                        if((ArrayCall[k].durata +
ArrayCall[k].maxovertime) <= temp.durata)
                        {
                            //è compatibile con tutte
le precedenti
                        }
                        else
                        {
                            //almeno una delle
precedenti non è compatibile con il nuovo percorso
                            compatibili = false;
                        }
                    }

                    if(compatibili) // se sono compaitibili
allora provo ad unire un'altra call e mi salvo l'attuale
                    {
                        PercorsiUniti = temp;
                    }
                    else //se non sono compatibili con il
nuovo percorso esco direttamente dal ciclo for
                    {
                        z = ArrayCall.size();
                    }
                    IndiceControllo = z;
                }
                percorsi.insert(PercorsiUniti);
            }
            else //i due percorsi non sono unibili e quindi il
percorso i non è unibile di conseguenza con nessun'altro da j in poi
            {
                IndiceControllo++;
            }
        }
    }

    return percorsi;
}

```

## Thread that controls the departure of the taxis

```
/*
```

sarà una classe runnable il cui compito è quello di allocare un taxi disponibile 10 minuti prima della partenza della corsa shared facendo un controllo ogni 10 secondi

suppongo che abbia la visibilità sull'array ritornato da MatchingSharedCalls (chiamato percorsi)  
\*/

```
void ControlSharedRide()
{
    Date[] alloca = new Date(); //memorizza all'interno le date (giorno ora
    e minuti) ai quali dovrà chiamare il metodo per allocare il taxi

    while(true)
    {
        //prima controllo se ci sono state delle modifiche nei percorsi

        if(percorsi.size() == alloca.size())
        {
            //non ci sono stati cambiamenti
        }
        else
        {
            for(int i=0; i<percorsi.size();i++)
            {
                date.insert(percorsi[i].StartTime-10);
                //ipotizzando che il tempo sia in minuti
            }
        }

        //controllo se devo allocare un taxi e nel caso rimuovo la corsa
        perchè già gestita

        for(int i=0; i<percorsi.size();i++)
        {
            if(date[i].equalsTo(System.CurrentDate))
            {
                AllocaTaxi(percorsi[i]);
                percorsi.removeElementAt[i];
                alloca.removeElementAt[i];
            }
        }

        Thread.sleep(10000); //lo metto in attesa per 10 secondi
    }
}
```

## Algorithm which with the position returns the corresponding zone

/\*  
utilizza una libreria esterna che disegna segmenti e crea esagoni così da sapere  
se un punto è interno o no

la nostra mappa è suddivisa in zone esagonali che ricoprono l'intera città

```

i punti relativi ai vertici dei segmenti di una zona sono memorizzati in un file
testuale
*/

int GetZoneFromPosition(Position pos)
{
    FileReader f = new FileReader("vertici.txt");

    BufferedReader b = new BufferedReader(f);

    while(s!=EOF)
    {
        int numZone = 1;
        String s = b.readLine();
        CreaEsagono(s, numZone); //la funzione CreaEsagono prende in
ingresso una stringa con le posizioni dei vertici e l'id da assegnare alla
figura disegnata
        numZone++;
    }

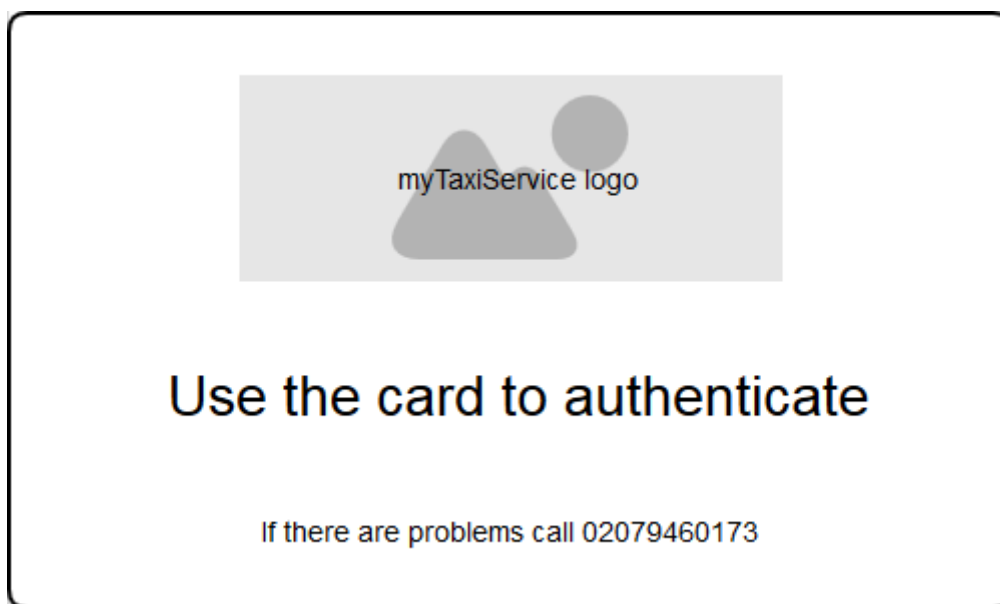
    Return GetIDIn(pos); //la funzione GetIDIn trova a quale esagono
appartiene il punto individuato da Pos e ne restituisce l'id
}

```


## 4. USER INTERFACE DESIGN

Here there are some mockups that give an idea of how the application will appear on the taxi. Some mockups instead relating to the application users are in the RASD.

Login screen



### Status screen



**Name:** Zlatan Rossi  
**ID:** FD34B34JK5Y

**Status:** Available

Available

Unavailable

Log out

### Request screen

**Meeting place:**  
66 Great Queen Street, London

**Time:** 18:15

**Type:** Shareable (10 m.s.)

**Destination:**  
29 Lampton Road, London

Accept

Refuse

 30 seconds remaining



Next meeting screen

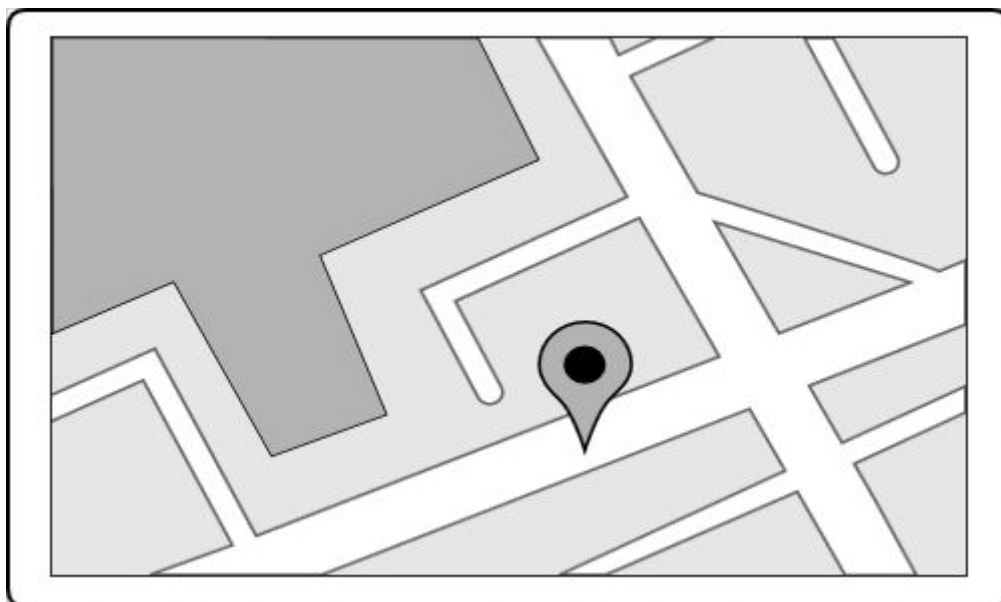


**Next meeting place:**  
66 Great Queen Street, London

**Time:** 18:15

**Passenger:** babbonatale93

Fullscreen map



### Next destination screen



## 5. REQUIREMENTS TRACEABILITY

Requirement	Component
The system should require the user to fill out the following fields: <ul style="list-style-type: none"><li>– Meeting address</li><li>– Destination address</li></ul>	User GUI
The system should allocate a taxi for the request	CallManager RideManager TaxiHandler ZoneController
The system should create a path for the ride and send it to the taxi	RideManager GoogleApi TaxiHandler ClientTaxi
The system should send	RideManager



to the user the code of the taxi that has taken on the ride and the waiting time	CallManager UserHandler UserClient
The system, to create a new reservation, should require the user to fill out the following fields: – Meeting Time – Meeting address – Destination Address – Travel mode – Repetition (No/Daily/Weekly/Monthly)	UserGui
The system should compute the duration of a ride and the expected arrival time	RideManager GoogleApi
The system accepts the reservation only if there is an interval time of at least 15 minutes between the expected arrival time of other scheduled reservations and the meeting time of the new reservation	CallManager RideManager
The system accept the reservation only if there is an interval time of at least 15 minutes between the expected arrival time of the new reservation and the meeting time of other scheduled reservations	CallManager RideManager
The system should associate a tracking number (ID_Reservation) to the Reservation	CallManager
The details of the new reservation are stored only if the reservation is Accepted	CallManager DataBaseController
The system 10 minutes before de meeting time mast allocate a taxi for the reservation.	RideManager(TimeController) TaxiHandler ZoneController ClientTaxi

The system should ask to each user that wants to share a ride, the maximum delay on the estimate arrival time that he wants to accept for sharing the ride.	UserGUI
The system must be able to join different paths to create a single paths that passes in all the start and destination addresses of each path.	RideManager GoogleApi
The system should require the user to fill out the following fields: – E-mail with confirmation – Password with confirmation – Surname and Name – Preferred payment way(credit card or paypal)	UserGUI
The system should require the user to insert a list of the zone from he often takes a taxi	UserGUI UserHandler
The system uses the email as the user identifier	CallManager
The subscription is accepted only if all the fields have been fulfilled and the email doesn't already exists in the database of the registered users	UserGui UserHandler CallManager DataBaseController
The system stores all the user information only if the subscription is Accepted	UserGui UserHandler DataBaseController
The system should send a welcome email to the user with all the necessary information to use the service.	Notification
The system must require the user to fill out the following fields: – E-mail used in the subscription phase	UserGui

– Password used in the subscription phase	
The login is accepted only if the password inserted corresponds with the email in the database of the registered users.	UserGui UserHandler DataBaseController
The user can have access to the service only if the login is accepted	UserGui UserHandler DataBaseController
Every time a taxi starts a ride from a zone A to a zone B then a taxi in the zone B moves towards the zone A.	RideManager TaxiHandler ZoneController
the system should send the details of a ride request to the first taxi in the queue of the zone that contains the start address of the ride: – Tracking number of the ride. – Path of the ride. – Path to reach the start address – Call type (Real Time or Reservation). – Travel Mode (Single or Shared).	RideManager TaxiHandler ZoneController ClientTaxi
The taxi driver taxes on the ride only if he accept the request within 30 seconds from its arrival.	RideManager(TimeController) TaxiHandler ClientTaxi
The system should pass the ride request to the next taxi in the queue only if the taxi driver has refused the request or it passes more than 30 seconds from the arrival of the request.	RideManager(TimeController) TaxiHandler ZoneController ClientTaxi
If the taxi driver refuses the request or he takes more than 30 seconds to accept his taxi is removed from the head of the queue and placed in the	RideManager(TimeController) TaxiHandler ZoneController ClientTaxi

tail.	
If no taxis are available in the zone the request is passed to the first taxi in the queue of a near zone.	RideManager(TimeController) TaxiHandler ZoneController ClientTaxi
The system should be able to show to the taxi administrator a table with the following information: - Actual number of taxi for each zone - Priority of the zones - Request rate for each zone.	ZoneController AdminHandler ClientAdmin
The system should display to the user a calendar with the scheduled reservation.	CallManager UserHandler UserClient(UserGUI)
The system should ask the user to select the day in which he wants to schedule a new reservation or see the details of once already scheduled	CallManager UserHandler UserClient(UserGUI)
The system, to create a new reservation, should require the user to fill out the following fields: – Meeting Time – Meeting address – Destination Address – Travel mode – Repetition (No/Daily/Weekly/Monthly)	CallManager UserHandler UserClient(UserGUI)
The system should compute the duration of a ride and the expected arrival time	RideManager GoogleApi
The system accepts the reservation only if there is an interval time of at least 15 minutes between the expected arrival time of other scheduled reservations and the meeting time of the new	CallManager RideManager

reservation	
The system accept the reservation only if there is an interval time of at least 15 minutes between the expected arrival time of the new reservation and the meeting time of other scheduled reservations	CallManager RideManager
The details of the new reservation are stored only if the reservation is accepted	CallManager DataBaseController

## 6.USED TOOLS

Microsoft Office: to write this document.

Gliffy.com: to create the sequence diagrams.

moqups.com: to create the mockups.

### Hours of work:

Simone Rosmini about 24 hours

Vincenzo Viscusi about 24 hours

Matteo Zambelli about 24 hours