Politecnico di Milano

Computer Science and Engineering

Project of Software Engineering 2

MyTaxiService

Design Document

Authors:

Antonio Iannacci - 854157

Daniele Romanini - 854732

Federico Seri - 854032

Reference Professor: Mirandola Raffaela

TABLE OF CONTENT

1. INTRODUCTION

1.1. Purpose

This document concerns the Design of the application "myTaxiService".
The purpose is to describe the implementation choices done in order to satisfy the functional requirements specified in RASD, respecting imposed constraints and also by reaching the Quality of Service required.
The organization of information and data is specified and every choice is justified.
Every interaction and association among objects is described, starting from a high-level point of view, going to a more-detailed one. Interactions already described in the RASD are specified here with a deeper level, specifying every single object involved, obviously respecting the previous vision.

1.2. Scope

This document is addressed mainly to developer that should respect the design constraints and the architecture described in this document.
The future project manager could benefit from this document in order to organize the development process, since the principle "divide et impera" was applied while thinking about the design. Interfaces are present in the application, so a programmer could program to the interface of another module, instead of to the module itself, and then the module could be merged.
The programmer team is free to use the desired programming language, given that they respect the design constraints specified.
Writing the document, we have also thought about the interaction between the various users that will use the application and the system itself.
Each future extension should refer to this document, where also the external provided API are specified.

1.3. Acronyms

We use the following Acronyms in the document:

- **TD**: Taxi driver;
- **SS**: Shared Set;
- **SC**: Shared Call;

1.4. Reference documents

In order to write this documents, we follow the "Template for the design document" (*Software Engineering II, AA 2015-2016, Politecnico di Milano*) provided by our commissioner.
The other documents we refer are the following International Standards:

- *IEEE Standard for Information Technology—Systems Design—Software Design Descriptions (1016 - 2009)*
- *Systems and software engineering – Architecture Description (ISO/IEC/IEEE 42010)*

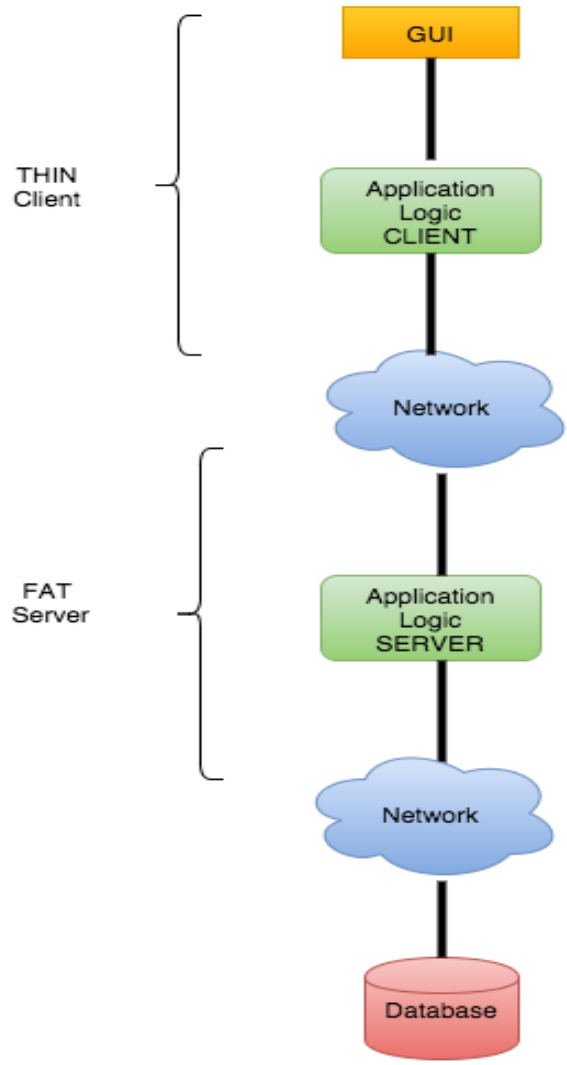
1.5. Document structure

In the document we analysed the Architectural Structure of the System, starting from a high level analysis, until the detailed view of each module, with its functionality and interfaces.
Moreover, it is explained how the system will be deployed, and its run-time behaviour, focusing on the different instances of the modules and the interaction among the various components.
The design-choices and the pattern used are then explained and justified, trying to be the most complete possible.
After that, there is an analysis of the main algorithms of the system, from an implementation viewpoint, in order to be very clear about the behaviour of the important functionality.
Then we present how the main user interface will look both in the mobile and web application.
Finally, we mapped the design elements specified in this document with the requirements defined in the RASD, showing that we have respected the assumptions made and the application is consistent with the assignment and the specification provided.

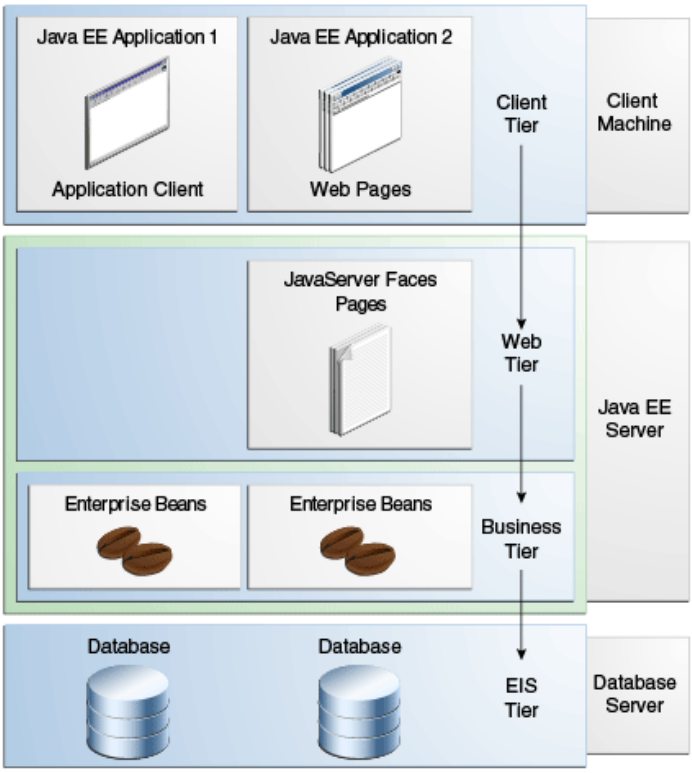
2. ARCHITECTURAL DESIGN

2.1. Overview

We choose to use a three-tier Client-Server architecture for the application, with "thin-clients" and a "fat-server".
In fact, the business logic is fully contained in the Server, while Client provides just a GUI, and few modules containing basic information.
The role of the network is fundamental: the Server should elaborate each information provided by the Client.
The Server accesses the Database through apposite interfaces. The Database could be located on a different machine.
The presentation is distributed: there are 3 different types of users that can access to the system, and, for each one, a personal user interface is provided, with different functionalities.
The adopted style for the Design is the "Object-Oriented" style.



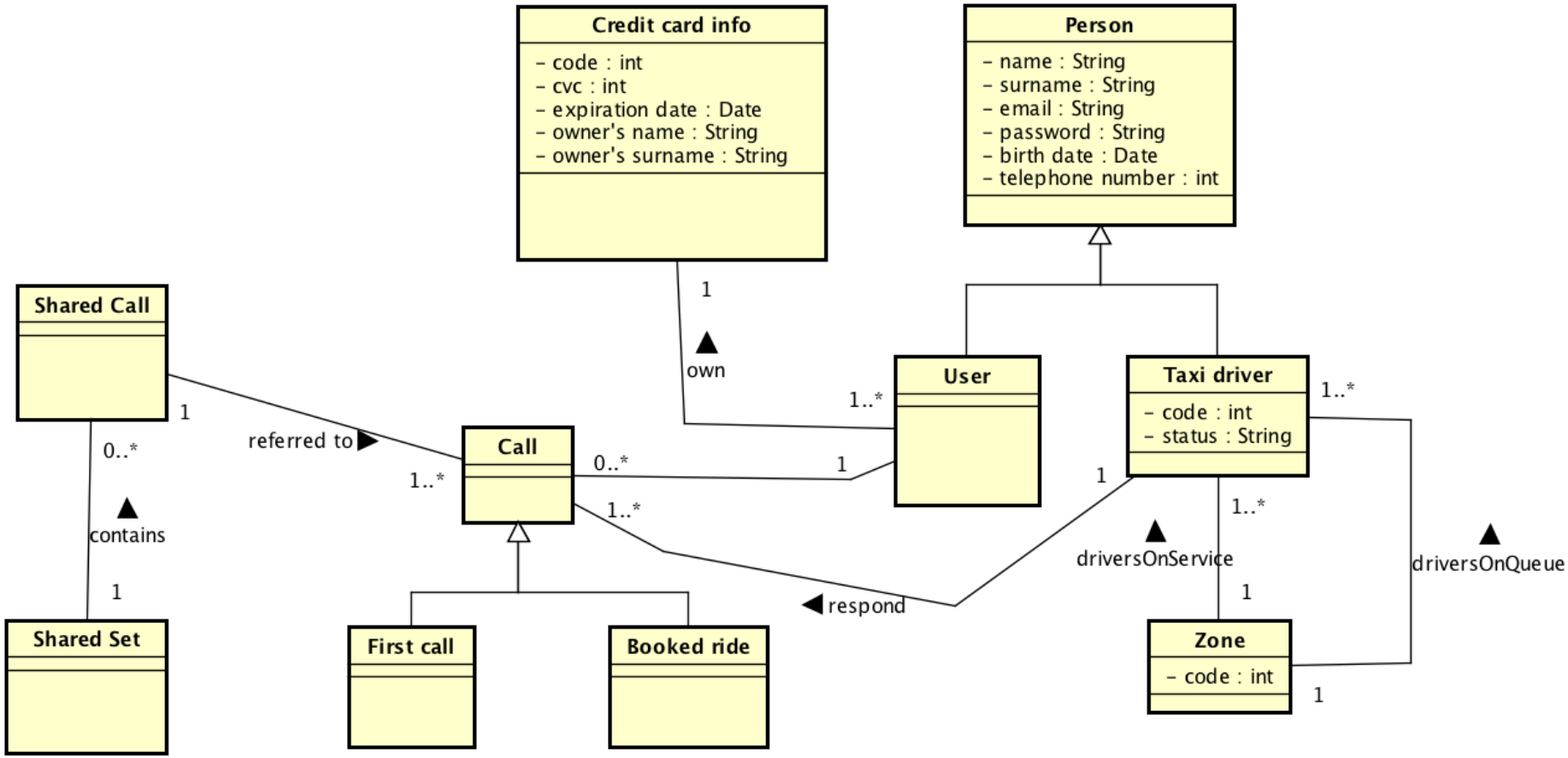
We choose to adopt the JEE framework to develop the application. As known, this uses a distributed multitiered architecture. Adopting this framework, the tiers are 4, instead of 3, because on the server side we have: Web Tier and Business Tier.



Following, we are going to view (from an higher level to a lower level) the main components and classes of the Server Side.
In the Component View section the interaction among server-components, client-components and external-components is illustrated.

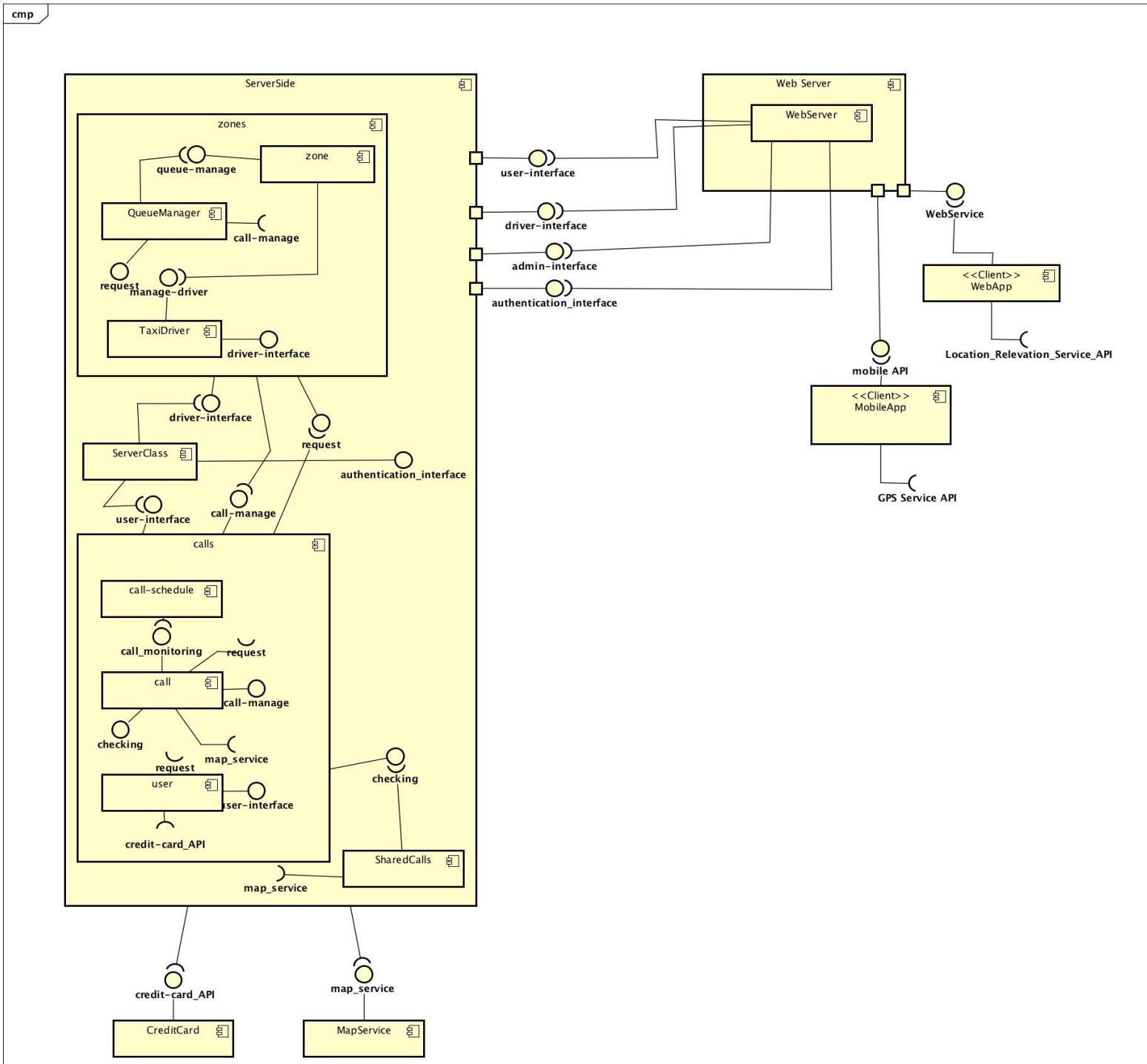
Detailed architectural and design decisions are illustrated in section 2.8.

2.2. High level components and their interaction

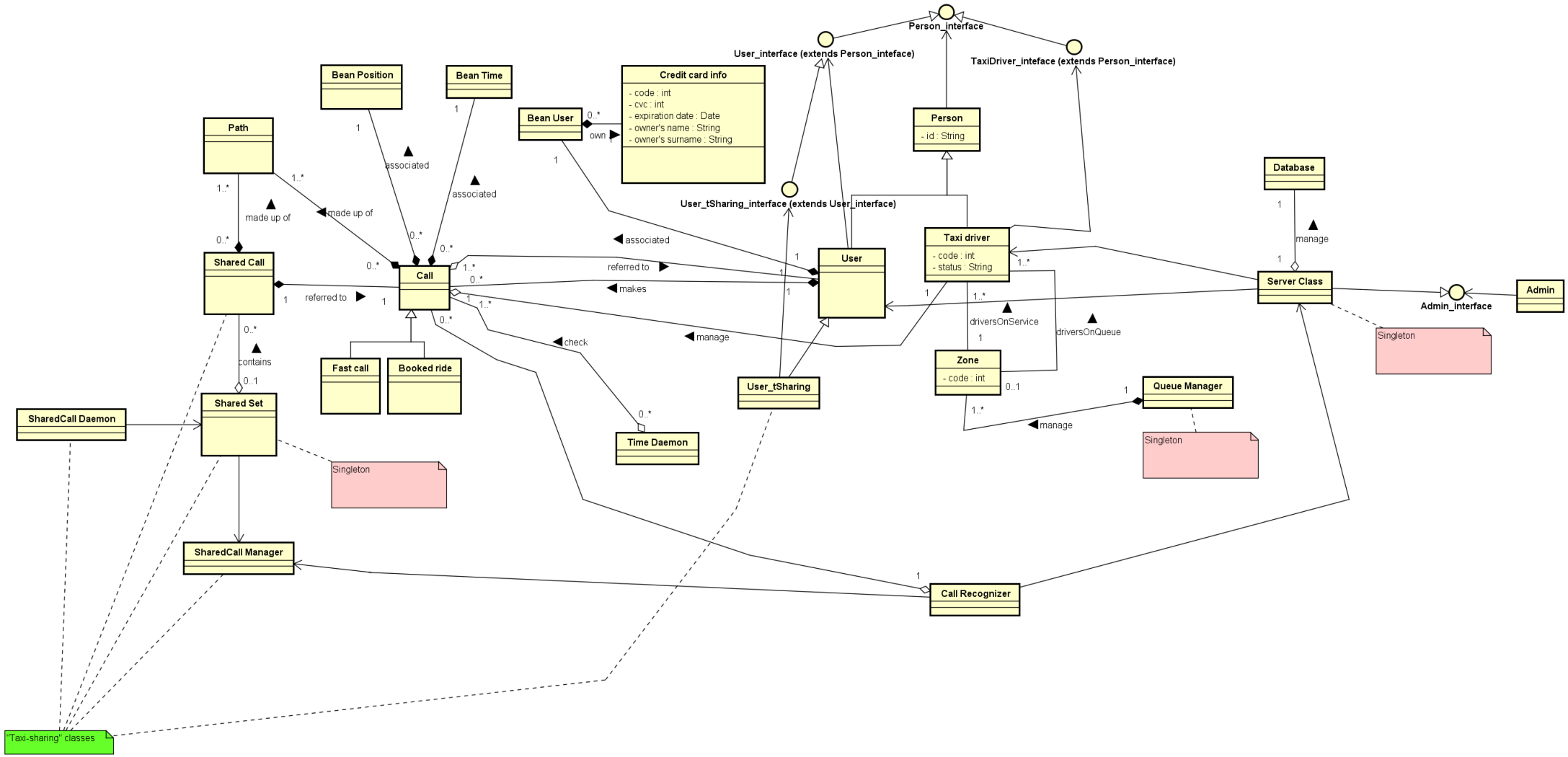


2.3. Component view

In the following diagram, we can see the system divided in modules and the various interactions among them. Every module is composed by components that interact each other in the same module but also with external ones.



2.4. Class diagram (lower level)

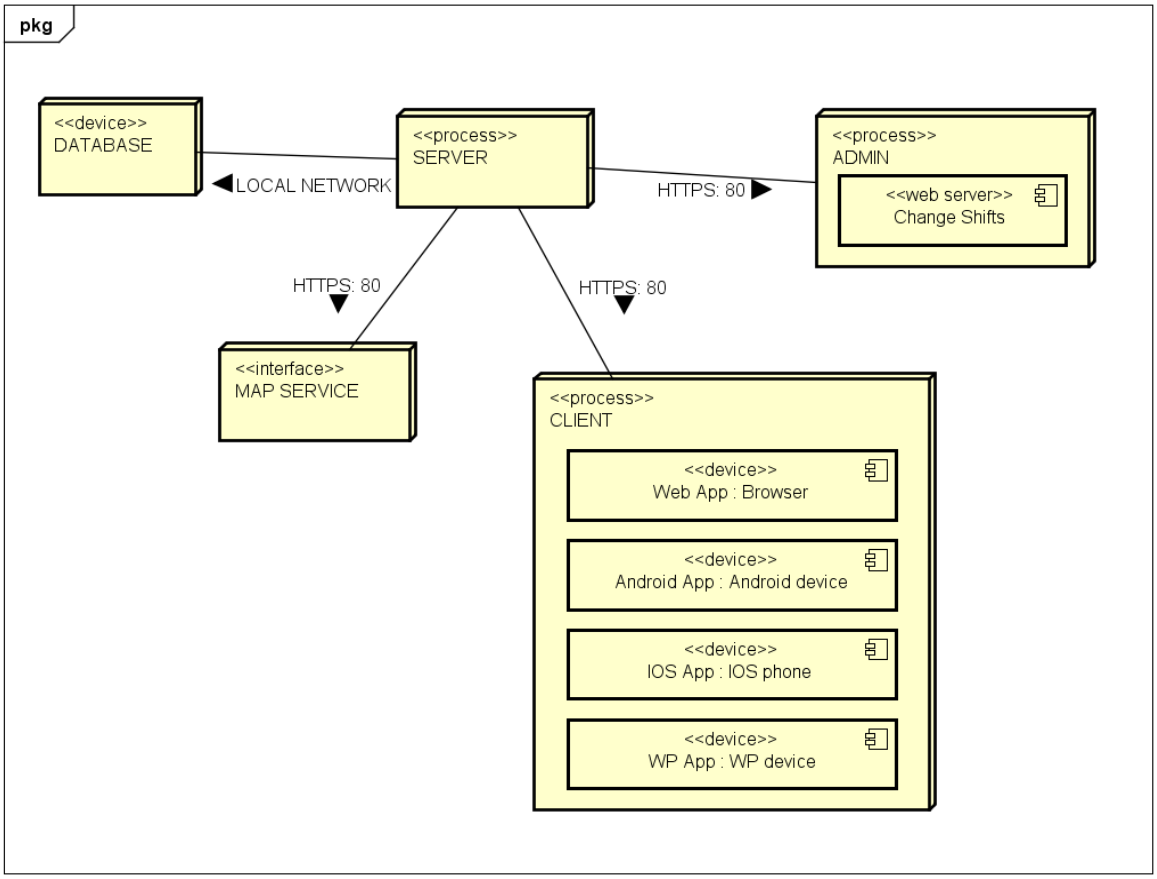


2.5. Deployment view

In this section we report the Deployment Diagram that represent on which the components are installed and executed. As we can see there are three types of nodes: device, process and interface.

"Device" node: the node is a physical computing resource with processing memory and services to execute software;
"Process" node: it is a software computing resources that runs within an outer node and which itself provides a service to host;
"Interface" node: it is a set of functions that helps us to manage some objects of the program.

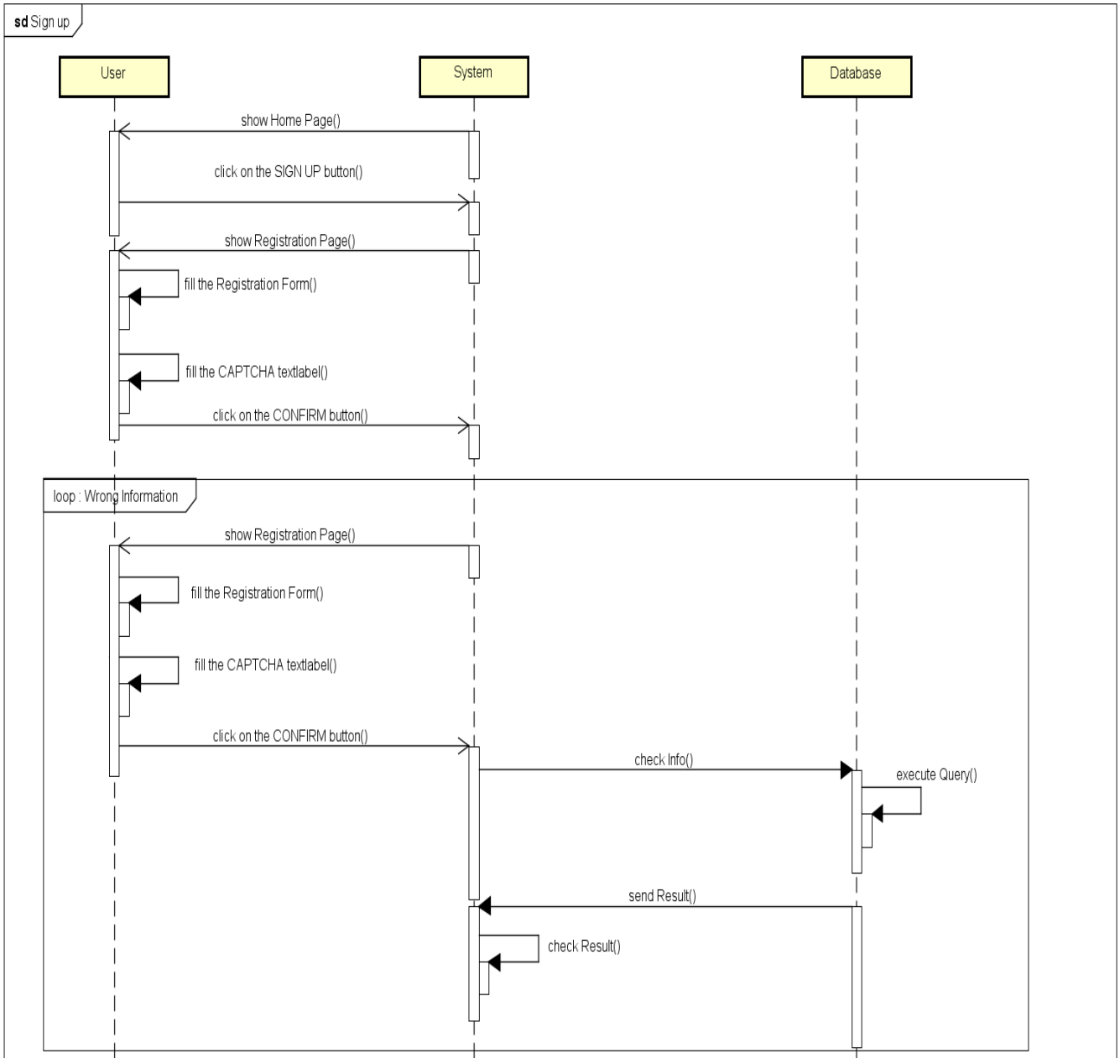
Client, Admin and MapService communicate with the server through the Internet in particular through the protocol "https" and the port "80". Instead, Database is connected to the server through a local network. Client can be installed on different kind of devices such as: Android devices, IOS devices, Windows Phone devices. It is possible to access to the system through the Browser at the address "www.mytaxiservice.com". The Admin can manage the taxi driver's shifts thanks to the function "change shifts" and it is installed on a Web Server device.

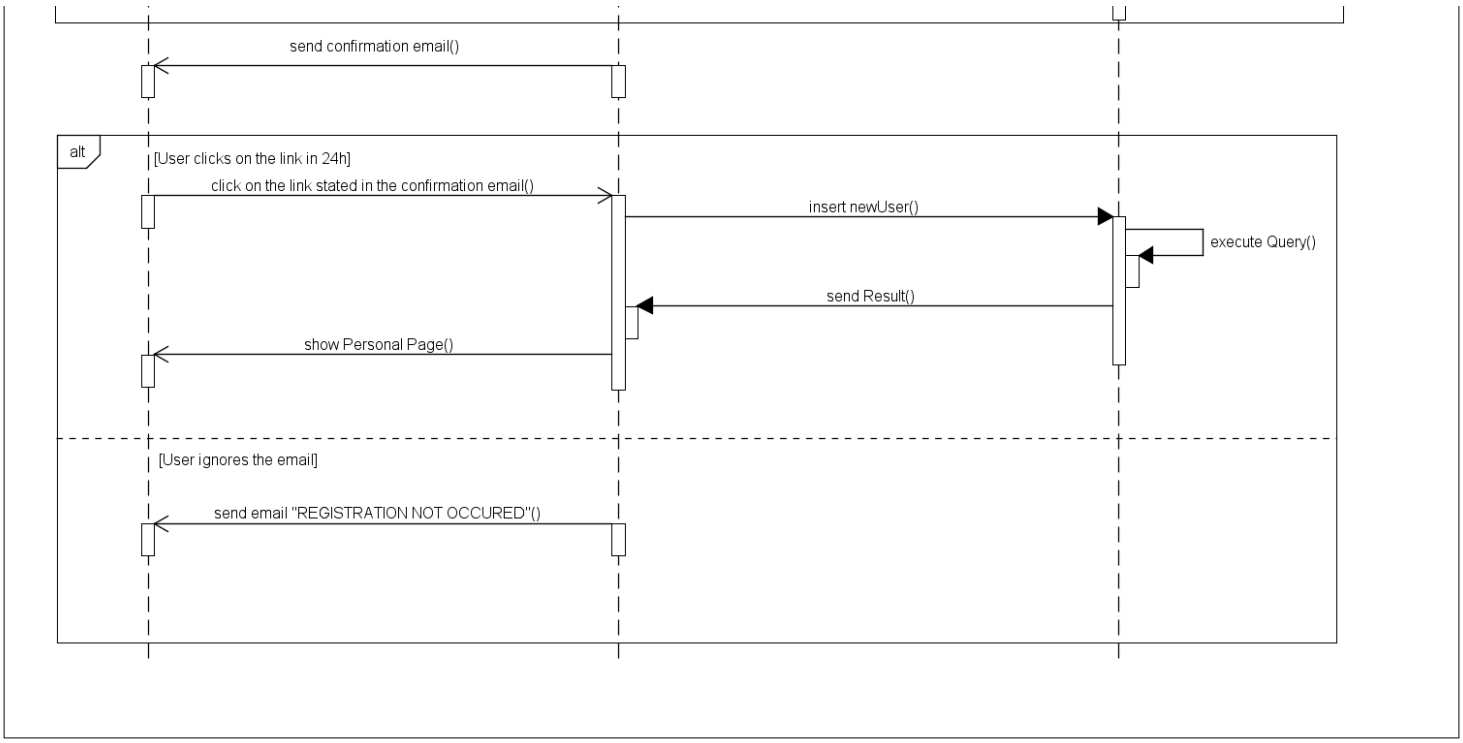


2.6. Runtime view (Sequence diagrams)

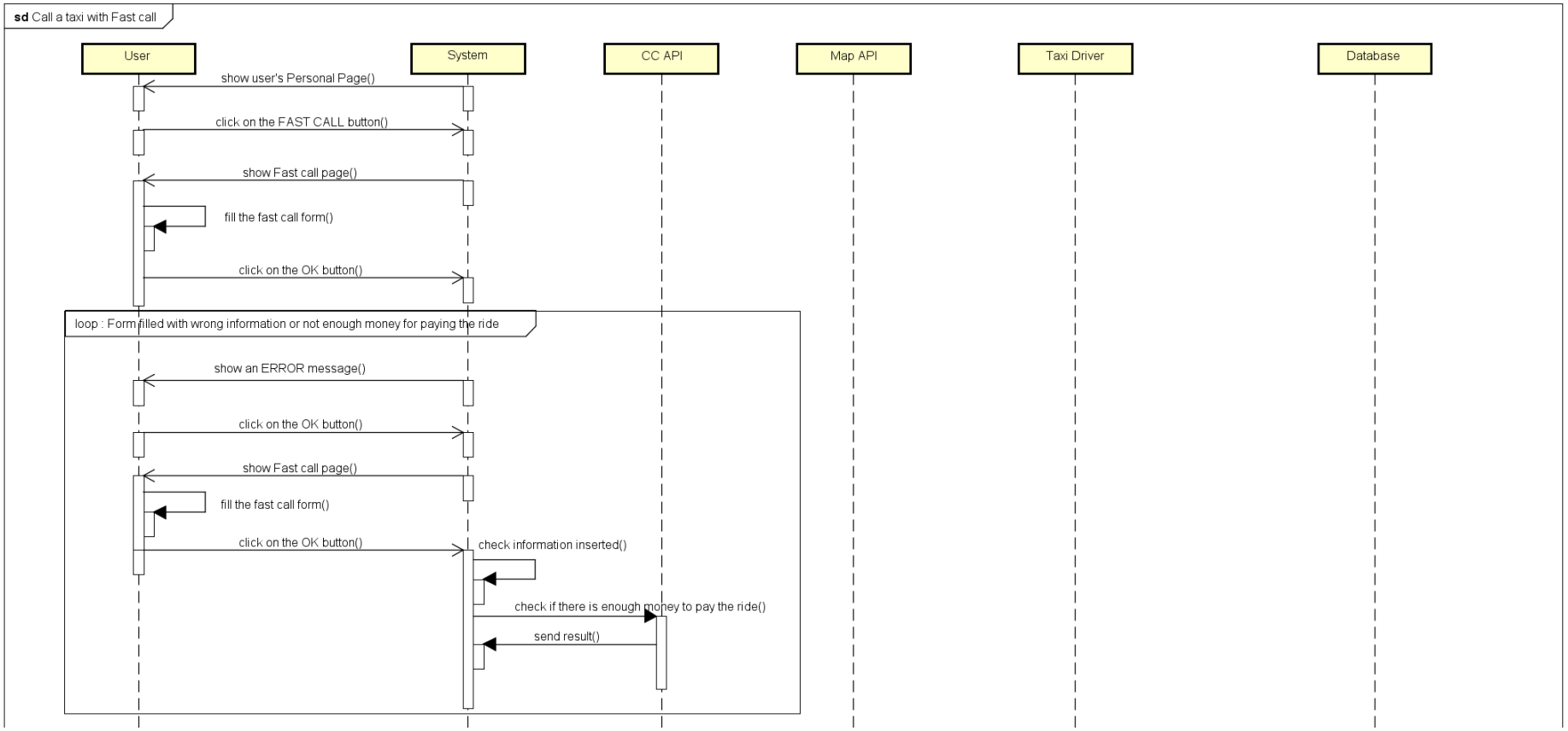
In this section we describe the run-time behaviour of the system in two main situations using Sequence Diagrams.

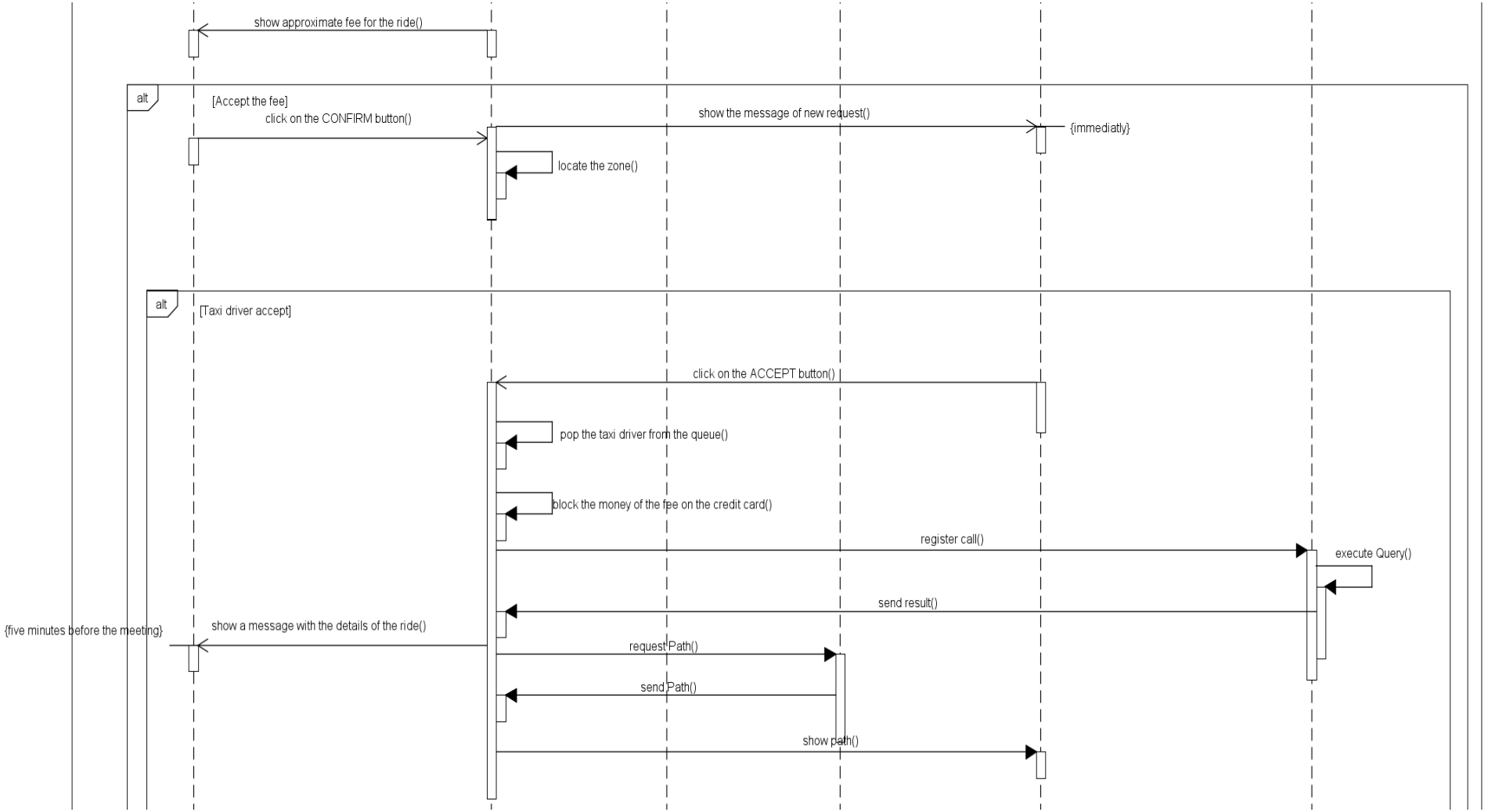
2.6.1. Sign up

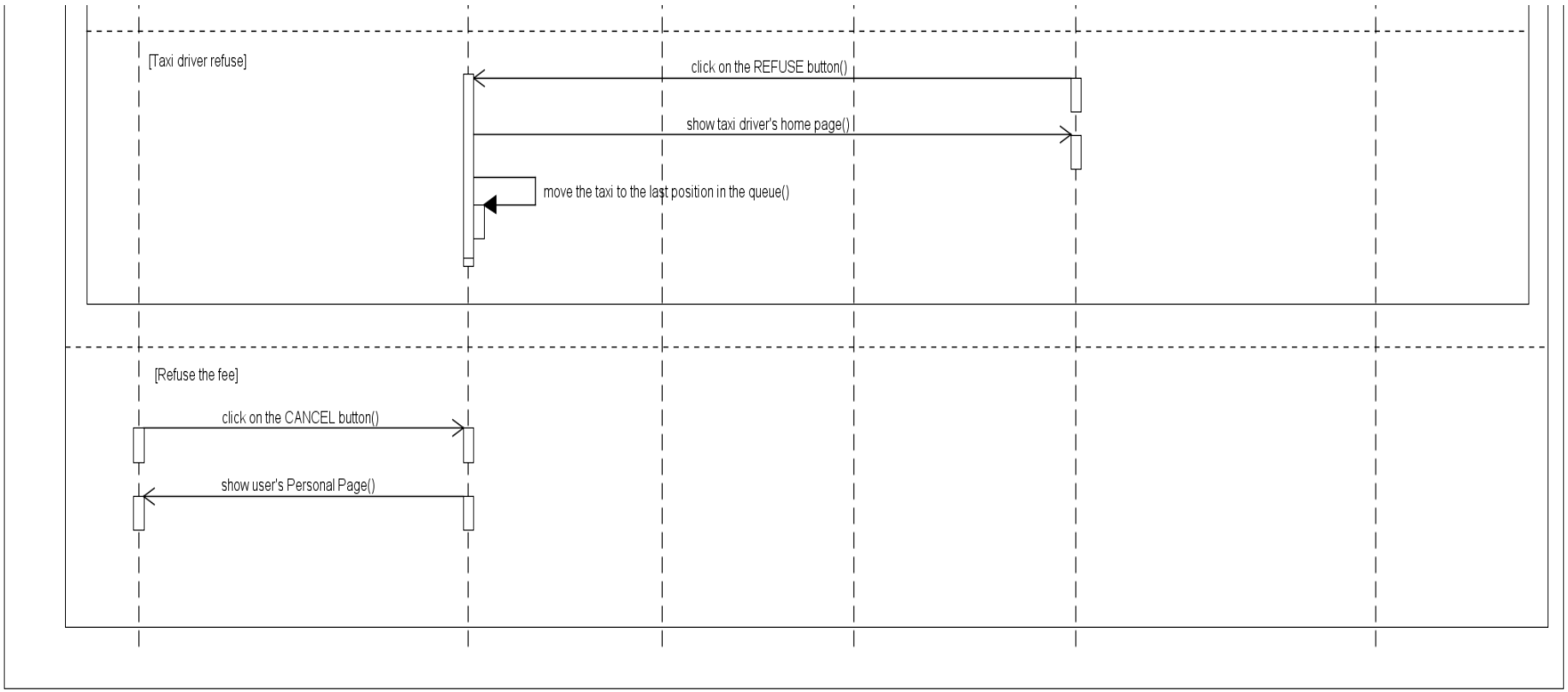



send confirmation email()
alt
[User clicks on the link in 24h]
click on the link stated in the confirmation email()
insert newUser()
execute Query()
send Result()
show Personal Page()
[User ignores the email]
send email "REGISTRATION NOT OCCURED"()

2.6.2. Fast Call




show approximate fee for the ride()
alt
[Accept the fee]
click on the CONFIRM button()
show the message of new request()
{immediatly}
locate the zone()
alt
[Taxi driver accept]
click on the ACCEPT button()
pop the taxi driver from the queue()
block the money of the fee on the credit card()
register call()
execute Query()
send result()
{five minutes before the meeting}
show a message with the details of the ride()
request Path()
send Path()
show path()


[Taxi driver refuse]
click on the REFUSE button()
show taxi driver's home page()
move the taxi to the last position in the queue()
[Refuse the fee]
click on the CANCEL button()
show user's Personal Page()

2.7. Component interfaces

ServerClass {

```
//The interface used to store all the information inside the DB.
DbInterface database;
Set<User> connectedUser;
Set<TaxiDriver> connectedTD;

//Makes the login of a user inside the system and return the instance
of the person logged in (TD o User) (asynchronous call)
Person login(String user, String psw);
//Register a new user inside the system. (Asynchronous call)
boolean register(BeanUser info);
//All the functions that we use to store the information in the DB - here
only few examples.
boolean dbSaveCall(Call myCall);
boolean dbAddUser(User myUser);
//Calculates the fare of a private ride. Is called by the callRecognizer.
void calculateFare(Call myCall);
//Calculates the paths of the call connecting to the API of a map
service and saves them in myCall.path (the path's set of the requested
ride)
void calculatePath(Call myCall);
```

}

Admin {

```
String id;

//All these function are called by the client of the admin.
(Asynchronous call)
//Adds a new zone to the DB (It calls the ServerClass)
boolean addZone(String zoneName);
//Show the Calls that the system is managing
void showCurrentCalls();
//Adds a new street to the DB
boolean addStreet(String nameZone, String streetName, int startN, int
endN);
//Adds a new TD to a zone.
boolean addTD(String id, String password, String zone);
//Adds a new shift to a TD, inside the 2 BeanTime there are the time
and the date of the start and the end of the shift;
boolean addShift(String idTD, BeanTime timeStartShift, BeanTime
timeEndShift);
//Removes a street from a zone (example: if the street is no more
served by the service)
```

boolean removeStreet(String streetName);

//Removes a zone from the service. The operation fails (return false) if there is at least a street in the zone
boolean removeZone(String streetZone);
//Removes a TD from the DB
boolean removeTD(String idTD);
//Removes a shift from a TD
boolean removeShift(String idTD, BeanTime timeStartShift);
//Shows the set of all covered zone by the system
Set<String> showZone();
//Shows the set of all the street of a zone.
Set<String> showStreet(String zone);
}

Person {
String id;
String name;
String surname;

//Makes the logout of a user (Asynchronous call)
boolean logout();
}

User {
UserInteface myCon;
BeanUser myInfo;
Set<Call> myCall;

//Creates a booked or fast call. The difference can be seen at runtime analysing 'BeanTime t'. (Asynchronous call - When user sends the request of a new Call)
Call makeCall(BeanPosition posStart, BeanPosition posArrive, int NPass, BeanTime t);
//Notify the user (synchronous call - after an events happens. Example: A booked ride is confirmed, in this case it is called by the TimeDeamon)
void notifyUser(String message);
//Changes the password of a user. (Asynchronous call)
boolean changePsw(String oldPsw, String newPsw);
//Changes the credit card of a user. (Asynchronous call)
boolean changeCC(CreditCard card);
//Shows to the user the set of his call
Set<Call> showMyCall();
// All these functions modify the information of a call (asynchronous call)

```
        boolean modifyStartPos(Call myCall, BeanPosition startPos);
        boolean modifyArrivePos(Call myCall, BeanPosition arrivePos);
        boolean modifyStartTime(Call myCall, BeanTime tStart);
        boolean modifyNumberOfPassenger(Call myCall, int nPassenger);
}
```

//It is a support class and it's shared between the client and the server.
//It has all the basic information needed for a user.
BeanUser {
String email;
String name;
String mobilePhone;
}

TaxiDriver {
Zone myZone;
TDInterface myCon;

boolean askForACall(Call newCall);

//Gives to the TD all the information about: shift, working zone, general working statistics.
String showInfo();
//Informs the system that the passenger has been delivered (The client checks the position and then find the user that is arrived)
void passengerDelivered(User userDelivered);
//Updates the map of a TD when there is a change (Example: the user accepts a call or a passenger is delivered) (synchronous request)
void showMap(Call myCall);
//Removes every block and temporary flags from the taxi driver. The function is called when a TD is newly available.
void initializeTD();
//This function is called by the Taxi-Driver client, automatically when he is not serving any call and he comes back to his zone.
The Client detects the position of the Taxi-Driver to GPS.
This function will invoke also the *appendNewTaxiDriver* function in the QueueManager in order to be appended at the end of his queue's zone.
void beAvailable();
//Inform the system that the passenger has been picked up
void passengerOnBoard(User userDelivered);
//Inform the system that the passenger is not present at the meeting point.
void passengerNotPresent(User userDelivered);
}

```
//This class will manage queues in all the existing zones.
QueueManager {
        Set<Zone> managedZone;

        //To manage simultaneous calls we used a sync function.
        //When a new call request arrive the first available (so non-blocked)
        TD is blocked, then a request is sent to him, and only after he answers
        the TD is unlocked. The full algorithm is explained in the algorithm part
        of the Design Document.
        boolean manageCall(Call myCall);
        //Returns the first non-blocked TD and block it. Calls the sync function
        in zone getFirstAvailable.
        TaxiDriver blockTaxi(Zone myZone);
        //Unlocks the taxi driver. The function is called when a TD answers to
        a request.
        void sync unlockTaxi(TaxiDriver myTD);
        //The TD is appended at the end of the queue. The function is called
        after a TD refuses a call or is newly available. This functions calls the
        function appendNewTD in zone.
        boolean appendNewTD(TaxiDriver myTD);
        //A request is send to the taxi driver. The function is called after a TD
        is blocked and it waits the TD for at maximum 15 seconds.
        boolean sendRequest(Call newCall);
        //Removes a TD from the class Zone after he accepted a call. This
        functions calls the function remove in zone.
        void remove(Zone zone, TaxiDriver myTD);
}

Zone {
        //Every zone is identified by a name (different name for different zone)
        String name;
        //The TD that are waiting a call without passenger.
        Set<TaxiDriver> myQueue;
        //All the TD that are actually working (waiting a call or delivering a
        passenger)
        Set<TaxiDriver> inService;
        //Set of all streets and house numbers covered by the zone
        Set<Path> streets;
        //Tells if a position is inside the zone or not
        boolean isMyZone(BeanPosition startPos);
        //Returns the first non-blocked TD and block it.
        TaxiDriver sync getFirstAvailable();
        //The TD is appended at the end of the queue. The function is called
        after a TD refuses a call or is newly available.
        boolean sync appendNewTD(TaxiDriver myTD);
        //Removes a TD from the class Zone after he accepted a call.
```

```
        void sync remove (TaxiDriver myTD);
}
//Class representing a call instantiated
Call {
        String id;
        BeanPosition startPos;
        BeanPosition arrivePos;
        BeanTime startTime;
        BeanTime arriveTime;
        Set<Path> path;
        TaxiDriver myTD;
        User myUser;
        int nPas;

        //Minutes of delay that the Taxi Driver has done respect to the
        scheduled arrival time due to traffic condition. It will be used for
        calculate the effective fare.
        int nDelayMinutes;
        //All getter and setter needed
}

Path {
        String road;
        int startN;
        int endN;
        int nPas;

        //These are calculated through API
        BeanTime Hp;
        BeanTime Ha;
        int dist;
}

//It's a support class and it's shared between the client and the server.
//It specifies the position of a user in the map.
BeanPosition {
        String nation;
        String city;
        String road;
        int houseNum;
        Float coordx;
        Float coordy;
}
```

```
//It's a support class and it's shared between the client and the server.
//It has the information about a specific time.
BeanTime {
        int year;
        int month;
        int day;
        int hour;
        int minute;
}

//A thread that checks the booked call in his list
TimeDaemon {
        //The lists of call that are waiting to be served
        Set<Call> waitingCall;

        //Appends a new call to the list of monitored calls.
        void addACall(Call newCall);
        //Wakes up a call and executes it. It is called the function manageCall
        in QueueManager.
        void notify(Call wakedCall);
}

SharedSet {
        //Lists of Calls with "taxi-sharing" option enabled already assigned to a
        Taxi-Driver
        Set<SharedCall> mySCall;

        //Manage a "Booked Call" after it is waked up by the
        SharedCallDeamon. This function calls manageCall and if it returns
        false calls createNewSCall.
        void manageCall(Call newCall);
        //Compares the call with the other in mySCall to find a compatible
        shared call. It's sync to avoid inconsistences
        boolean sync compareCall(Call newCall);
        //It's called if no compatible calls are available with newCall. Creates a
        new SharedCall and add it to the SharedSet.
        void createNewSCall(Call newCall);
}
```

SharedCall {
Set<Call> call;
Set<Path> streets;

//Checks if the time of the Call is neither before the SC nor after 10 minutes.
boolean checkTime(Call myCall);
//Tells if a path is contained in this SC
boolean findPath(BeanPosition posStart);
//Check if the number of seats are enough to satisfy the request.
boolean checkSeat(Call newCall);
//Inserts the newCall to the set of calls.
void addCall(Call newCall);
//Inserts the new passengers to the SC
void addPass(Call newCall);
//Inserts the path inside the SC
void splitPath(Call newCall);
}

//Class to manage calls done with "Taxi-Sharing" option enabled.
SharedCallManager {
//Calculates the fare of a shared ride. Is called by the callRecognizer.
void calculateFare(Call myCall);
}

//This class represents the extension of the User, able to make Shared Calls.
UserTSharing extends User {
//The new interface with all the new necessaries function.
UserTSharingInterface mySCon;
//It's like makeCall of User but with this function the Call will be managed by the SharedSet.
Call makeSharedCall(BeanPosition posStart, BeanPosition posArrive, int NPass, BeanTime t);
}

//This component is the first one that has been invoked after a request arrives to the Server.
This component has a reference to "Server" and to "SharedCallManager".
It will call the appropriate functions for instantiate new Calls when a request arrives and if the Call has been done with "Taxi Sharing option" enabled.
It is also able to recognize if a call is shared or not, and it will call the appropriate function to calculate the fare of a road, for example.

CallRecognizer {

SharedSet sharedSet;
ServerClass server;

//List of all the current instantiated Calls.
Call<Set> calls;

//from the call details (passed by the client) he will take the reference to the effective call searching in its set.
Call takeReferenceToCall(call details);
//Recognizes if the call is shared or not.
boolean recognizeSharedCall(Call newCall);
//It recognizes if a call is shared or not and call the appropriate function for computing the exact fare for the ride. In fact it will call "calulateFare" in the ServerClass in case of a private ride, "calculateFare" in the SharedCallManager otherwise. This function is called after that the taxi-driver has delivered the passenger and has pushed the button "passenger delivered" on the screen of his on-board computer.
void calculateAppropiateFare (call details);

}

2.8. Selected architectural styles and patterns and other design decisions

JEE decisions:

In the Web-Tier we adopt the JSF (Java Server Faces) technology. JSF uses the MVC Pattern and simplifies the development of the user interface.
The Web Tier communicates with the business tier, which contains Enterprise Java Beans, components that encapsulate the business logic of the application.
Since a lot of clients connect to the Server (for example to perform call request), Session Bean will be used to represent the client instance inside the Server.
To manage the Data-tier we use a MySql Server.
The Business Tier should communicate with the Data Tier through JPA (Java Persistent API), used also for managing the persistence of data.
Moreover, we provide API using JAX-RS (Java API for Restful Web-Service).

Server description:

The **Server** manages all the logic of the application and it provides different interfaces for the various type of Client. After an authentication session, the Server is able to recognize which kind of user has been connected through the Network, thus, it provides the correct interfaces to communicate (User, Taxi-Driver, Admin).

The Client packages are two: one for the Admin, and one for User/Taxi-Driver. Thus, we provide the Admin package to the person/people that will administrate the system and will have permissions to access to the Database, while the other package is provided to people that will use the application daily (users and taxi-drivers).

Given that the package for Users and Taxi-Drivers is the same, whenever a Client tries to connect to the Server, the Server is able to recognize if the credentials inserted (if existing) belong to a User or to a Taxi-Driver, so a new instance of "Person-Interface" is instantiated, using a Factory Method. In fact, thanks to the use of inheritance, "Taxi-Driver Interface" and "User Interface" inherit some attributes from "Person Interface". The same principle is used also for the concrete objects User and Taxi Driver. They have some common attributes and methods, so they simply extend the same "Person" class.

In the Server part, a class contains all the functions to connect to the Database. This class should be unique, so we used the Singleton pattern to ensure that.

The same pattern was applied to design the "Queue Manager", which is an object managing all the queues in every zone. Thus, it has the references to each "Zone". In order to manage a call, the object "User" invokes a method in

"Queue Manager". Initially, a User should not have a reference to this object. This is another reason for using the Singleton pattern: a reference to Queue Manager is given to the User only when it requires it.

To manage the "Call" part, we thought from the beginning that different types of call could exist: the "Call" class is abstract, which means that cannot be instantiated because we need to specify the particular features of the call made. The first type is "Fast Call", which extends the parent class. The same thing can be done with "Booked Call".

In order to manage the booked call, we added an object called "Time Scheduler": it is actually a thread that has a complete list of Calls and controls when is time to start a new procedure for assigning a new request (i.e.: looking for a taxi-driver, assign the taxi driver who accepts to that call, etch…).

There are some classes having the prefix "Bean" in the name. These small objects are thought to be in both packages of Server and Clients (User/Taxi-Driver). In this way small structured information could be shared.

Successively, we thought how to extend the system designed in order to add the Taxi-Sharing service.

In order to do so, in the Server side, we extended the User, adding the functionality to make a "shared call". We choose to do this because, in our opinion, in the future we can decide to have some users enable to do shared call and other not. This approach is very flexible because it allows creating new type of users and taxi-drivers (i.e.: premium-users or gold Taxi-Driver).

To manage the entire system of shared call, we added also other classes, referring only to "Call" (so we are able to manage either booked calls or fast calls indiscriminately).

Calls done with "Taxi-Sharing" option enabled and served by the same taxi-drivers in a unique ride are grouped in a "Shared Call" object.

There is also another object called "Shared Set" (which is unique, thus we use the Singleton pattern also there) which groups all the Shared Call-s served in the current moment. Using this approach, whenever a call with "Taxi-Sharing" option enabled arrives to the Server (or it is time to serve it, in case of "Booked Call"), the system looks firstly in the Shared Set looking for a instantiated "Shared Call" with a compatible path: if one is found, the call is assigned to the correspondent "Shared Call", otherwise a new "Shared Call" is instantiated and assigned to "Shared Set" (look at algorithm part for clarification).

Client description:

The **Client part** is thought to be built using MVC (Model-View-Controller) Pattern.

The only logic part located in the Client is the control of the syntactic correctness of information inserted by the user (i.e. the email address contains the character "@", the credit card code is composed by 16 numbers, etc. …).
Moreover, it is able to detect the position of a Taxi thanks to the GPS installed on board, and communicates it to the Server.

The Application Client connects himself to the Web service through an HTTPS Connection.

The Mobile Application is though to be an extension of the Web-one, and it is developed using the API exposed by the Web Application.

Communication decisions:

Most of the messages Client to Server are synchronous. For this reason, we can adopt a Request-Response method.
Regarding the asynchronous messages, we can adopt a MOM (Message Oriented Middleware).
The asynchronous messages that arrive to the Served Side are all from Taxi-Drivers, when they communicate that:

- passenger is on board (as soon as he has picked up him)
- passenger has reached the destination (as soon as he leaves the taxi, being arrived to his destination)
- passenger is not present to the meeting point

Other decisions

The application should provide Rest - API to extend the service.
Moreover, we duplicate the server to guarantee the availability of the service (the duplicated Server should be located on another machine).
We duplicate also the Database for safety (the second Database should be on another machine).

SharedCall decisions

The Shared Call part is developed using API exposed by the main application. To implement this part, we adopt a sort of Publish-Subscribe Architecture: every time a User requests a Shared Call, the Server control if a compatible call already exists: in the negative case, the client publish a new Call; in the positive case it subscribes itself to the existing compatible Call. In this case, actually, all the participant to a Call are subscribers: in fact, every time a new passenger join the call, a message of updating is forwarded to all the people interested in the same route (even if they do not know the identity of the other passenger). A dedicated component (the dispatcher) should be used for this purpose.

3. ALGORITHM DESIGN

3.1. Manage a call

This algorithm is used to manage the taxi queue for each zone and to assign a taxi driver to every request that arrives to the system. It gets as parameter the Call that the user has made and provides as output the algorithm is called ManageCall and it belongs to the QueueManager class.
We thought to simplify the algorithm ManageCall by using another function called *isMyZone*. This function, having the start position of the user that makes the new request as parameter, allow us to identify from which zone the user is asking for a taxi.
The objects used in the algorithm are:
Zone: we use this class to scan the zone that are covered by our service

- *TaxiDriver:* we use this class to store the first free taxi driver of the queue in a specific zone
- *Call:* from which we get the origin and destination addresses and house numbers
- *Path:* used to find the specific road in which the passenger is requesting a taxi

Here we report the support function *isMyZone:*
boolean isMyZone(BeanPosition startPos):

```
        //scan every path contained in the set named "streets" of the zone

        foreach (path in streets)

        //check if the road of leaving is equal to the road corresponding to
        that path
        if(path.road is equal startPos.road) {
            //check if the house number of the start position of the user is
            included into the startN and endN of the path
     if(startPos.startN is between path.startN AND path.endN) {


    //if the conditions are satisfied, the function returns TRUE. It means
that the road is in the zone
                    return true;
            }
     }
    }
    //if all the paths of the zone have been scanned and no one matches
    with the start position of the user, then the road is not in that zone
    return false;
}
```

Here we report the general algorithm called ManageCall:

```
boolean manageCall (Call myCall) {
//this variable is used to store the zone in which will be stored the corresponding zone of the road from which the user is making his request
Zone zone <- NULL;

//this variable is used to store the first free taxi driver of the queue TaxiDriver myTD <- NULL;
//this variable is used to store the response of the taxi driver about a ride request
boolean requestOK <-false;
//scan all the zones covered by the system
foreach(z in managedZone) {
        //checks if the start position of the user is in that zone
        if(z.isMyZone(myCall.startPos)) {
                //if the condition is true, then zone will be equal to z
                zone <-z;
                break;
        }
}
//checks if no zone has been found
if(zone equals to false) {
        //Zone not covered by the service;
        myCall.user.notify("Zone not covered");
       //it means that the system does not cover that address
       //the function returns FALSE if the address is not covered by the system
       return false;
}
//if the zone is found we continue with the algorithm
//performs the cycle until there exists a TaxiDriver that can accept the ride (is free)
while(not all available TD are checked in zone.myQueue) {
        //blocks the TD in such a way that no other requests can be sent to him while he is valuating if accept or not the request that was already sent him. The function returns the first free taxi driver of the queue in zone
        myTD <- blockTaxi(zone);
        //Check if exist at least 1 TD
        //if there no exists any free taxi driver the user will be notified and the function returns FALSE
        if(myTD equals to NULL) {
                //Taxi not available;
                myCall.user.notify("No Taxi available");
                return false;
        }
        //IF here TD found
```

```
        //set the attribute Taxi driver of myCall with the value of myTD
        myCall.setTD(myTD);
        //send request to the Taxi Driver and the response is stored in the
        variable "requestOK"
        requestOK <- sendRequest(myCall);
        //remove the TD from the queue of the zone
        remove(myTD);
        if(requestOK) {
        //if the TD accepts the request
        //ServerClass is singleton => We get the reference
                //invoke the function dbSaveCall of myServer object to store the
                request myCall into the database
                myServer.dbSaveCall(myCall);
                //calculate the set of path relative to the Call and save them in
                the Call's instance
                myServer.calculatePath(Call myCall);
                //the user will be notified about the result of the operation
                myCall.user.notify("TD found");
                return true;
        }
        else {
                //if the TD rejects the request
                //the TD is inserted in the last position of the queue
                appendNewTD(myTD);
                //he will be set unlocked in such a way that he can receive new
                request
                unlockTaxi(myTD);
                        //set the TD of myCall with null
                        myCall.myTD.setTD(null);
                        //initialize the variable myTD for the next iteration
                        myTD <- null;
        }
}
return false;
}
```

3.2. Manage Taxi-Sharing requests

This algorithm is used to check if a taxi-sharing ride, compatible with the new taxi-sharing request, is already present into the SharedSet. If a compatible ride is found, then the path will be updated and the new passenger will be added to the number of passengers of the relative ride; at the end the algorithm will return TRUE; otherwise the algorithm return FALSE and a new instance of SharedCall is created. This algorithm is contained into the SharedSet.
We defined different support functions to simplify the general algorithm.

Following the functions:

- *boolean checkSeat(Call newCall);*
- *boolean findPath(BeanPosition posStart);*
- *void splitPath(Call newCall);*
- *boolean checkTime(BeanTime timeStart);*

In the algorithm, we used the following objects:

- *newCall*: is an instance of Call
- *path* is an instance of Path which represents every street that the taxi driver has to go through
- *posStart* is an instance of BeanPosition which contains information about the position of the new passenger
- *oldPath* is an auxiliary object used to help in the split operation. It will contain the information of the path before that the new passenger was added to the ride
- *timeStart* is an instance of BeanTime which contains information about the meeting time

Here we report an explanation of the functions:

boolean checkSeat (Call newCall):

1. From the origin Path, we will check, through the control at the point 2, all the successive paths, corresponding to the specific ride, until it has the same destination road of the ride and the house number of destination between *path.startN* e *path.endN*.
2. If the attribute *nPas* in Path, plus *nPas* in *newCall*, is always less or equal to 4 for each path contained in *newCall*, the algorithm returns true; otherwise, return false.

boolean findPath (BeanPosition startPos):

For each path in the set named "streets" of the class *SharedCall*:

1. Check if the *path.road* is equal to the *startPos.road*;
2. If it is equal: check if the *startPos.houseNum* is between *path.startN* and *path.endN*;
3. If this control is true then the function returns TRUE;

When all paths are checked, if the function has not returned yet, it returns FALSE.

void splitPath (Call newCall):

1. Search the path (that we call *oldPath*) which contains the origin address of the *newCall* and, if the *newCall's* house number is strictly included in the *oldPath startN* and *endN*, remove the old path from the set of "streets" in *SharedCall* and add two new instances of *Path* in the following way.
   Create two new instances of *Path* (*path1* and *path2*) with the same street and the same number of passengers.

2. **Instance of path1:**
   //the startN of the new Path1 is equal to the startN of the oldPath
   path1.startN = oldPath.getStartN();
   //the endN of the new instance Path1 is equal to the passenger's start position house number
   path1.endN = newCall.getStartPos().getHouseNum();
   //**Hp**: time of the departure in that road **Ha**: time of the arrival in that road **dist**: km to travel through that road.
   These attributes will be calculated through Map API
   Hp, Ha e dist = = calculated through API
   Instance of path2:
   //the startN of the new instance path2 is equal to the passenger's start position house number
   path2.startN = newCall.getStartPos().getHouseNum();
   //the endN of the new instance path2 is equal to the endN of the oldPath
   path2.endN = oldPath.getEndN();
   //these attributes will be calculated through Map API
   path2.Hp, path2.Ha and path2.dist = calculated through API

   Search the path (*oldPath*) that contains the destination address of the *newCall* and, if the house number of the *newCall* is strictly included in the *oldPath startN* and *endN*, then the *oldPath* is removed and then two instances of *Path* will be added to the set "streets" in the following way:
   Create two new instances (*path3* and *path4*) of *Path* with the same street and the same number of passengers.

   Instance of path3:
   //the startN of the new instance path3 is equal to the startN of the oldPath
   path3.startN = oldPath.getStartN();
   //the endN of the new instance path3 is equal to the passenger's start position house number
   path3.endN = newCall.getStartPos().getHouseNum();

//these attributes will be calculated through Map API
path3.Hp, path3.Ha and path3.dist = calculated through API

Instance of path4:
//the startN of the new instance path4 is equal to the passenger's end position house number
path4.startN = newCall.getEndPos().getHouseNum();
//the endN of the new instance path4 is equal to the endN of the oldPath
path4.endN = oldPath.endN;
//these attributes will be calculated through Map API
path4.Hp, path4.Ha and path4.dist = calculated through API;

boolean checkTime(Call myCall):

1. Calculate the time to reach the passenger's start position house – number, from the start of the road, in which the passenger has requested a ride.
2. If there is a taxi that, with a tolerance of 10 minutes, passes through that road, the function returns TRUE; return FALSE, otherwise.

Here we report the general algorithm used to manage taxi-sharing requests:
boolean sync compareCall(Call newCall):

1. *SharedCall sc <- null;*

For each call in mySCall:

2. Checks if *call.checkTime(newCall.getStartTime())&&call.findPath(newCall.getStartPos())*
   //checks if the time of departure and the start position of the newCall are compatible with a ride already present into the SharedSet
3. If the conditions are satisfied: sc will be equal to call;
4. Checks if there are enough seats through: *if(sc.checkSeat(newCall))*
5. If the condition is satisfied, the algorithm will invoke: *sc.addCall(newCall), sc.addPass(newCall), sc.splitPath(newCall) and returns TRUE;*
6. When all calls are checked, if the function has not returned yet, it returns FALSE.

3.3. Calculate fare for a "taxi-sharing" call

This algorithm will be able to calculate the exact fare for a user that has made a call with "Taxi-sharing" option enabled.
The function will be invoked by the component *CallRecognizer* when it has to calculate the fair for a ride.
The computation of the fair takes place after that the taxi-driver of the call has pushed on the button "Passenger Delivered".

- Price per Kilometres is a constant called *PK*.
- *myCall* is an instance of *Call*.
- *mySharedCall* is an instance of *SharedCall*, which has a reference to *myCall*.
- *currentUser* is the *User*

In this algorithm the following objects are involved:

- *Call,* from which we get the origin and destination addresses and house numbers.
- *SharedSet*, in which we search the single path, and we look inside that to find the number of passenger involved, and to check if this is the effective destination of the ride or we have to go on and add paths and the equivalent fare.
- *Path*, consequently.
- *User*. The user that has made the call.

1. //The departure address of the call I am interested in
   String originRoad = myCall.getOriginPath.getRoad
2. //The origin house number of the call I am interested in
   Int originNumber = myCall.getOriginPath.getStartN
3. //The destination address of the call I am interested in
   String destinationRoad = myCall.getDestinationPath.getRoad
4. //The destination housenumber of the call I am interested in
   Int destinationNumber = myCall.getDestinationPath.getStartN
5. Initialize an integer variable *R* to 0 (this variable will contain the final fair the user should pay)
6. Get the *nDelayMinutes* (number of minutes of delay) in *Call*. Then multiply this number for the *DelayMinutePrice* (constant). Save the result in a variable called *delayFare*.
7. Then perform a search in the set of Paths in *mySharedCall* until a Path with *startN==originNumber* and *road==originRoad* is found.
8. Save the Path's instance in a temporal variable called *currentPath*.

Repeat the following operations in the set of *Path* in *mySharedCall* until a Path with *road==destinationRoad* and *endN==destinationNumber* is found:

9. Look in the *currentPath* instance the variable *dist* (number of kilometres), and multiplies it for *PK*. Divide the obtained number for *nPas* (the number of passengers).

10. Sum the result to R.
    Thus do: *R = R + ((currentPath.dist*currentPath.PK)/currentPath.nPas*)
11. Finally, the total fair is saved in R.

Now do: *R+delayFare.*

Afterward we have to ensure that the fare calculated does not exceed the maximum fare allowed, which is double of the normal one. So check if the result (*R+delayFare*) is less than *2*R*. If it is not, set the result to *2*R*.

Then we call the function "*payTheRide*" in *Call* that, through the reference to *User*, will connect to the API of his credit card taking the fair computed.

4. USER INTERFACE DESIGN

Refer to the section of the RASD Document. There it is possible to find the main GUI Design both for the Mobile Application and the Web Application.

5. REQUIREMENTS TRACEABILITY

In the following section we show how the design elements (explained in this document) map the requirements defined in the RASD.

Requirements matching:

Sign Up functionality:
We provide this functionality through the function "*register*" located into the *ServerClass*.
(Look at Sequence diagram 2.6.1)

Log in functionality:
We provide this functionality through the function "*login*" located into the *ServerClass*. This function will instantiate a new *Person*. The Client will be represented in the Server with this class.

Logout functionality:
There is the function "*Logout*" in the class Person. So both User and Taxi-Driver inherit this function.
After having executed this method, the correspondent Person class will be deleted.

Call a taxi:
User class has a *makeCall* function, which is called after the request of a User-Client. This function checks also the correctness of information inserted by the User.
(See Sequence Diagram 2.6.2. for *FastCall*).

Pay the ride:
The component *callRecognizer* has a function called *calculateFare*, which recognizes the appropriate type of call (private or shared) and calls other respective function in *ServerClass* (for private calls) and *SharedCallManager* (shared call).
After having calculated the fare, the class previously called will connect to the API of the credit card of the user in order to provide the payment.

Book and manage a reservation:
The object Call is extended by "*Booked Call*". User has several functions to manage a reservation: *modifyStartPos, modifyArrivePos, modifyStartTime, modifyNumberOfPassenger* (to modify, in order, the initial position, the destination, the departure time and the number of passenger).

Enable taxi-sharing option:
In order to provide this functionality, we added some classes such as: *SharedCall, SharedSet, SharedCallDaemon, and SharedCallManager.*
To allow User to share the taxi we have to check if a path is compatible to another one: we do that through the *compareCall* algorithm, contained in *SharedSet*.
(See Algorithm 3.2 in Algorithm Design Section)

Accept or refuse a call:
In *QueueManager* there is the *sendRequest* function. This function will send a request to the Taxi-Driver client, which has to accept or refuse the ride proposed.

Inform the client about the incoming taxi details:
On the Server-side, there is the function notify inside the User Class, which will inform the correspondent user on the client-side about generic information regarding his call, so also about the incoming taxi details.

Calculate the path of the trip:
The *manageCall* function in the *QueueManager* invokes a *calculatePath*, a function in the *ServerClass*, which is able to connect to the API of a map service and calculates the paths of the ride.

Manage taxi queues:
Taxi-drivers are assigned to the different zones by the Admin, which inserts the different shifts of the drivers in the Database.
The *QueueManager* manages the various calls through the following functions: *blockTaxi, sendRequest, unlockTaxi, appendNewTaxiDriver.*
To control the availability of the *TaxiDriver*, when a driver return available, the Client detects his position through GPS and, on the Server Side, a function called *beAvailable* (in the *TaxiDriver* class) will be called in order to append it to the queue of his zone.

Assumptions matching:

The system assigns a number of taxis in each zone proportionally to the amount of traffic in that zone.
The Admin assigns different shifts to Taxi Drivers, saving them in the Database communicating with the *ServerClass* through the "*AdminInterface*".

A unique set of shared-taxi exists. When a taxi driver accepts a call with "taxi-sharing" option, he is added to the set of shared-taxi.
There is the *SharedSet* class, which is Singleton. It will keep track of the Shared Calls already instantiated. When a new request of Shared Call arrives, it will be looked before here if a compatible call already instantiated exists.

The taxi driver will receive a notification about the presence of a new passenger and his map will automatically update showing the new path.
When a new passenger is assigned to a Shared Ride, the map of the Taxi Driver will be updated.

The taxi driver communicates to the system information about the ride through some buttons:

- *Passenger is on board;*
- *Passenger is not present in the meeting place at the specified time;*
- *Passenger has been brought to destination.*

There are functions in the *TaxiDriver* Class: *passengerOnBoard*, *passengerNotPresent*, *passengerDelivered*.

Revision History:

04/12/2015 - First Version.
10/02/2016 - Improved architectural style (Jee Framework) and other decision decisions (Strategies adopted for synchronous and asynchronous messages; Shared Call part architecture).