



Politecnico di Milano
2015-2016

Software Engineering 2: “MyTaxiService”
Design Document
version 1.0

Author: Nenad Petrovic

22nd November 2015

Contents

1	Introduction.....	3
1.1	Purpose.....	3
1.2	Scope and document overview.....	3
1.3	Definitions, acronyms, abbreviations	4
1.4	Reference documents	6
2	Architectural design	6
2.1	Overview	6
2.2	High level components and their interaction.....	8
2.3	Component view	10
2.4	Deployment view	13
2.5	Runtime view	15
2.6	Component interfaces.....	21
2.7	Selected Architectural Styles and Patterns	37
3	Algorithm design	45
3.1	Scheduling algorithm	45
3.2	Taxi zone determination.....	47
3.3	Cost estimation algorithm	51
4	User interface design.....	53
5	Requirements traceability	65
6	Database design	67
7	Software and tools used	68
7.1	Hours of works.....	69
8	References.....	69

1 Introduction

1.1 Purpose

The Design Document (DD) that is going to be presented focuses on presenting the most important concepts used in the design of the MyTaxiService application, which includes the architectural design, algorithm design, data design and user-system interaction design.

This document represents the main design ideas of the application, so its main purpose is to support the implementation and development phase of the project and to give an overview what is the software to be, in fact – so the developers, programmers, system engineers and user interface designers would have an overview what is going to be their task, and how should they do it.

Also, this document is going to be presented to the business owners (and, if necessary to other stakeholders), in order to give them an idea how the product is going to be shaped and deployed, and give them brief overview what their interaction with the system that will be developed is going to look like.

1.2 Scope and document overview

As it has been told previously, this document presents the main ideas related to architectural design, algorithm design, decisions related to user-system interaction design and how system requirements affect the design decisions of the MyTaxiService application in order to shape the image of what is going to be done during the implementation.

Architectural design: The part of the document that will present the complete system architecture overview. First, it will focus on high level components and their interaction. Then, the architectural design will be presented from 3 different points of view – component, deployment and runtime view. Component view presents the components of the system and the interfaces they use in their interaction, deployment view presents how the components themselves are deployed, and runtime view shows how the components interact in order to accomplish specific use cases, using sequence diagrams. After that, component interfaces are going to be described. And, finally, the design styles and patterns used, along with other design decisions are going to be explained and described.

Algorithm design: This part of the document deals with the algorithms used in application. Algorithms are going to be explained and described using pseudocode (notice: not real implementations) and necessary graphical representations in order to make the ideas more clear. Please notice that not every algorithm is going to be presented – only the most important, that represent the core of the MyTaxiService and are the most specific, but not the algorithms that are intuitive and trivial. The algorithms considered will be related to taxi scheduling, taxi zone determination and cost calculation, as they are closely related to the specific domain and represent the core concepts of this system.

User interface design: Here, the graphical user interface is going to be presented – so the reader can get an idea of the look and feel of the system that is going to be developed. This part is already present in RASD document, so Design Document completely references to RASD document in this part, as RASD document contains a complete set of mockups from different points of view, together with state transitions described in statechart diagram.

API design: This part will present how the API interface intended for future developers is going to be designed. This API would allow usage of the application from external application, which could help developers to embed the functions of MyTaxiService into their creations, so this would make this application reusable.

Database design: This part will present the database that is used in MyTaxiService system and its ER diagram.

Requirements traceability: This part will explain how the requirements defined in RASD map into design elements and affect design decisions.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

- Starting point: is a location where the drive should start from, determined by its GPS coordinates.
- Ending point: is a location where the current drive stops, determined by its GPS coordinates. At this point, taxi driver's availability changes automatically to available.
- Request: is a message which consists of user's desired destination for a taxi drive and, optionally, maximum waiting time . The user himself is a „sender“ of a request, while the taxi driver who receives the request is called „receiver“. If there is no taxi driver who can receive the request, the receiver part of the message is empty (user receives a message stating that there is no taxi available and gives him/her an option to send the same request again or change the desired destination), and in case of forwarding the message to another taxi driver, system changes the receiver to the taxi driver polled from a queue. The request contains starting point-determined by sender's GPS location and ending point- which is, in fact the desired destination selected by user.
- Response: is a message which contains „y“-yes in case where taxi driver accepts the request or „n“-no in case where taxi driver rejects the request by user. This message also contains the estimated time needed for taxi driver to come to the requested starting point and estimated price. In this case, taxi driver has a role of sender, while the user who sent the request is receiver. Receiver part of this message can't be blank. User can accept or reject the drive offer.
- Report: is a message written by user or taxi driver during the drive event, in order to mention bad behaviour of the other side. There is no strict definition for „bad behaviour“, so the one who reports has to write reason and describe the situation itself as close as possible. After that, administrators can view the reports and decide to delete user from system or not. User that is banned from system is prevented from using it and can't login or register once again, because his/her fiscal code is on the „black list“. Single user can receive many reports. Reports are stored in database and could be viewed by administration.

- Taxi zone: a part of city (approximately 2km²), which is defined by its center point, and its boundaries are calculated and managed by the system. Each taxi zone has its taxi queue, which consists of car identification numbers of available taxi drivers whose current location belongs to its boundaries. The city has at least one taxi zone (in case of a very small town).
- Availability: could have value „y“(yes)- which means „available“ or „n“(no)- which means „unavailable“. If taxi driver is available, he/she is placed in taxi queue related to his/her taxi zone and is able to respond to requests belonging to this taxi zone. If taxi driver is unavailable, he/she can't receive requests. When taxi driver accepts the request, his availability switches to „unavailable“. After finishing the drive, the availability switches to „available“ and is placed in a queue belonging to his/her current taxi zone. In certain cases, when something goes wrong (traffic rush, accident etc.), taxi driver can manually switch to „unavailable“. When taxi driver sends S.O.S signal, his/her status changes to „unavailable“ automatically.
- Drive event: is a system abstraction of a taxi drive, and consists of request by user, driver's response and reports during a drive. There could be many reports, or no reports at all. Administrations can browse the database of drive events and see the actors of a drive event related to request and response. During active drive event, user can report driver, or driver can report user for bad behavior by describing the reason of report as close as possible. Drive event starts after the user accepts the offer by taxi driver. Sometimes, it is called simply “drive” in this text.
- Estimated waiting time: time needed for a taxi driver to come to the user's current location.
- Maximum waiting time: maximum time that is user ready to spend waiting for a taxi to come.
- Bad behaviour: aggression, avoiding payment, offensive behaviour etc.

1.3.2 Acronyms

- API: Application Programming Interface.
- DD: Design Document.
- DBMS: DataBase management system.
- DB: DataBase.
- EJBs: Enterprise JavaBeans (EJB) is a managed, server software for modular construction of enterprise software, and one of several Java APIs. EJB is a server-side software component that encapsulates the business logic of an application. The EJB specification is a subset of the Java EE specification.
- GUI: Graphical User Interface
- HTTPS (HTTP over SSL, and HTTP Secure) is a protocol for secure communication over a computer network which is widely used on the Internet. HTTPS consists of communication over Hypertext Transfer Protocol (HTTP) within a connection encrypted by Transport Layer Security or its predecessor, Secure Sockets Layer. The main motivation for HTTPS is authentication of the visited website and protection of the privacy and integrity of the exchanged data.

- JPA: The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.
- JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.
- JEE: Java Enterprise Solution
- JVM: Java Virtual Machine
- OS: Operating System.
- MVC: Model View Controller
- REST (REpresentational State Transfer) is an architectural style, and an approach to communications that is often used in the development of Web services.

1.4 Reference documents

This document references to RASD document especially in part which deals with GUI.

- Requirements and Analysis Specification Document („RASD 1.4.pdf“).
- Design Document („DD 1.0.pdf“). [this document]
- Inspection Document. [to be developed]
- Testing document [to be developed]
- Function points document [to be developed]

2 Architectural design

2.1 Overview

In this part, the overall architecture of the system is going to be considered using the top-down approach in order to identify all the subsystems. This approach is used in order to give a brief overview, to make it easier to understand how the system would look like and to separate the functional units in order to make deriving component, deployment and runtime view more understandable.

In what follows, relying on identified usecases in RASD document, the decomposition of application into subsystems is going to be presented – starting from the biggest functional unit – the MyTaxiService itself, and getting closer and closer to the smaller units.

System is separated into following subsystems:

-Unregistered user subsystem – includes login and registration related functions encapsulated into Login and Registration subsystems.

-Registered user subsystem – includes sending taxi request, reports, responding to drive offers, profile modifications and log out. Each of the operations mentioned corresponds to a subsystem that deals with related problem.

-Taxi driver subsystem – the same as previous one, but also includes status availability change and accepting/rejecting drive requests.

-Scheduler subsystem – determines the taxi zones and forwards requests and responses to corresponding queue. This part of the system also acts a mediator between users and taxi drivers.

-Time and cost calculator subsystem – this subsystem is used in order to calculate the estimated waiting time for the taxi and cost. It is used by scheduler during negotiations in order to make drive offers to users.

-Taxi zone determination – calculate the corresponding taxi zone for a given location.

-Developer subsystem – operations related to external API offered to developers. They can virtually access any function from external app.

-Administrator subsystem – gives ability to browse users, edit them, delete them, view their drives and reports or promote them to taxi drivers or downgrade them

-Data subsystem – stores the classes of the necessary entities, deals with database and operations performed over database.

The application will use client-server architectural style in general, with some variations. So, in what follows a brief overview of the separation is given.

Unregistered user, Registered user, Taxi driver, Developer user and Administrator subsystems have their separated counterparts on the both client and server sides. The client side of these subsystems are in interaction with users, while the server-side counterparts handle the user actions and requests. Scheduler, Time and cost estimation and Data subsystems are present on the server side in order to deal with these requests, create responses according to the environment conditions (negotiations between users and taxi drivers, availability status changes, their decisions etc.).

Data subsystem could be located on a separated server, in order to fulfill the security and other non-functional requirements, but this part will be explained in details later, when deployment and architectural style are considered.

Session subsystem is also necessary, as it is needed to handle log in and log out operations. Once users log in, they can use service, without need to log in again after each action. When they finish the use of the application, they can log out.

2.2 High level components and their interaction

Taking into consideration previous conclusions, the high level components and their interactions are derived and described.

As it can be seen, the system consists of clients that belong to different types of users (unregistered/registered, taxi driver, administrator and developer), server which consists of parts that handle the requests from corresponding clients, and data layer part, which is separated from the application part of the server and deals with database queries. In fact, the application part of the server uses data layer part, in situations when it is needed to deal with database – add users, delete users, show reports, update etc.

Unregistered user's actions could be either log in or register. The corresponding part of the server deals with these requests. In case of log in, the Unregistered user manager consults the data layer to check if the user really exists. When register action is taken, application server invokes insert database query and adds new user (assume that data is correct and valid).

When user is logged in, the client changes its appearance, corresponding to the type of user. User that is logged in can send taxi requests, modify profile, send offer responses, logout. The part of the server that deals with typical users is UserManager. If the user that is logged in is taxi driver, the person gets interface that allows to change availability status and also respond the drive requests. Part of the server that deals with taxi drivers is TaxiDriverManager.

Administrators can browse users, view reports, drives, promote to taxi drivers or even revert back taxi drivers to users.

Developers have programmatic interface that lets them call any function from MyTaxiService, except the functions related to administration. Component that handles the API calls is DeveloperManager and belongs to server part of the application.

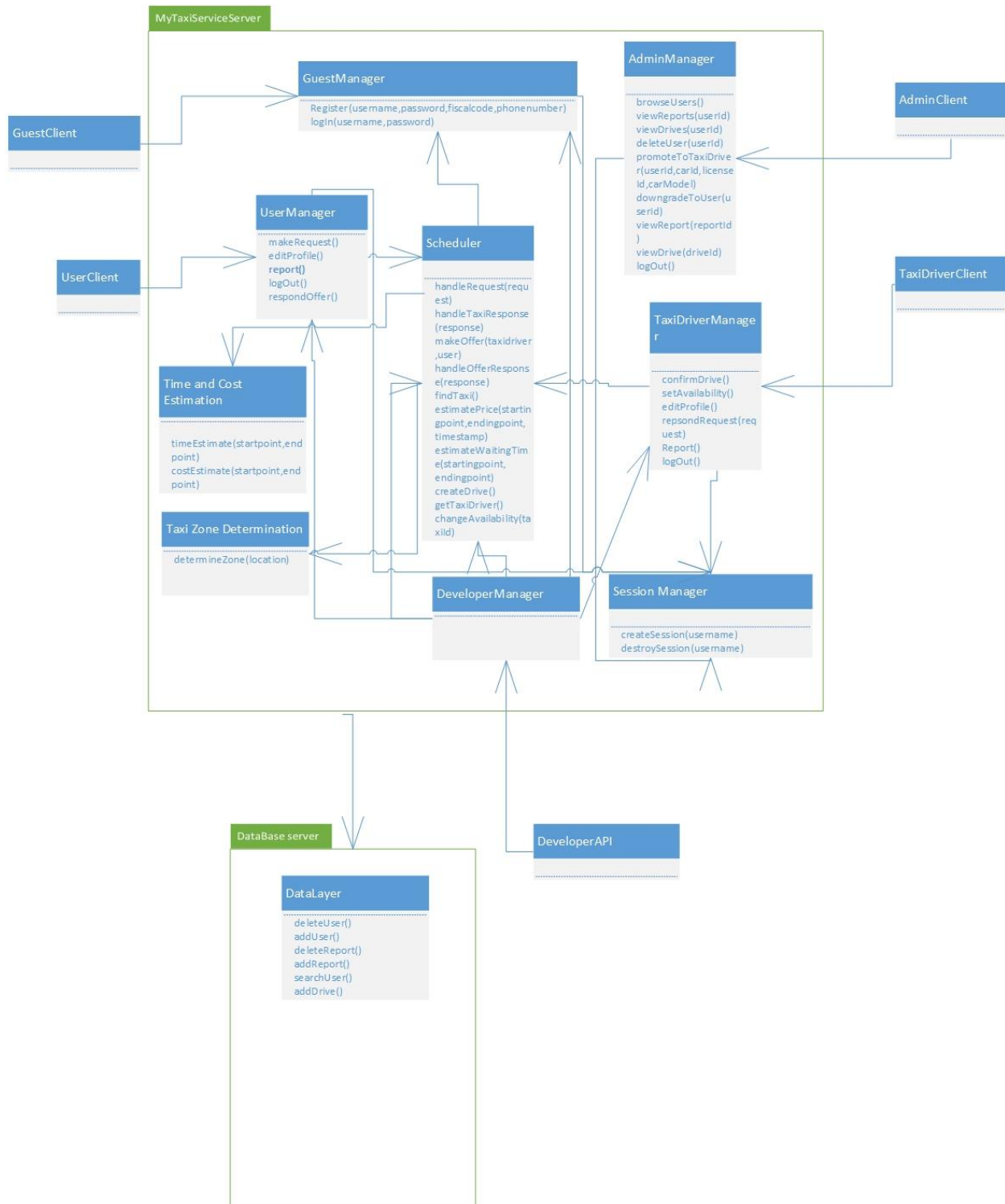
Scheduler is responsible for handling requests, responses, taxi scheduling and estimation of waiting time and price. This part is used by other components of the system, such as UserManager and TaxiDriverManager in situations when it is necessary to handle the negotiations between them.

DataLayer is responsible for actions taken over database and is used by application server of the MyTaxiService. It encapsulates necessary actions that interact with database: search, insert, delete and update queries, so the application server has to use it in order to satisfy requests that need access to the database. For example, to check if there is already user with same username during registration, validity during log in, checking car id validity and fiscal code during taxi driver promotion and registration, adding and viewing reports, browsing users etc.

Session manager is responsible for log in and log out. It creates session, in case of log in, and destroys it, in case of log out. Each client manager uses it in log out, and Guest Manager uses it for log in and register (when new user is registered, he/she is automatically logged in).

On the next page, high level components and their interaction are presented in a diagram.

Notice that some parts are omitted (parameters and attributes), and only the most relevant among them are listed in order to make image easier to understand. For example, Taxi driver Manager contains all the same interfaces as Registered User Manager plus additional interfaces. Only additional interfaces are displayed. Parameters are going to be explained later in component interfaces part.



2.3 Component view

The system follows client-server style for what concerns the management of the interaction with the users (including system administrators, taxi drivers and developers) and interacts, through a

service interface (REST interface, for example) with GPS coordinates service and city government service. At the same time, the server offers REST interface to developers who want to use functionalities of the MyTaxiService in their application in order to extend it. The developers are outside of the system, so their client is not considered.

The server contains main logic and is in charge with interacting with four kinds of clients – RegisteredUserClient, TaxiDriverClient, AdminClient and UnregisteredClient. Moreover, the part of the server that is related to data that needs to be checked if it is valid (fiscal code, driving license, car id) uses external access to city government databases. Furthermore, the clients of users and scheduler use GPS coordinates service (like Google Maps API etc.) to send location data, receive it, manipulate it.

The server consists of the following parts:

- UnregisteredManager is a component which gives ability to unregistered users to register or log in. This part of the server uses external city government databases through REST service in order to check the validity of fiscal code. It also uses DataLayer to check if there is same user already in a database or the user is banned and can't be registered. In case of successful registration, this component uses DataLayer to add new users to database once they are registered. This component, as it can be seen, doesn't use the scheduler, as unregistered users can't make taxi requests.

- RegisteredUserManager is a component which enables users to logout, receives all the messages from users - drive requests, drive offer responses, reports, SOS signal, modify their profile, log out and all other functions that are present in RASD usecase diagrams. This component can access GPS coordinates service in order to determine the user's location when necessary. In order to accomplish all of the operations, this component interacts with the DataLayer and visualises all the necessary elements when it is needed (maps, etc.). Also, this component uses Scheduler in order to serve the users in requests related to taxi drives.

- TaxiDriverManager is similar to previous one, but also includes some additional options, like accepting or rejecting a drive request and changing the availability status, in order to support TaxiDriver clients.

- AdminManager is supporting AdminClients, and gives them ability, using the DataLayer, to browse users, read reports, promote users to taxi drivers, prevent some users from being drivers, delete users in case of bad behaviour, and log out. Notice that Admin manager doesn't use Scheduler, as it isn't necessary (Administrators can't make taxi requests).

- Scheduler is the core component of the system that cross-references the location data and the system messages in order to dispatch a taxi from the corresponding taxi zone using the DataLayer. It uses GPS coordinates service in order to determine the taxi zone of the current user, deals with taxi queues and forwards the request to another taxi in current taxi zone if there is available taxi. It also receives the drive offer responses from users and creates drive events. Scheduler uses results from time and cost calculator when makes drive offers to users.

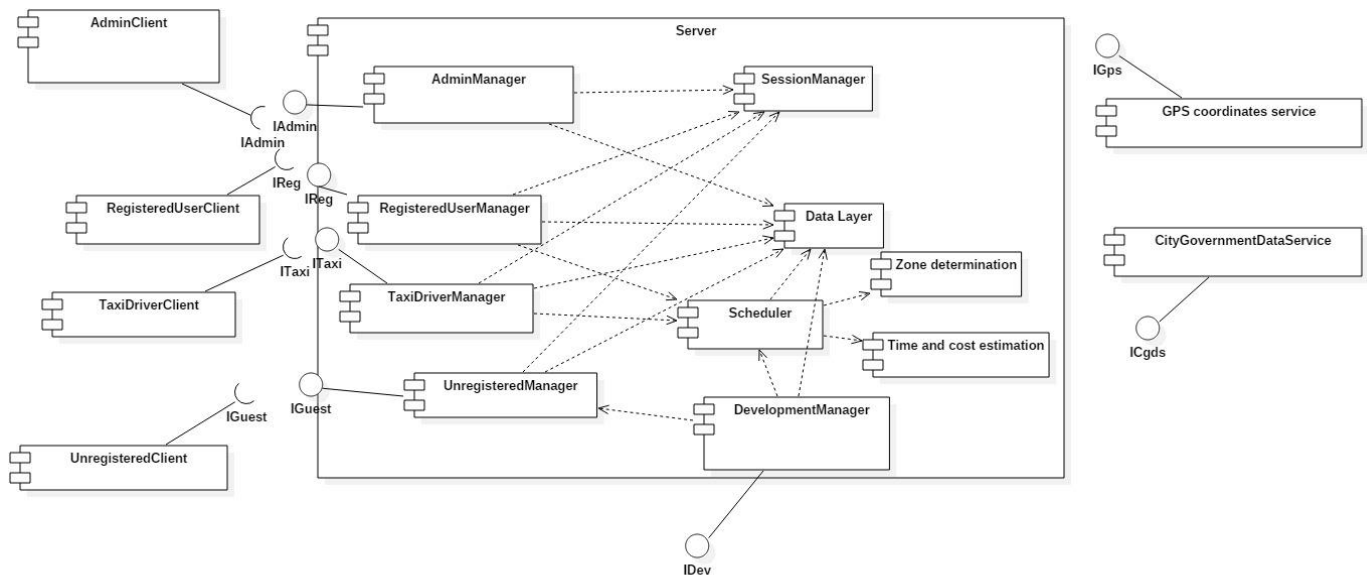
- Time and cost calculator. It estimates the time and price for a taxi drive. It uses data from DataLayer in order to have enough information to do its part of job. By the way, the Scheduler uses this component when it makes a drive offer to user - so it makes this system fair. Users can accept or reject the offer if the price is too high.

- Zone determination – component that determines the taxi zone corresponding to a given location. Uses external GPS and Maps API.

-DataLayer encapsulates the entities relevant to the system – different messages (Request, Response, Report), drive event, taxi zone, different types of users – User, Taxi Driver, Administrator and deals with their storage into database.

-DevelopmentManager is a part of the system which offers access to basic system functions to developers that are outside the system via some kind of service (REST, for example) in order to give them ability to embed or extend MyTaxiService in their applications. They can externally request a taxi, for example by API or register or log in from external application (for example, Facebook).

-SessionManager, a part of the system that is responsible for creating and destroying sessions (log in- create session, log out-destroy session). It is necessary to have this part of the system included, because it distincts logged in and logged out state.



2.4 Deployment view

In this section, the deployment of the components and processes is presented. Using this information, reader can find out where each of the components is located and executed and details about communication between nodes. As it is already told, the application consists of server and client parts. Server side has web and application tier separated. Data tier is also separated from them and is used to handle the database operations. This kind of separation gives option for better security implementations, as it has been told in RASD.

In what follows, the system deployment is going to be described and illustrated.

Client could be either mobile phone which runs mobile application or personal computer running web browser.

In case of mobile phone the application communicates directly with application server via REST¹ service. JAX-RS² is used for providing REST service in the application.

In case of web browser, we have HTTPS³ communication with WEB server which is running Glass Fish with Java Server Pages and has Session Bean that is responsible for session.

Then, the web server communicates via REST with application server. JAX-RS is on top layer for communication, while the business logic itself is contained in Enterprise Java Beans. JPA⁴ is used for persistence and provides object-relational mapping which gives ability to translate SQL query results into objects and use them.

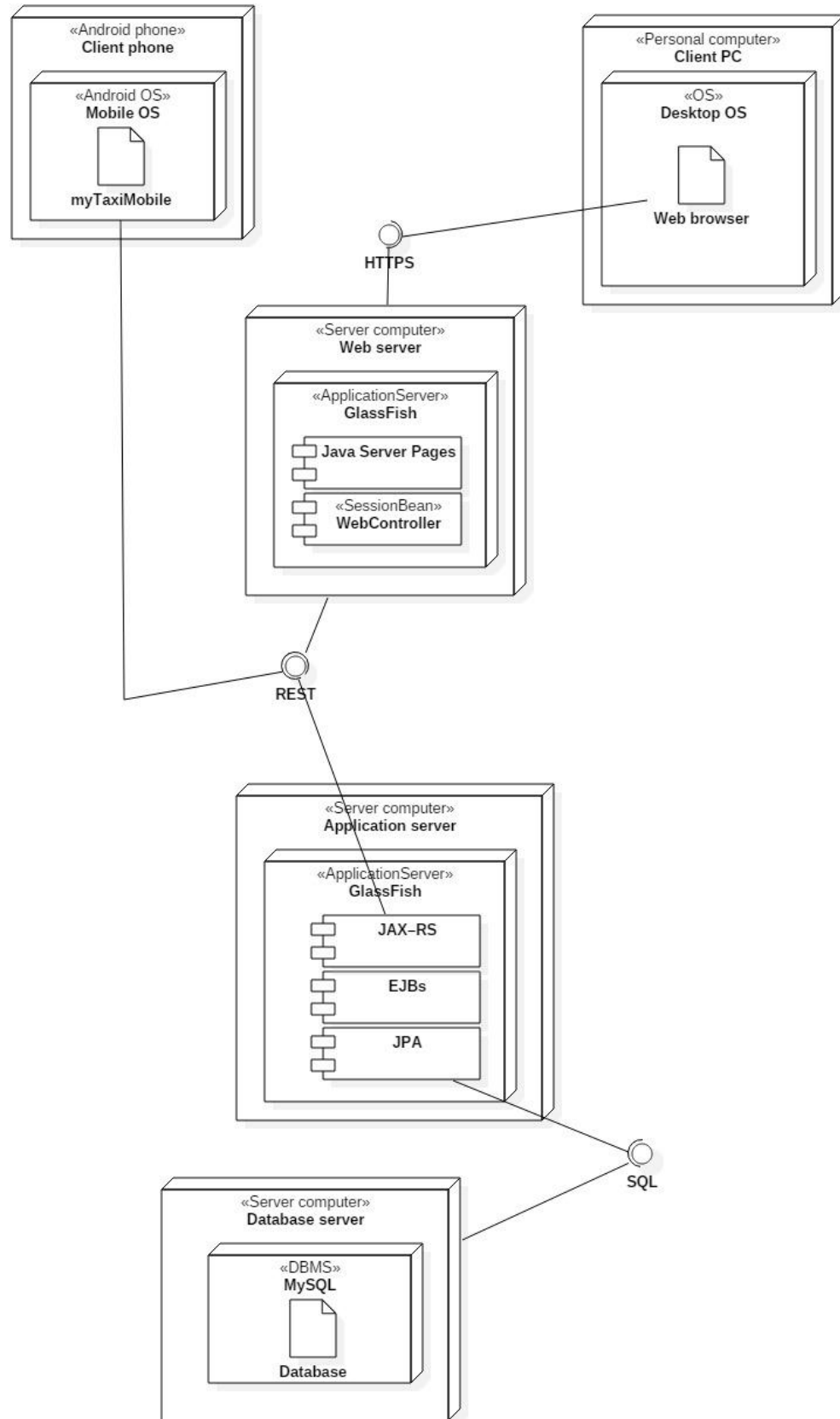
And finally, there is a database server which runs MySQL and is responsible for database operations performed by the application server. Queries are formed and executed according to the business logic, and results are transformed into objects using object-relational mapping. So, in this form (objects), it is easier to use the query results and present them to users.

¹ REST (REpresentational State Transfer) is an architectural style, and an approach to communications that is often used in the development of Web services.

² JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.

³ HTTPS (HTTP over SSL, and HTTP Secure) is a protocol for secure communication over a computer network which is widely used on the Internet. HTTPS consists of communication over Hypertext Transfer Protocol (HTTP) within a connection encrypted by Transport Layer Security or its predecessor, Secure Sockets Layer. The main motivation for HTTPS is authentication of the visited website and protection of the privacy and integrity of the exchanged data

⁴JPA:The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

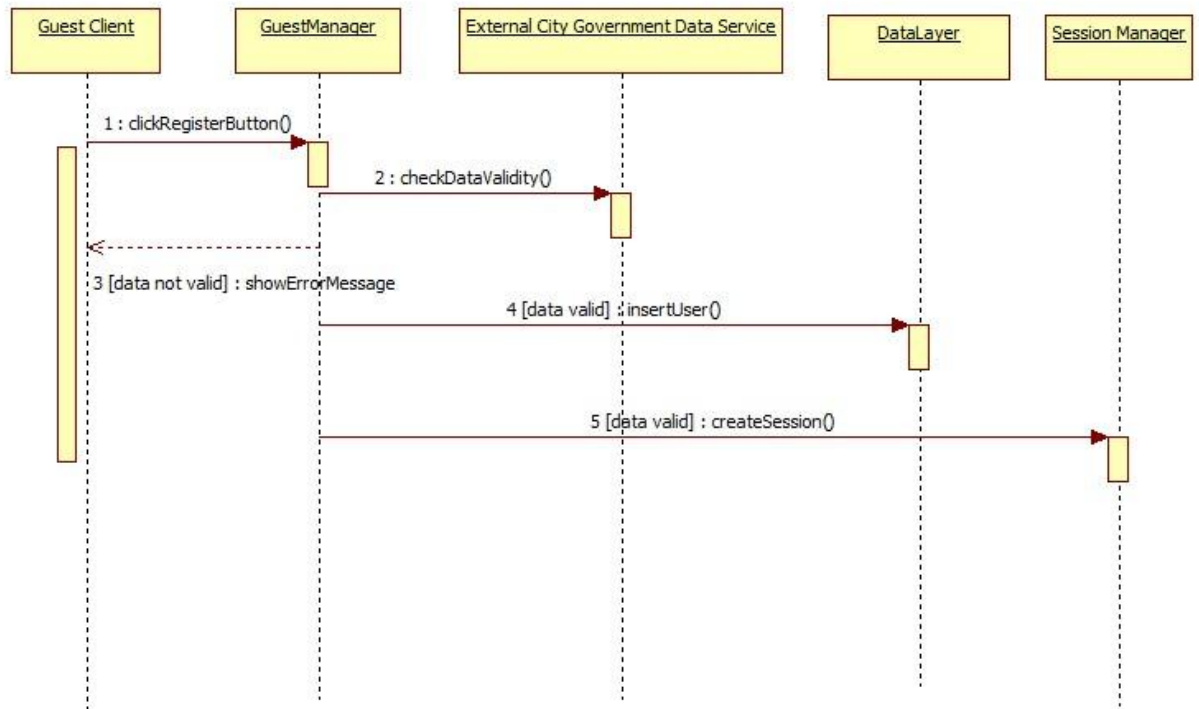


2.5 Runtime view

This section deals with the components interaction during runtime. Each time some action is performed in the application, it is translated to a set of component interactions. To illustrate runtime component interactions, sequence diagrams are going to be used.

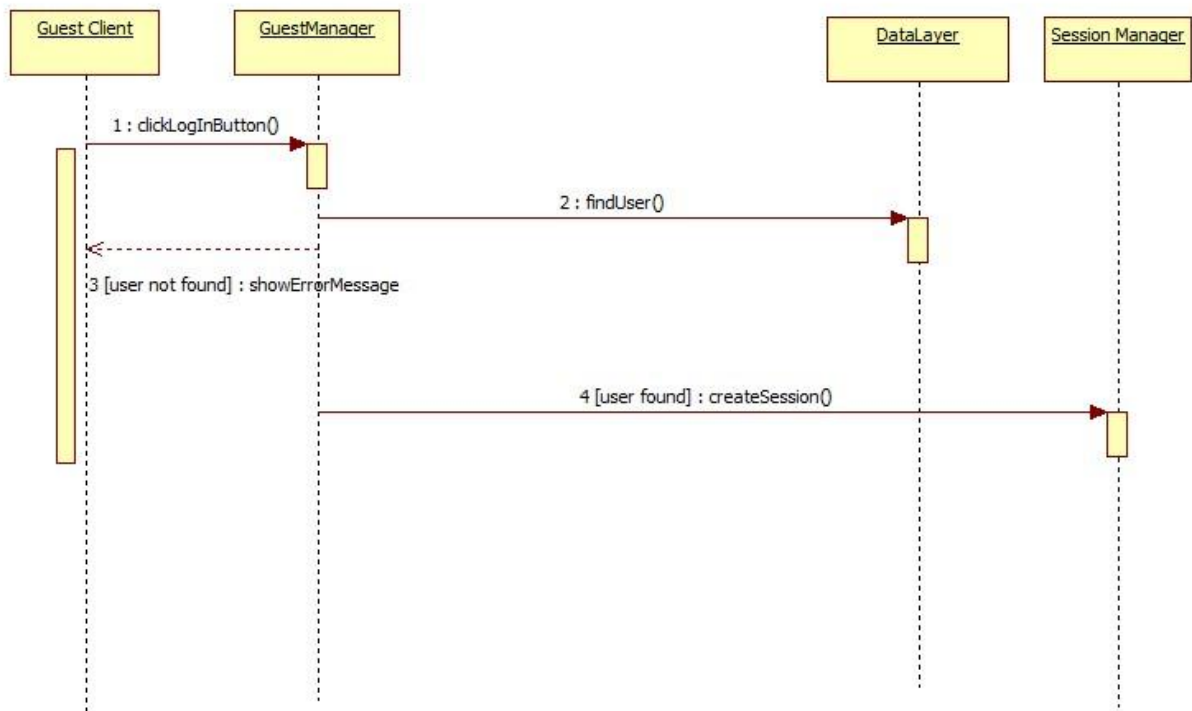
Register user

New users can register by entering the necessary data. When users finish entering their data, they click on register button. GuestManager then checks if data is valid (fiscal code and identity correspondence) using external database provided by City Government. If data is not valid, guest is informed in his/her client. Otherwise, GuestManager interacts with DataLayer and inserts new user into database. After that, user is automatically logged in, so the new session is created in interaction with Session Manager.



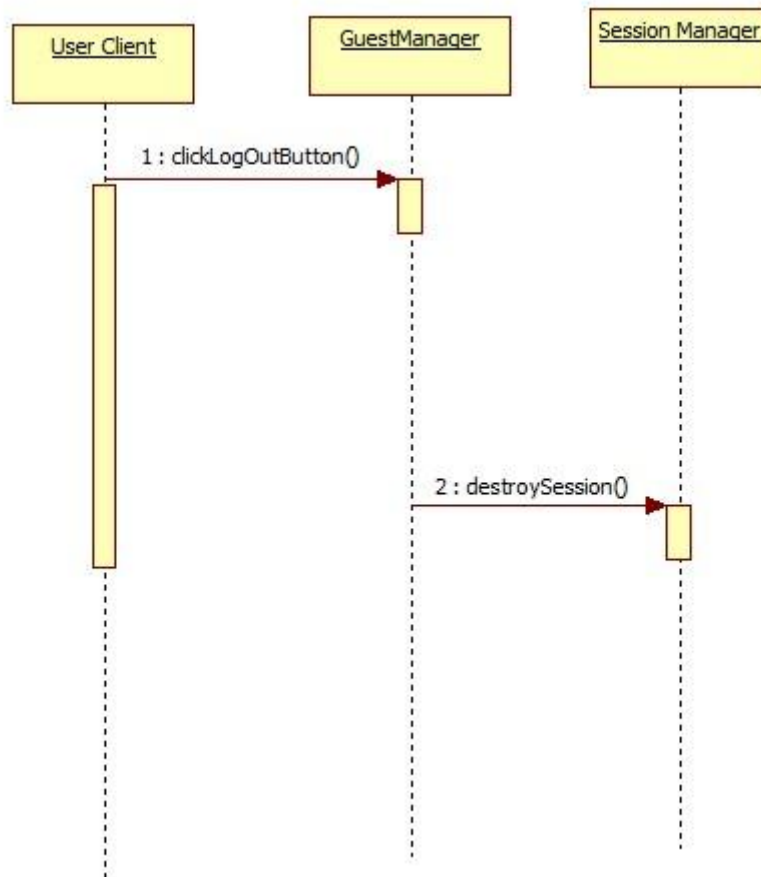
Log in

If user enters valid combination of username and password, he/she is going to be logged in and new session is created.



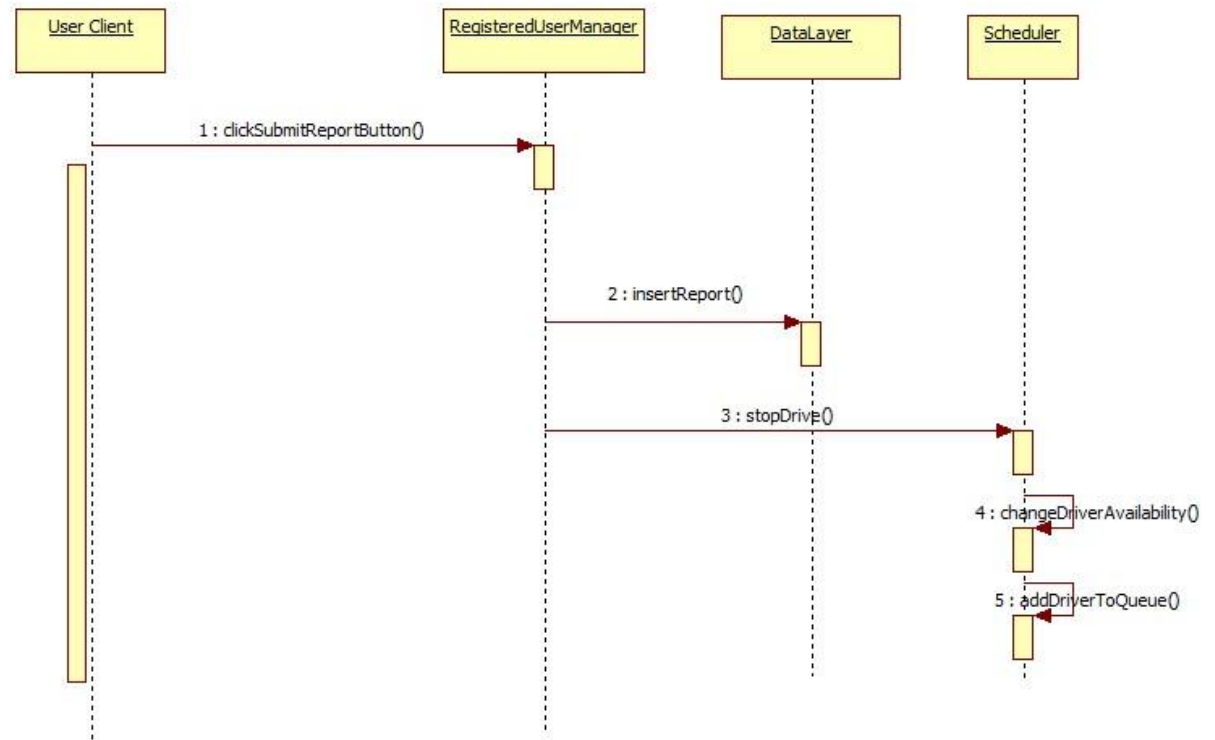
Log out

If user decides to log out, session is destroyed.



Report user

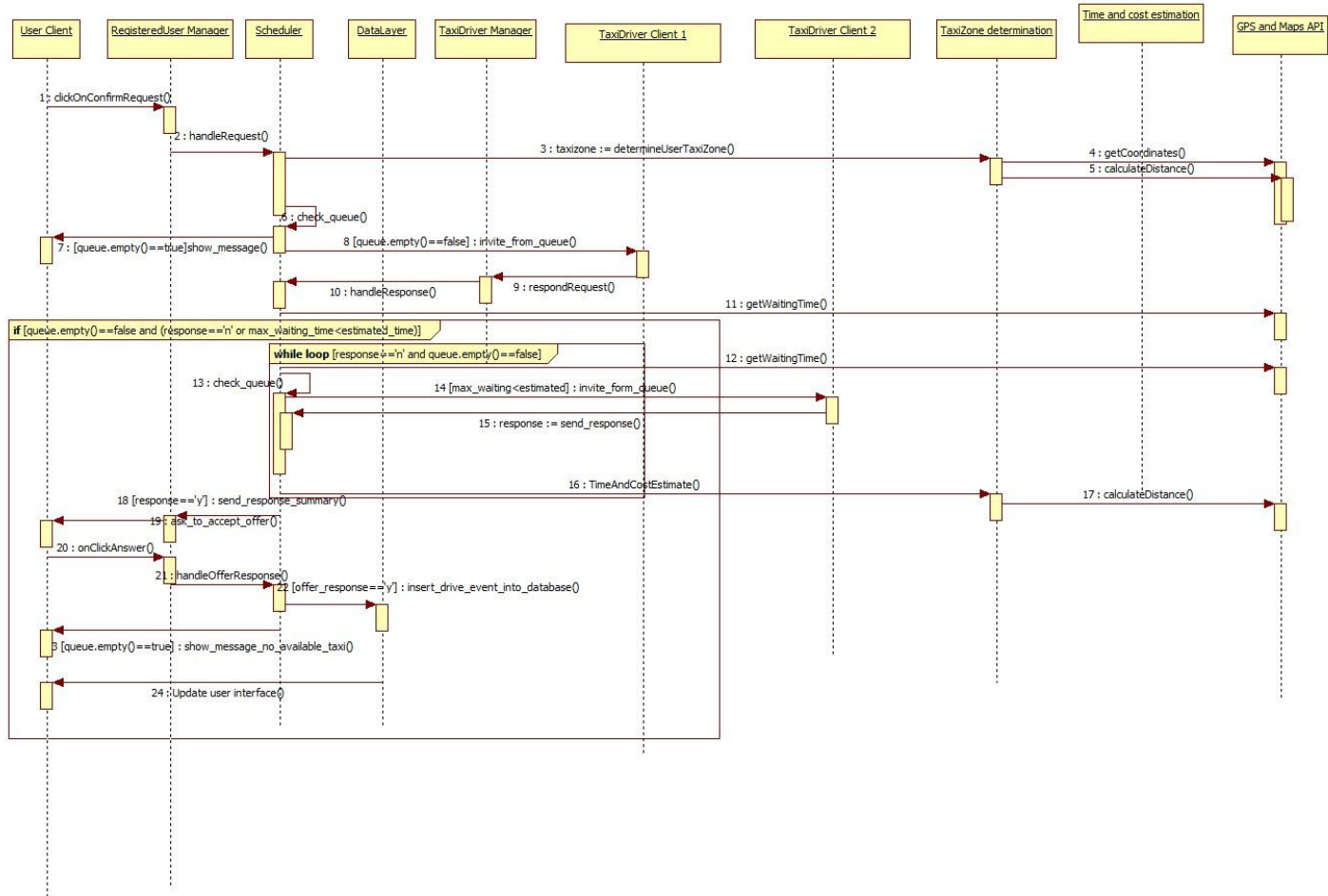
User can report another user typing in the reason and clicking the SubmitReport button. After that, RegisteredUserManager component interacts with DataLayer, executing query that inserts previously written report into database. It is also needed to stop drive event, and taxi driver is available again and added to queue. In this case, RegisteredUser Manager interacts with Scheduler, while the scheduler itself changes driver availability and adds it to taxi queue.



Promote to taxi driver

Administrator can promote users to taxi drivers. Administrator first finds the user he/she wants to promote to taxi driver, then enters the data. After submitting the data, car id number and driving license validity are checked using the external database. If data is not valid, administrator gets message. If data is valid, update query is performed in database (interaction between AdminManager and DataLayer).

component responsible for GPS maps API is used, while cost estimation is part of the system. If users accepts the offer, then, new drive event is inserted into database. But, in situation where user rejects the offer, than Scheduler stops searching for taxi.



2.6 Component interfaces

In this part of the document, component interfaces are going to be described. In section 2.3, interfaces between components are presented, but not in details. In what follows, each of the component interfaces is going to be broken into functions whose parameters and return values are going to be described based on what is presented already in sections 2.2 and 2.3 inside this document.

Interfaces:

1. IAdmin

This interface is used by admin users and contains functions that allow administrators to:

1.1 Browse users

Administrators can browse users by a given criteria (name, username, fiscal code, etc.). If one of the fields is left blank, this part is not going to be used for SQL query formation.

When administrator clicks the browse button in web or mobile app form, button click event handler is activated. This handler looks like:

```
void onClickBrowse(Event e)
```

Let's consider Event as an object which contains the current containers and GUI components values at the moment of activation related to the event that triggered the handler.

And, inside this function, another one is called:

```
Users[] browseUsers(string username, string firstName, string lastName, string fiscalCode)
```

Where the function itself returns an array of User objects using the object-relational mapping. \

After a call of this function, user view is updated according to the results obtained from the formed query.

It uses data layer in a following way:

For example, this function forms query like this when function is called in this form `browseUsers('','USERNAME1','','')`

```
SELECT      USERNAME,FIRSTN,LASTN,FISCAL      FROM      USERS      WHERE
USERNAME='USERNAME1'
```

As a result, array of Users that satisfy the criteria are returned, so the GUI components can be updated.

1.2 View reports

Administrators can view the reports by users, by selecting the user whose reports want to browse. There would be some event handler that would extract the selected user id:

```
void onSelectUser(Event e)
```

Where e is encapsulation of the event. From this object, ID of the selected user could be obtained in this way:

```
id=e.SelectedItem.text
```

It uses datalayer and executes query which shows all the reports related to selected user like:
`SELECT * FROM REPORTS WHERE SENDERID=USERID :`

While function:

```
Report[] viewReports (string userid)
```

returns an array of report objects corresponding to a particular user. Objects are formed using object-relational mapping.

1.3 View drives

Administrators can view the reports by users, by selecting the user. There would be some event handler that would return take the selected user id from GUI component:

```
void onSelectUser(Event e)
```

Where e is encapsulation of the event. From this object, ID of the selected user could be obtained in this way:

```
id=e.SelectedItem.text
```

It uses datalayer and executes query like: `SELECT * FROM DRIVES WHERE SENDERID=USERID :`

While function:

```
Drive[] viewReports (string userid)
```

returns an array of drive objects corresponding to a particular user. Objects are formed using object-relational mapping.

1.4 View drive and view report

These component interfaces are very simple and similar to previous, but, return only one object,instead.

```
void onSelectDrive(Event e)
```

```
void onSelectReport(Event e)
```

Where e is encapsulation of the event. From this object, ID of the selected report/drive could be obtained this way:

```
id=e.SelectedListItem.text
```

After that, function is called inside controller

```
void viewDrive(string driveId) or
```

```
void viewReport(string reportId)
```

It uses datalayer and executes query like: `SELECT * FROM DRIVES WHERE ID=DRIVEID`

`SELECT * FROM REPORTS WHERE ID=REPORTID :`

While function:

Drive viewDrive (string userid)

Report viewReport (string userid)

returns drive/report object . Objects are formed using object-relational mapping. After that, information obtained could be used in order to update the Admin GUI.

1.5 Delete user

This interface uses a similar approach with event handler and interface (void onSelectUser(Event e)), but the most important part is interface to the data layer:

```
bool deleteUser(string userid)
```

which executes query

```
DELETE FROM USERS WHERE ID=USERID
```

If operation is successful, return value is true. Otherwise, false is returned.

In case where we delete the user, we should add it to another table in order to keep track of blacklisted users. So, there would be another table called BLACKLIST, where each record consists of fiscal codes of banned users.

So, before delete, this query could be executed:

```
INSERT INTO BLACKLIST VALUES(ID)
```

But, in case if we want to prevent users from making account again, we can keep them in database, but with their attribute „Blacklisted“ set to 1. So, the query would be update instead of delete:

```
UPDATE USERS SET BLACKLISTED=1 WHERE ID=USERID
```

Promote to taxi driver

This interface allows administrators to promote users to taxi drivers, by entering the necessary data for this procedure. The necessary data contains car identification and driving license id, while car model is optional.

Situation is similar here

```
void onUpdateUser(Event e)
{
    userid=guiPromote.textBoxUserId.text
    carId=guiPromote.textBoxCarId.text
    carModel=guiPromote.textBoxCarModel.text
    licenseNumber=guiPromote.textBox.LicenseNumber.text
    if (isEligible(carId,licenseNumber)) then
        updateUser(userid,carId,carModel,licenseNumber)
    else
        showMessage("Driver not eligible")
}
```

The last interface:

```
bool updateUser(userid,carId,carModel,licenseNumber)
```

forms query to data layer

```
UPDATE USER WHERE ID=USERID SET DRIVER='Y',
CID=CARID,CMODEL=CARMODEL,LNUM=LICENSENUMBER
```

Return value is set to true, if such user exists and data is valid, otherwise return value is set to false.

It should be mentioned that :

Bool isEligible(string carId, string licenseId) is interface to external government database that checks the validity, as mentioned in 2.3.

1.6 Downgrade taxi driver

This one is similar to previous:

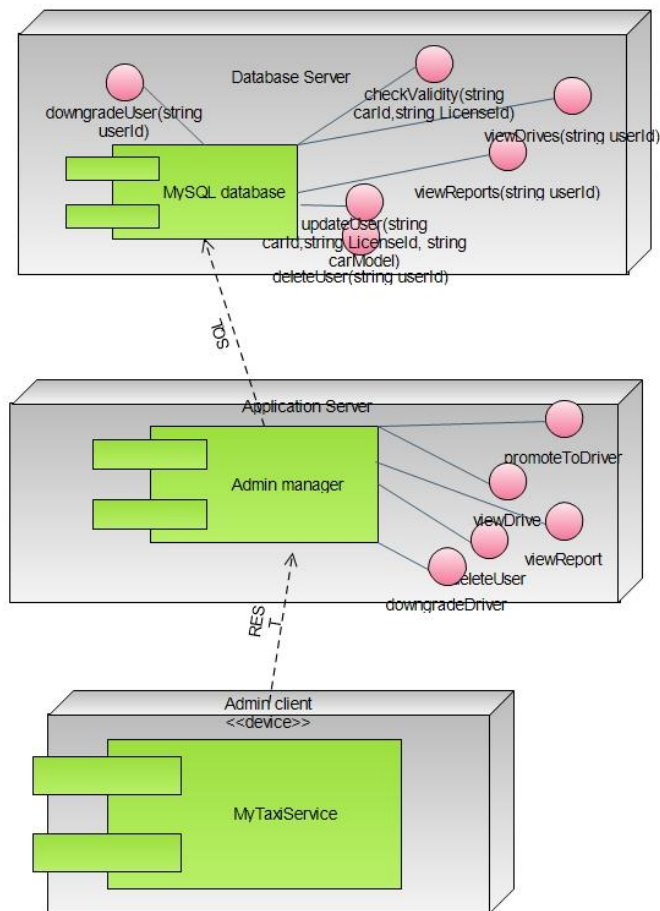
```
bool downgradeUser(userid)
```

But, the query is different

```
UPDATE USER SET DRIVER='N',CID=' ',CMODEL=' ',LNUM=' ' WHERE
ID=USERID
```


On next page, you can see a whole picture that presents interfaces used by administrator user and how components are connected in this case. Please notice that log out is omitted (and session manager).

Admin management



2. IGuest

This is, in fact, interface used by unregistered users when they come to site. They can either register or login.

2.1 Register

Guests can register entering the necessary data.

Event handler is activated:

```
void onClickRegister(Event e)
```

And, inside this function, another one is called:

User registerUser(string username, string firstName, string lastName, string fiscalCode, string password, string confirmPass, string picturePath, character Gender, string mobilePhone)

Field picturePath is optional and could be blank.

In fact, inside event handler we have something like this:

```
username=guiRegister.textBoxUserName.text;
```

```
firstName=guiRegister.textBoxFirstName.text;
```

and so on, until the last parameter

```
mobilePhoneNumber=guiRegister.textBoxMobile.text;
```

Now, the data is checked (fiscal code validity) using external database, and using MyTaxiService database (to check if there is the same user or is banned)

After that:

```
User newUser=RegisterUser(username, firstName, lastName, fiscalCode, password, confirmPass, picturePath, Gender, mobilePhoneNumber)
```

Where the function itself returns a User object and destroys the existing Guest object. It uses data layer in a following way:

```
INSERT INTO USERS VALUES (ID, USERNAME,FNAME,LNAME,FCODE,PASSWORD,PICPATH,GENDER,MOBILE)
```

After that, if data is valid and data doesn't overlap with the existing user, new user is added into a database.

Log in is automatically performed after registration, so the UnregisteredManager is in interaction with session component. New session is created, which corresponds to the username of the new user that is registered, using the interface towards session manager:

```
void createSession(string username)
```

2.2 Log in

Guests can log in. After that, Guest object is changed with the corresponding object, which depends on type of the credentials – do they belong to User, Taxi driver, Developer or Administrator,

First, event handler is activated:

```
void onClickLogin(Event e)
```

And, inside this function, another one is called:

```
User loginUser(string username,string password)
```

In fact, inside event handler we have something like this:

```
username=guiHome.textBoxUserName.text;
```

```
password=guiHome.textBoxPassWord.text;
```

Now, data is checked, if it exists in database

After that:

```
User newUser=Login(username, password)
```

Where the function itself returns a User object (or other, more specific user type) and destroys the existing Guest object, in order to change it with new one.

Its interface to data layer is SQL code which looks like:

```
SELECT      *      FROM      USERS      WHERE      USERNAME=USERNAME      AND  
PASSWORD=PASSWORD
```

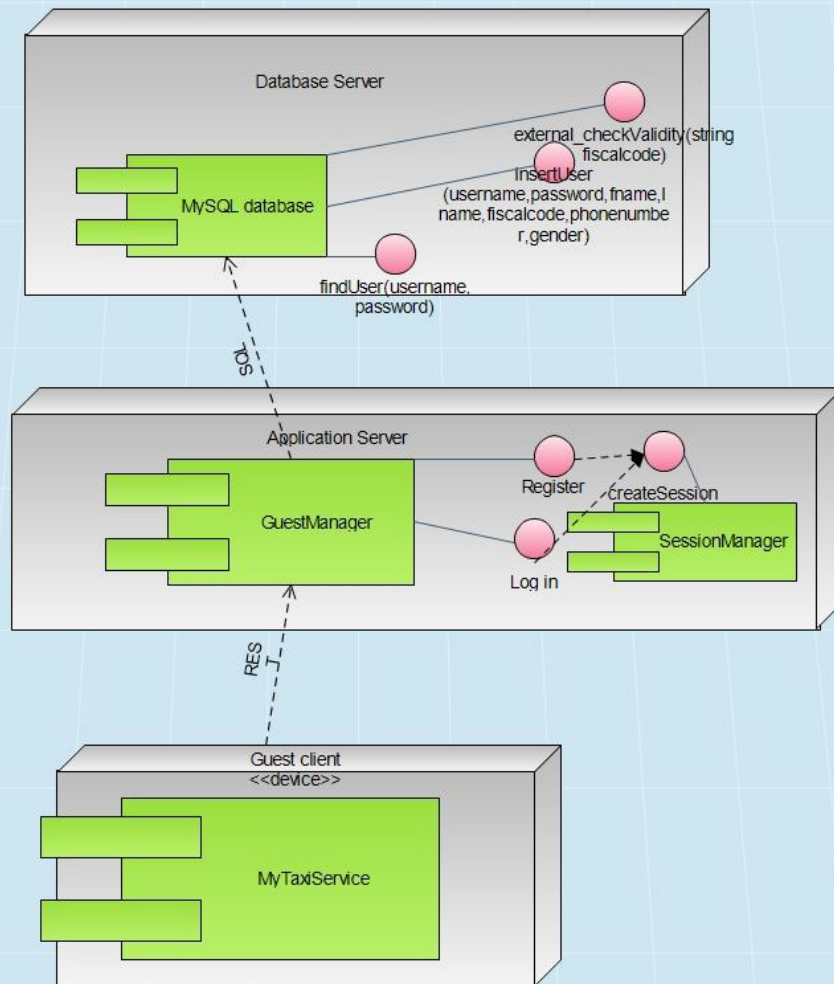
Using the object-relational mapping layer, the object is formed. Its attributes ADMIN, USER, DRIVER and DEVELOPER are checked. According to value of this attribute (not null), new corresponding object is created.

Also, the interaction with session manager is performed. New session is created

```
void createSession(string username)
```

On the next page, you can see picture that presents connection between components of the subsystem that deals with guests (unregistered users).

Guest management



3. IReg

This is interface related to registered users. They can edit profile, make requests for a taxi, log out, and respond the drive offer.

3.1 Edit profile

Guests can register entering the necessary data.

Event handler is activated:

```
void onClickModify(Event e)
```

And, inside this function, another one is called:

```
User modifyUser(string username, string firstName, string lastName, string fiscalCode, string password, string confirmPass, string picturePath, string mobilePhone)
```

Field picturePath is optional and could be blank.

In fact, inside event handler we have something like this:

```
firstName=eguiEditUser.textBoxFirstName.text;
```

and so on, until the last parameter

```
mobilePhoneNumber=guiEditUser.textBoxMobile.text;
```

After that:

Old user object is changed with new (modified):

```
user=ModifyUser(user.username, firstName, lastName, fiscalCode, password, confirmPass, picturePath, Gender, mobilePhoneNumber)
```

Where the function itself returns a User object and destroys the existing Guest object. It uses data layer in a following way:

```
UPDATE                                USERS                                SET
FNAME=FIRSTNAME,LNAME=LASTNAME,FCODE=FISCALCODE,PASSWORD=PASS
WORD,PICPATH=PICTUREPATH,MOBILE=MOBILEPHONUMBER                WERE
USERNAME=USER.USERNAME
```

User data is updated in database, and new User object is in use now.

3.2 Report

Users can report each other.

Event handler is activated:

```
void onClickReport(Event e)
```

And, inside this function, another one is called:

```
Report newReport(string username1, string username2, string reason)
```

```
username1=user.username
```

```
username2=getTaxiDriver(username1)
```

```
reason=guiReport.textBoxReason.Text
```

Inside this, RegisteredUserManager is using interface towards Scheduler:

```
TaxiDriver getTaxiDriver(string username)
```

This interface returns taxi driver which corresponds to the drive of current user who is writing a report.

And finally, the interface towards DataLayer is used:

```
INSERT INTO REPORTS VALUES(ID,USERNAME1,USERNAME2,REASON,TIMESTAMP)
```

After that, new report is inserted into database and can be viewed later by the administrator.

3.3 Log out

This function is same for admin,taxi driver and typical user.

Event handler is activated:

```
void onClickLogout(string username)
```

This function interacts with Session manager and calls function

```
void destroySession(string username)
```

After that, object corresponding to user that was previously logged in is destroyed and changed with Guest object.

3.4 Request taxi

One of the most complex and most important parts of the application – sending taxi request.

Again, we start from the event handler that is activated when user clicks on button to submit the request

```
void onClickSubmitRequest(Event e)
```

As we have GUI where map is displayed and user selects the location, GPS coordinates are extracted from the event.

```
location=guiMapScreen.map.location
```

Also, the maximum waiting time is extracted.

```
maxTimeWaiting=guiMapScreen.textBoxMaxTime.Text
```

After that, a interface towards Scheduler is used by RegisteredUserManager :

TaxiDriver handleRequest(Request request), which returns an available taxi driver who accepted the request, if there is one. Otherwise, it returns NULL.

3.5 Respond offer

User can accept or reject offer.

To form an offer, request is analyzed, taxi is found, and offer is composed. Price and time are estimated.

The user accepts or rejects the offer by pressing the corresponding button

Void onResponseOffer(Event e)

Response is formed according to the answer.

response=New Response(e.ClickedButton.text)

After that, the RegisteredUserManager communicates with Scheduler by :

Drive handleResponse(Response response)

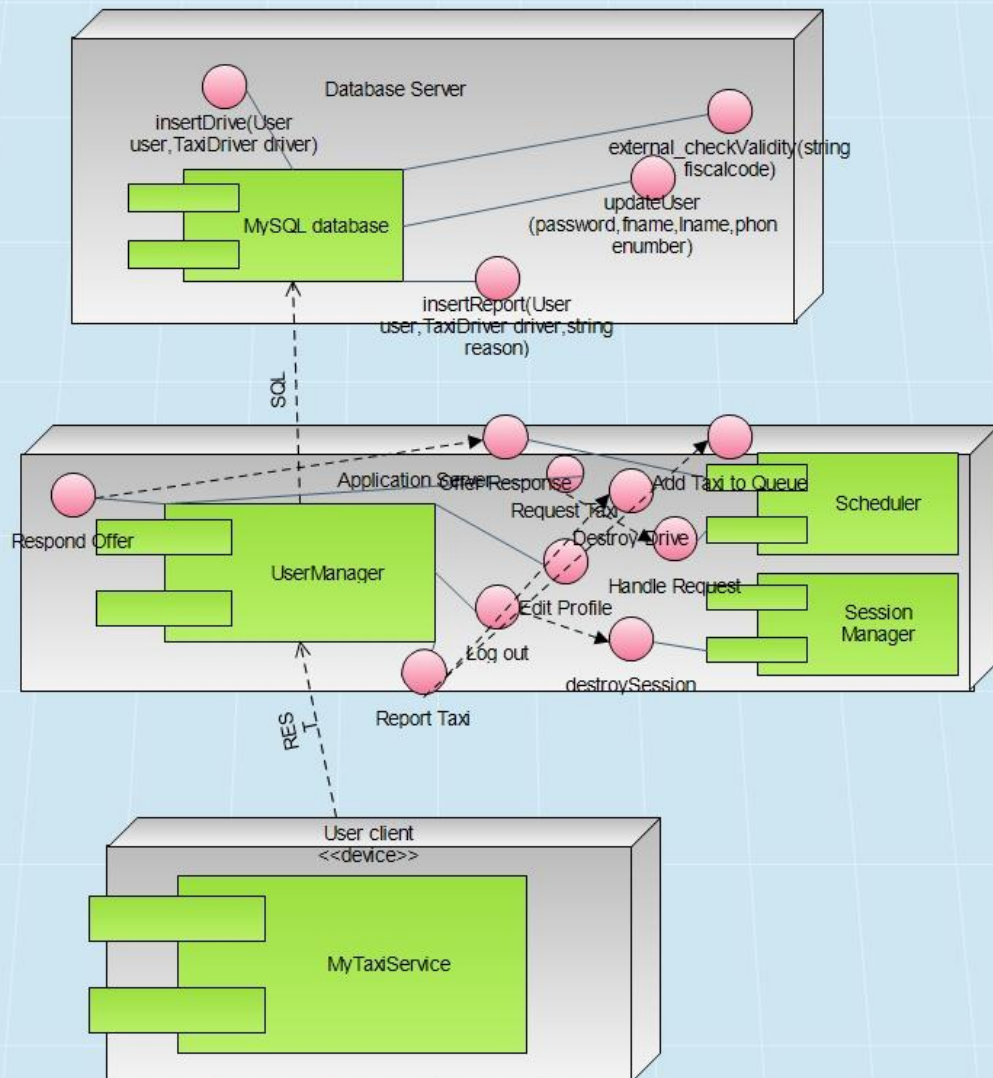
The Scheduler will handle the response according to the answer. If answer is „NO“, return value is set to NULL, but in case of „YES“ answer, new drive event is created and inserted into database by datalayer SQL interface:

```
INSERT          INTO          DRIVES          VALUES
(ID,TIMESTAMP,SENDER,RECEIVER,STARTPOINT,ENDPOINT)
```

SENDER- user who sent a request

RECEIVER- a taxi driver who accepted a drive request

User management



4. ITaxi

Now, let us consider Taxi Manager interface. It is very similar to IReg. In fact, it includes all the functionalities by IReg, but also includes some new interfaces related to request accept or reject.

4.1 Change availability

Each taxi driver could either be available or not. After this action, availability state becomes its complement.

When taxi driver pushes the switch, event handler is activated

```
void onAvailabilityChange(Event e)
```

After that, interface towards Scheduler is used

```
bool changeAvailability(string taxiId)
```

Scheduler changes availability of the taxi and removes it from or adds it to corresponding queue.

4.2 Respond request

Taxi driver accepts or rejects the request for a drive. Taxi driver has interface to TaxiDriverManager to respond the request by accepting or rejecting it.

```
bool onClickButton(Event e)
```

```
{  
    if (e.ClickedButton.text=="Yes")  
        return true  
    else return false  
}
```

TaxiDriverManager contacts the Scheduler by interface that can handle the response

```
TaxiDriver handleTaxiResponse(Response response)
```

Return value is null if request is rejected, otherwise, the response is Taxi Driver that accepted the request.

If answer is positive, then Scheduler will make an offer and forward it to user. Two functions will be used in order to estimate time and price:

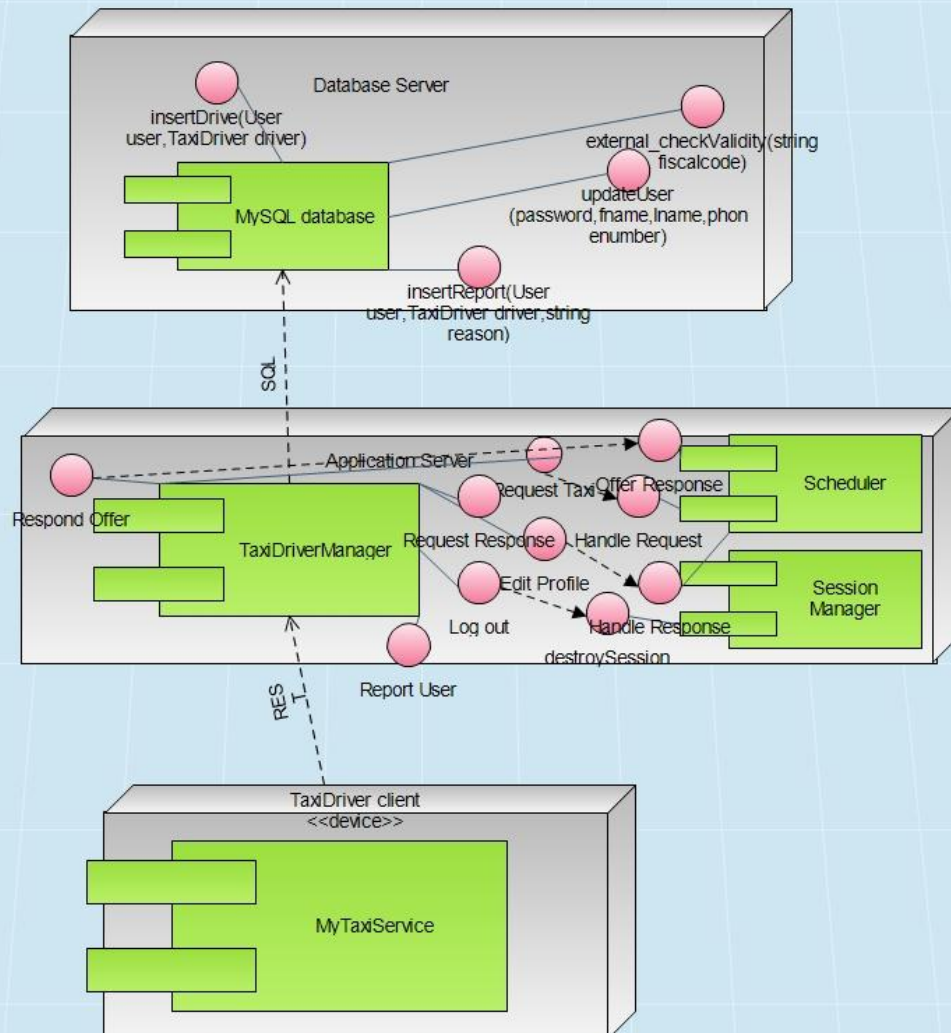
```
float estimatePrice (startpoint, endpoint, timestamp)
```

Return value is price in euros. In section 3.3 (Cost estimation algorithm) this function is going to be defined completely.

```
time estimatedWaitingTime(startpoint, endpoint)
```

This interface is used to estimate how much user will have to wait until the taxi comes. After that, the Scheduler makes offer to the user. User can either accept or reject the offer.

Taxi management



5. IDev

We shouldn't forget that the application will have to offer API to developers. This API could give developers a chance to make their own applications that use MyTaxiService interfaces. In fact, they can use any of the interfaces provided and make application that either extends or uses MyTaxiService. Of course, these developers should be registered by government in order to avoid misuse of this kind of freedom.

6. IGps

This is typical GPS API. We assume that it implements some basic operations, such as:

6.1 float distance(location startpoint, location endpoint)

This function returns distance between two GPS points.

6.2 Location getUserLocation(string id)

This function returns current GPS coordinates of the user, who is identified by device id number.

6.3 Time estimateTime(Location startpoint, Location endpoint)

This function estimates time required to travel by car from one point to another (Google Maps Api-alike function)

7. ICgds

City government database service is external interface to government's databases outside of the system that are related to data validity check, according to legal regulations.

It includes fiscal code check, car id check and driving license check.

7.1 bool isValidFiscalCode(string fiscalCode)

If the fiscal code checked is valid, the answer is true, otherwise it returns false.

It is used during registration by the component that is responsible for user registration.

7.2 bool isValidCarId(string carId)

It is used during promotion to taxi driver to check the car id validity.

7.3 bool isValidLicenseNumber(string licenseNumber)

It is used during promotion to taxi driver to check the driving license validity.

On the next page, you can see a brief overview of the connections and interfaces between components previously mentioned.

2.7 Selected Architectural Styles and Patterns

2.7.1 Introduction

As it has been told already in the RASD document, the application will be released as a both web and mobile application. The application will provide API that could help other developers extend its functionality by developing additional services using the MyTaxiService functions as a primitives. Users can access the service both using mobile web application and browser-based web application. In order to use application as a taxi driver, taxi driver user has to login into mobile version of the application (because of the use of GPS sensor). Administrator's application is targeted to be a web browser application, but will have also a mobile counterpart.

Considering the statements above, and according to the part 3.6.3.4 in RASD document version 1.4, the overall architectural design of the application will inherit the constraints of the programming languages, frameworks, technologies and communication interfaces used.

So, in what follows, the architectural design will be explained in terms of the technologies used – describing the architectural styles and patterns used and how their usage reflects on the design of this system.

2.7.2 Multitier Architectural Style

The selected architectural style is multitier based on Java Enterprise Edition implementation of this architectural style.

In what follows, the advantages of this architecture and reasons for the selection are going to be explained.

The main benefits of the N-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.

Java Enterprise Edition is designed to develop large-scale and multi-tiered applications that are scalable and meet reliability conditions and are secure at the same time.

According to RASD, considering these and all the other functional and especially the non-functional requirements (RASD 1.4, part 3.6), it could be concluded that JEE is more than acceptable solutions in terms of these requirements.

MyTaxiService is going to be a multitier application that is also scalable (offer service to thousands of customers at the same time), but reliable and secure at the same time, maintaining high availability.

Server side needs to meet the special non-functional requirements in terms of security (RASD 1.4, part 3.6.4.3) – so the server side needs to be split into web, business logic and database part. Java EE offers the separate client, web, business and database tier, which is exactly what is needed. So, this will give ability to place firewalls between each two parts and make application secure and meet the security requirements.

So, in what follows, the core concept of JEE is going to be explained and the overall idea of the multitier implementation using JEE in terms of this application.

Multitier architectural model, in this case, consists of:

- Client tier that is running on the client machine. It contains Application Clients and Web Browsers and it is the layer that interacts directly with the actors. The client machine could be either mobile phone running application or web browser or personal computer running web browser in this case. Taxi drivers, according to RASD 1.4, part 2.1. must use mobile application.

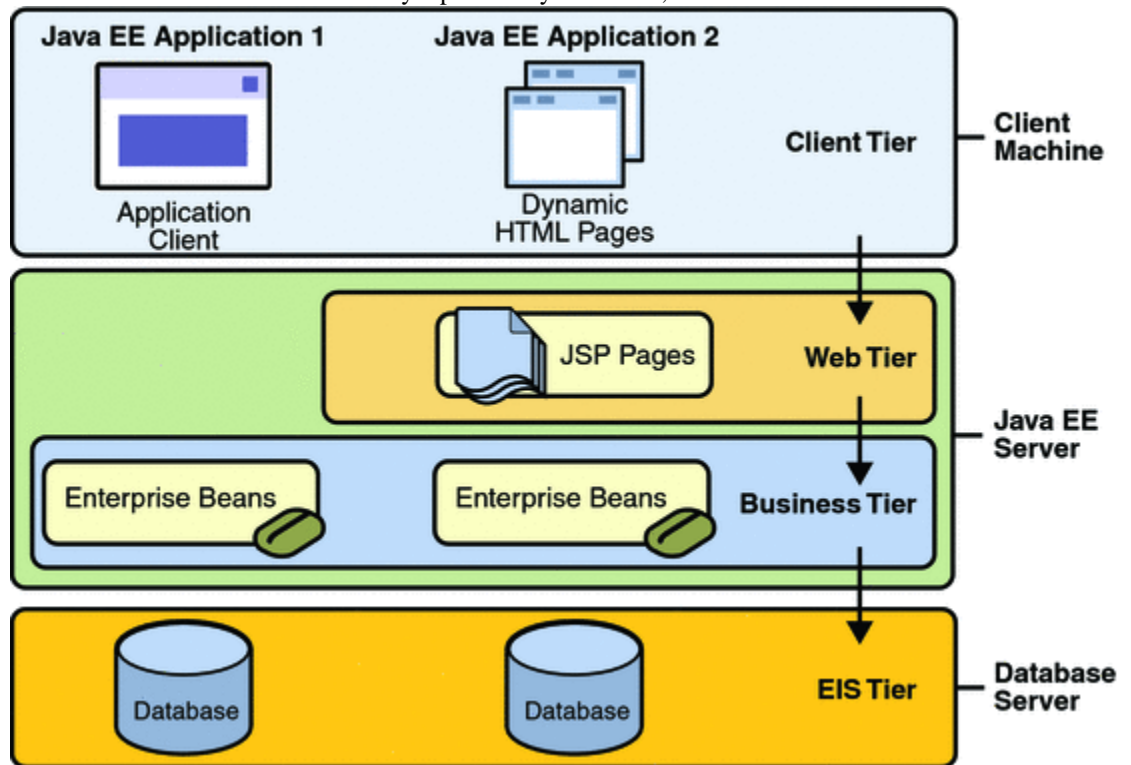
- Web tier, running on the Java EE server. It contains the Java Server Pages. This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier.

- Business tier, running on the Java EE server. It contains Java Beans, that contain the business logic of the application and Java Persistence Entities.

- Enterprise information system (EIS) running on the database server, consisting of data sources, to be more precise, databases and stores the data that needs to be retrieved and manipulated.

The server part is going to be run on a more powerful machine than a client – a high performance PC, according to RASD 1.4, part 3.6.2.1.

The illustration of the architectural style previously described, could be viewed below.

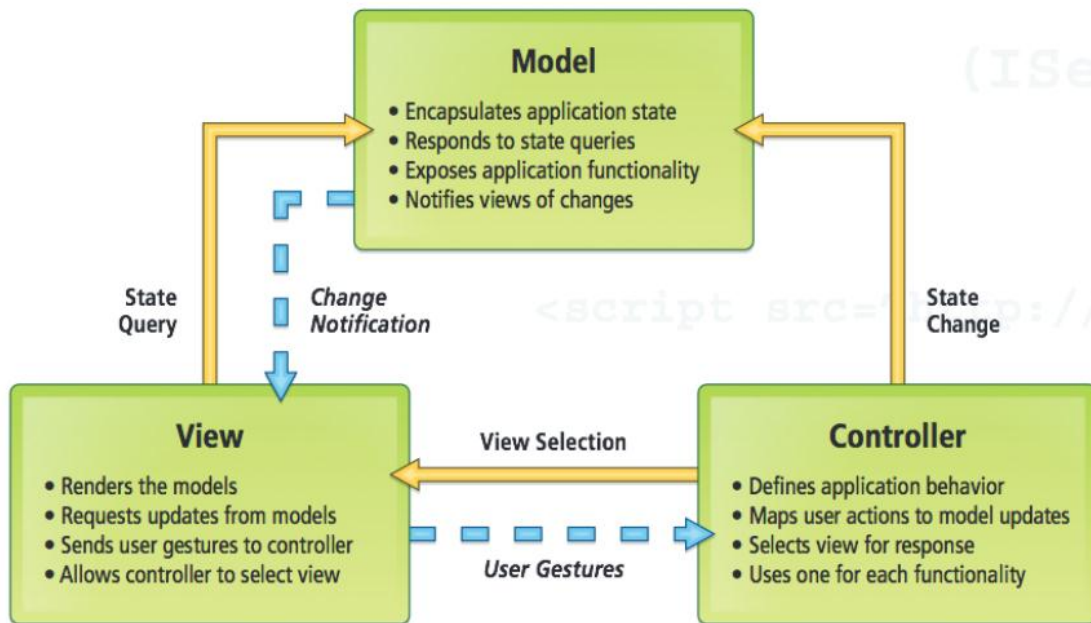


Picture: Java EE multitier

2.7.2 Model-View-Controller Pattern

MVC is a standard pattern that separates the user interface (View) and the business rules and data (Model) using a mediator (Controller) to connect model to the view.

The main benefit is the separation of concerns. Each part of the MVC takes care of its own work: the view takes care of the user interface, the model takes care of the data, and the controller sends messages between both of them.



Picture: Model-View-Controller pattern

In MyTaxiService case, this pattern is going to be implemented on client side of the application – either a web or mobile app.

All the forms and pages that are used to interact with users (Register form, Login form, Edit profile form, Request taxi form etc.) belong to View. As users take actions – click on buttons – these forms send user gestures to controller. Controller maps user actions to model updates. The Controller classes are going to be defined for these forms that are previously mentioned. User actions trigger the state change. Model encapsulates application state and responds to state queries. When the model gets data, it changes according to the data received, so the model will notify the view about the changes, so the view could render the model.

2.7.3 Session facade⁵

Enterprise beans encapsulate business logic and business data and expose their interfaces, and thus the complexity of the distributed services, to the client tier.

In a multitiered Java 2 Platform, Enterprise Edition (J2EE) application environment, the following problems arise:

- Tight coupling, which leads to direct dependence between clients and business objects;
- Too many method invocations between client and server, leading to network performance problems;
- Lack of a uniform client access strategy, exposing business objects to misuse.

A multitiered J2EE application has numerous server-side objects that are implemented as enterprise beans. In addition, some other arbitrary objects may provide services, data, or both. These objects are collectively referred to as business objects, since they encapsulate business data and business logic.

J2EE applications implement business objects that provide processing services as session beans. Coarse-grained business objects that represent an object view of persistent storage and are shared by multiple users are usually implemented as entity beans.

Application clients need access to business objects to fulfill their responsibilities and to meet user requirements. Clients can directly interact with these business objects because they expose their interfaces. When you expose business objects to the client, the client must understand and be responsible for the business data object relationships, and must be able to handle business process flow.

However, direct interaction between the client and the business objects leads to tight coupling between the two, and such tight coupling makes the client directly dependent on the implementation of the business objects. Direct dependence means that the client must represent and implement the complex interactions regarding business object lookups and creations, and must manage the relationships between the participating business objects as well as understand the responsibility of transaction demarcation.

As client requirements increase, the complexity of interaction between various business objects increases. The client grows larger and more complex to fulfill these requirements. The client becomes very susceptible to changes in the business object layer; in addition, the client is unnecessarily exposed to the underlying complexity of the system.

Tight coupling between objects also results when objects manage their relationship within themselves. Often, it is not clear where the relationship is managed. This leads to complex relationships between business objects and rigidity in the application. Such lack of flexibility makes the application less manageable when changes are required.

When accessing the enterprise beans, clients interact with remote objects. Network performance problems may result if the client directly interacts with all the participating business objects. When invoking enterprise beans, every client invocation is potentially a remote method call. Each access to the business object is relatively fine-grained. As the number of participants increases in a scenario, the number of such remote method calls increases. As the number of remote method calls increases, the chattiness between the client and the server-side business objects increases. This may result in network performance degradation for the application, because the high volume of remote method calls increases the amount of interaction across the network layer.

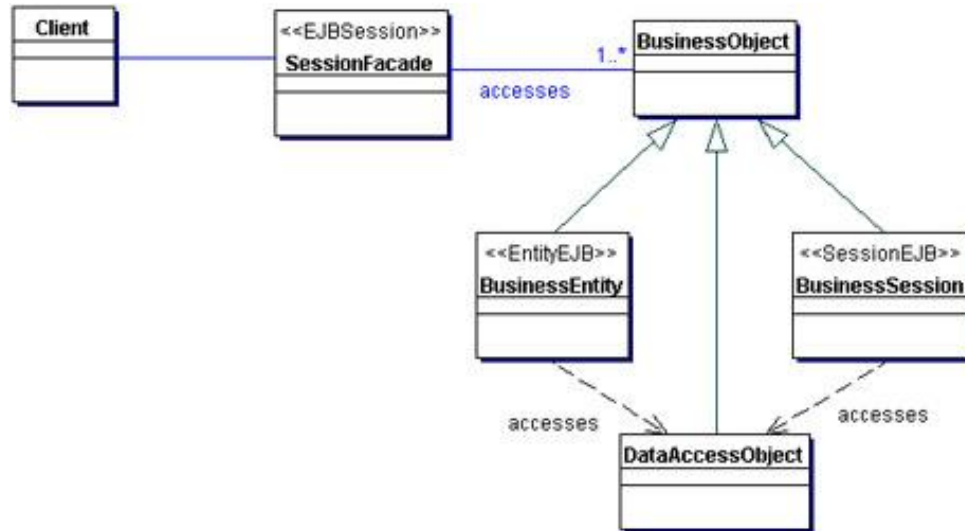
A problem also arises when a client interacts directly with the business objects. Since the business objects are directly exposed to the clients, there is no unified strategy for accessing the business objects. Without such a uniform client access strategy, the business objects are exposed to clients and may reduce consistent usage.

The solution to this problem is:

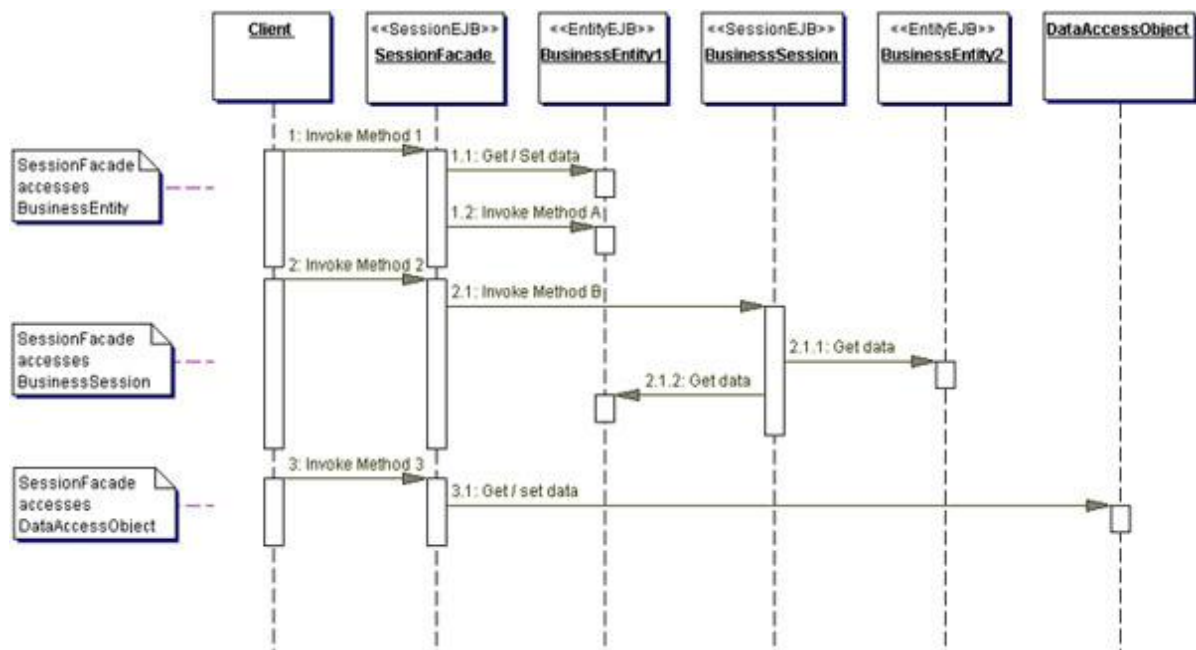
⁵ Based on <http://www.oracle.com/technetwork/java/sessionfacade-141285.html>

Use a session bean as a facade to encapsulate the complexity of interactions between the business objects participating in a workflow. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.

Class diagram representation Session Facade pattern:



Runtime component interaction:



Participants:

Client

This represents the client of the Session Facade, which needs access to the business service. This client can be another session bean (Session Facade) in the same business tier or a business delegate (see "Business Delegate" on page 248) in another tier.

SessionFacade

The SessionFacade is implemented as a session bean. The SessionFacade manages the relationships between numerous BusinessObjects and provides a higher level abstraction to the client. The SessionFacade offers coarse-grained access to the participating BusinessObject represented by the Invoke invocation to the session bean.

BusinessObject

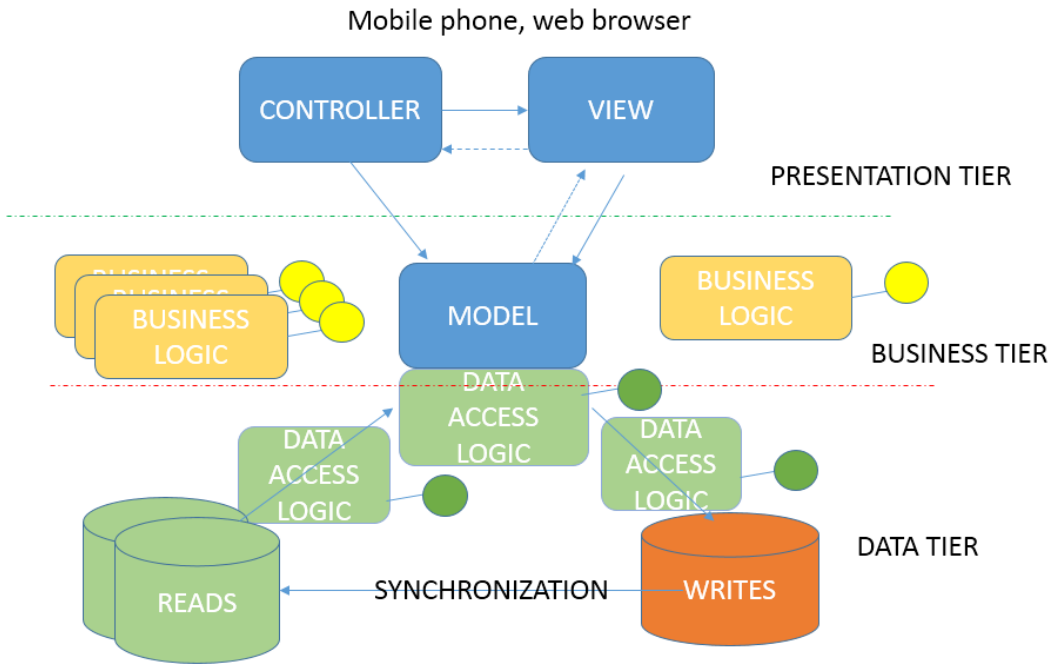
The BusinessObject is a role object that facilitates applying different strategies, such as session beans entity beans and a DAO (see the next section, "Strategies"). A BusinessObject provides data and/or some service in the class diagram. The SessionFacade interacts with multiple BusinessObject instances to provide the service.

Strategies

The Session Facade is a business-tier controller object that controls the interactions between the client and the participant business data and business service objects. In a complex application, there may be numerous Session Facades that can intermediate between the client and these objects. You can identify where a Session Facade might be useful by studying the client requirements and interactions typically documented in use cases and scenarios. This analysis enables you to identify a controller layer-composed of Session Facades-that can act as facades for these scenarios.

2.7.4 MyTaxiService architecture

Taking previous considerations into account, architecture of the whole MyTaxiService could be derived as a combination of n-tier client-server variant with MVC pattern implementation. Client and view are located on Client side as a part of Presentation tier, while Model is located on server side as a part of Business Logic tier. View would contain pages that contain input forms. Controller classes are classes that handle the user requests on these pages – send requests and responses, and according to that, update the user interface. In what follows, the derived architecture is illustrated:

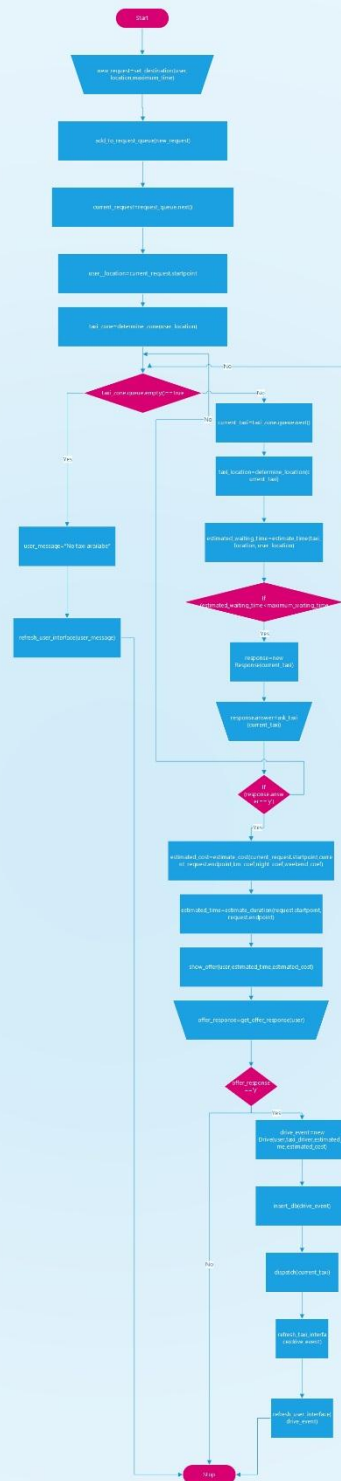


3 Algorithm design

This section explains the most important algorithms that are part of MyTaxiService application. There are three important algorithms that are unique to this application and they are going to be explained and illustrated, while others are not considered here, because most of them are trivial and well-known (register procedure, log in, etc.)

3.1 Scheduling algorithm

This is the most important algorithm of the application, and represents the core of MyTaxiService. It represents the main idea of the MyTaxiService – fair taxi management, based on taxi queues assigned to taxi zones. The algorithm itself is performed by the Scheduler component, responsible for taxi dispatch. In what follows, high-level algorithm is going to be displayed – in a form of a diagram with high-level pseudo-code (see ALGORITHMS folder for a higher quality JPEG picture and file algorithm_schedule.jpg or visio project file).

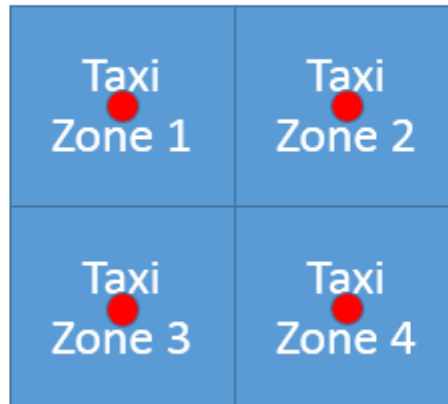


First, the user makes a request for a drive to the desired destination by selecting the desired location on a map and setting, optionally the maximum waiting time. It should be mentioned that maximum waiting time is used as the first criteria to forward request to next taxi. If taxi has to spend more time than user entered as maximum waiting time, in order to come to user's location, then the request is forwarded to next taxi driver. After that, user submits the request, and after that, the taxi zone corresponding to user's current location is determined (algorithm is going to be explained in 3.2). Scheduler checks the queue for the corresponding taxi zone if it is not empty. The Scheduler polls the first taxi from the queue and forwards the drive request by user. Taxi driver can either accept or reject. If driver rejects, the Scheduler continues to forward the request until it reaches the end of the queue for the corresponding taxi zone. If the driver accepts the request, then the user is notified about the drive offer that displays the estimated time and price (3.3). If the user accepts the offer, then the taxi is dispatched, and drive event is created. Otherwise, if user rejects the drive offer, algorithm will finish and user will be returned to previous screen.

3.2 Taxi zone determination

As it is previously assumed in RASD document, the determination of the corresponding taxi zone is done by system. According to the definition of taxi zone in terms of this problem, each taxi zone is defined by its center point. It is also assumed that the division of the city region on taxi zones itself is done by the government manually (center points). But, it should be mentioned that it is required to have taxi zones that are approximately 2km^2 each. So, it would be necessary to find a model for this problem.

Let's consider each taxi zone as a square of approximately 2km^2 with center point defined in advance.



So knowing that:

$$P = a * a = 2 \text{ [km}^2\text{]}$$

where a is length of the side, it could be concluded that maximum distance from the center in a particular taxi zone could be equal to half of the diagonal (outer circle radius), while we have:

$$d = a\sqrt{2}$$

$$a = \sqrt{2} \text{ [km]}$$

$$d = a\sqrt{2} = 2 \text{ [km]}$$

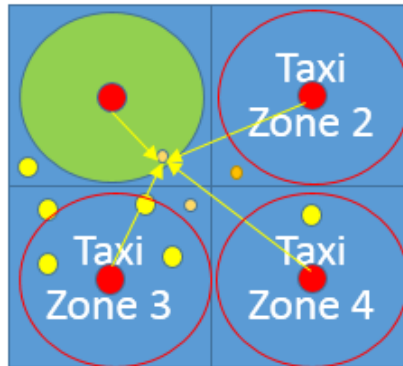
$$\text{maxdistance} = \frac{d}{2} = \frac{\sqrt{2} * \sqrt{2}}{2} = \frac{2}{2} = 1 \text{ [km]}$$

Now, let's consider a possible situations.

1. We want to know what is the situation when we are completely sure that some point belongs to a certain taxi zone. In case that location is inside the inner circle of the square ($r=a$, so $r/2=\sqrt{2}/2$), we can be sure that it belongs to a particular taxi zone.

$$r = a = \sqrt{2} \text{ [km]}$$

If($\text{distance}(\text{current}, \text{center}_i) \leq r/2$)



Current- user's currentlocation

Small orange circles – users

Yellow circles – taxi cars

Red circles – taxi zone center

Green circle – inner circle related to the corresponding taxi zone

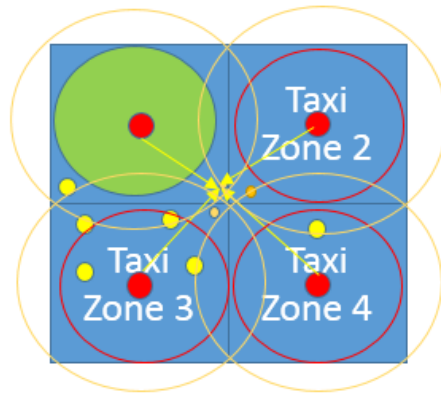
r – radius of the inner circle

$center_i$ – center corresponding to the i th taxi zone

2. Now, let's consider a situation when the tested point doesn't belong to the inner circle. In this case, the outer circle is considered

$$R/2 = \frac{d}{2} = \frac{\sqrt{2} * \sqrt{2}}{2} = \frac{2}{2} = 1 [km]$$

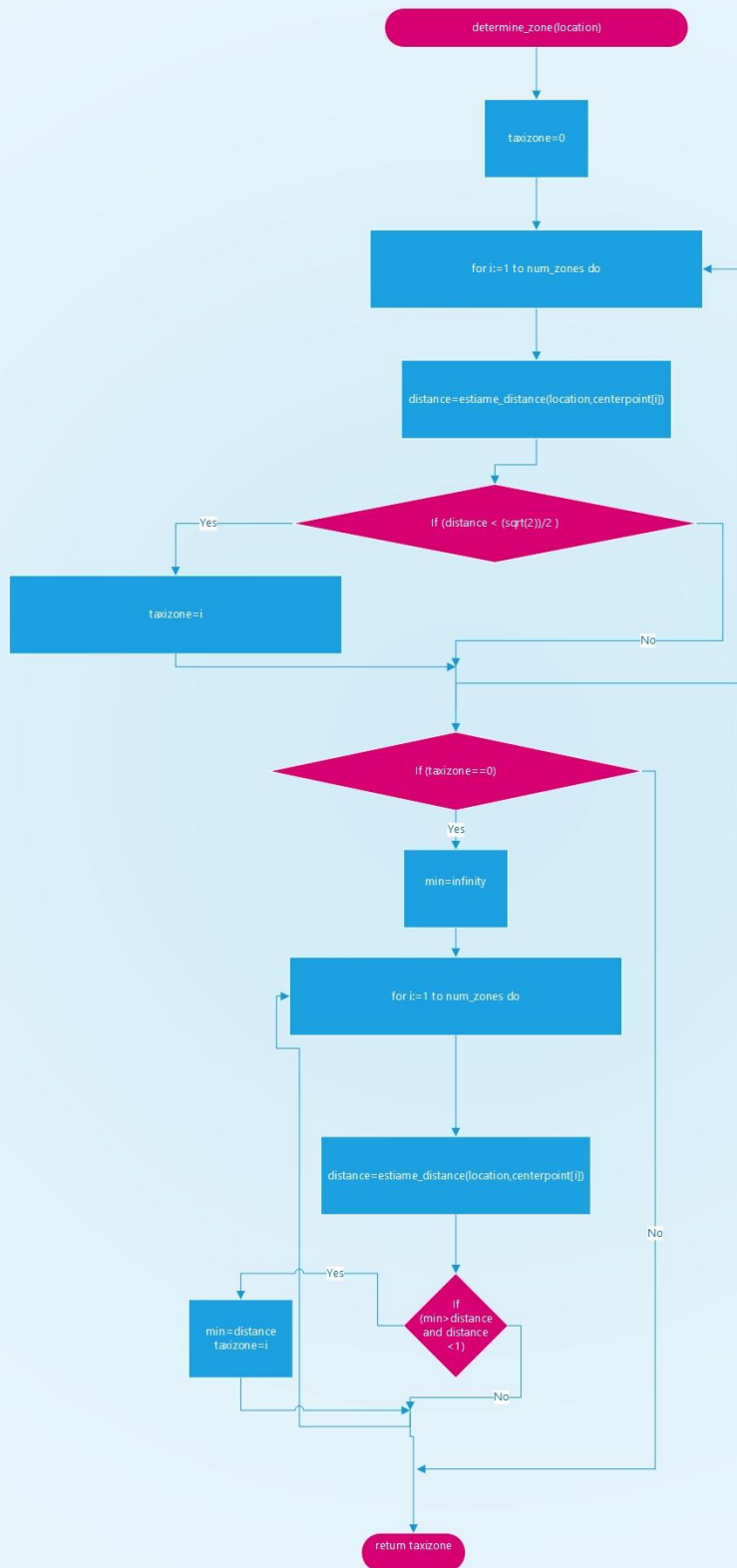
$\text{Min}(\text{distance1}, \text{distance2}, \text{distance3}, \text{distance4})$



After that, to find the corresponding taxi zone, it is necessary to find the taxi center of a taxi zone which is the closest to the current location, because, as it can be seen, the current location can belong to intersection of taxi zones.

3. If the location is out of city boundaries, the taxi zone will stay 0 (see the algorithm), and it would be a sign that user has selected invalid location.

In what follows, the high-level algorithm interpretation is given in form of a diagram and high-level C++-like pseudo-code. If you are having difficulties reading the algorithm, please see high-quality JPEG picture. [ALGORITHMS/taxizone.jpg](#) or corresponding visio project file.



3.3 Cost estimation algorithm

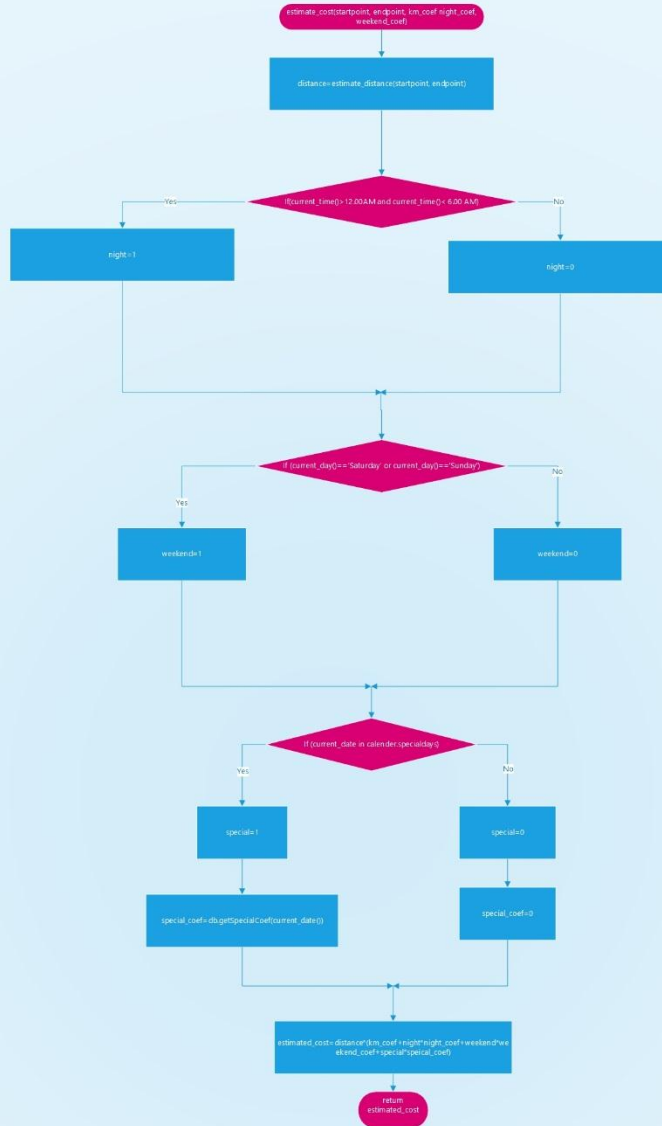
As a part of the drive negotiation protocol, cost estimation plays important role, as it gives overview to users what would be the price for a taxi service to the desired destination, so the user can either accept or reject the drive offer.

As the pricing policy could be modified and changed, the algorithm is going to be flexible. Several different factors are taken into account:

1. Distance – the government determines what would be the price per kilometer - rate
2. Time of the day – from 12.00 am to 6.00 am, price could be increased
3. Weekend price – on weekend, price could be lower or higher, depending on government's decision
4. Special days – it depends on current date. Calendar is checked. If it is a special day or holiday (Christmas, New Year, Thanksgiving etc.) a special price increase or decrease could be defined, so the database is checked for the coefficient of increase (positive) or decrease (negative coefficient).

Each of these coefficients could be changed according to the specific city government's decisions, so this coefficients and rates are function parameters that could be easily changed.

In what follows, the high-level algorithm interpretation is given in form of a diagram and high-level C++-alike pseudo-code. If you are having difficulties reading the algorithm, please see high-quality JPEG picture. `ALGORITHMS/cost_estimate.jpg` or corresponding visio project file.



4 User interface design

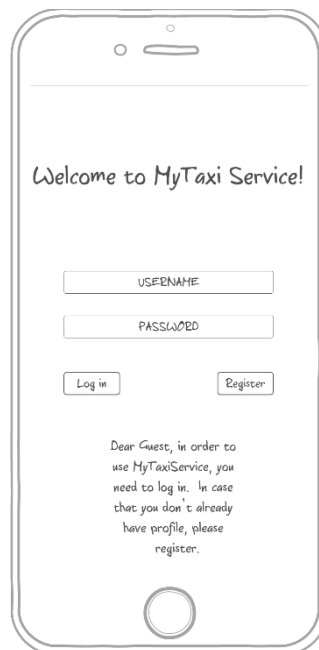
This part of the document references to RASD document, section 3.1.1.

3.1.1 User Interfaces

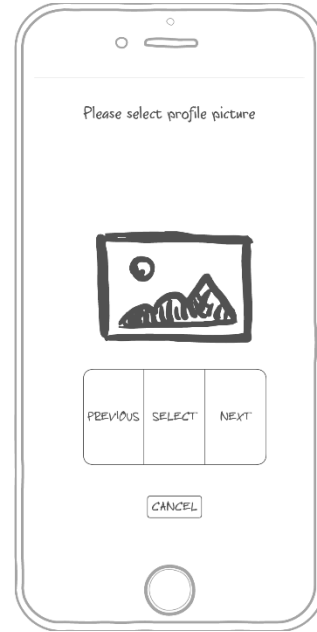
In what follows, mockups that represent the idea and conceptual design of the application are presented – mobile and web application. The mockups presented may differ from the final product and they are placed in this document for mainly illustrative purposes – to give you better idea how is the final product going to look. First, the user's point of view is presented, and then, taxi driver's and administrator's, in this order.

User's point of view

3.1.1.1 Login - The mockup bellow shows the home page of MyTaxiService. Here users can log in to the application and guests can access to the registration form in order to use the service. Here, the mobile application mockup is presented (the web version will look as similar as possible)



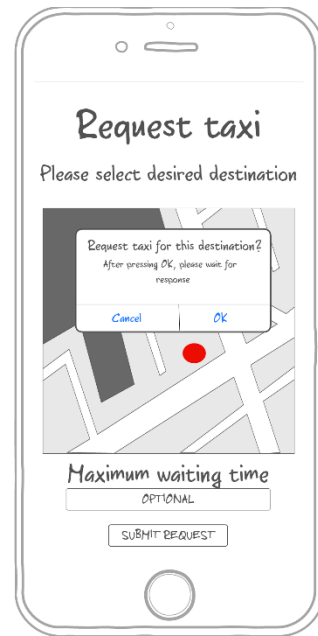
3.1.1.2 Registration form - This mock shows the registration form page on a mobile device. This is form that consists of user-filled textboxes. Profile picture is optional and helps in situations when user/driver needs to recognize another person that is involved into drive process. After pressing the button called „Confirm registration“, the guest becomes registered user and can benefit from usage of this system. Profile picture is optional, and user can select a picture from device's storage.



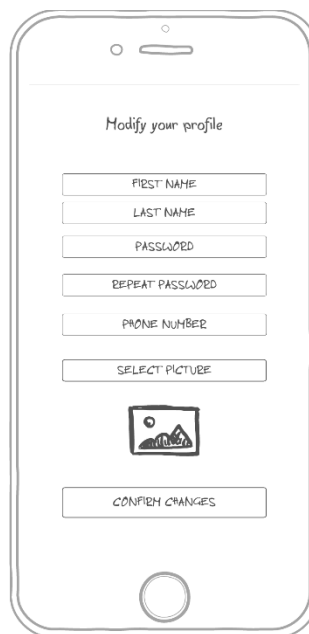
3.1.1.3 User menu – This mockup shows what user can do once logged in – request a taxi, edit profile or log out.



3.1.1.4 Request a taxi - This mock up shows what happens when user wants to send a request for a taxi drive. User selects desired location (left picture), optionally enters maximum waiting time and submits the request (right picture).



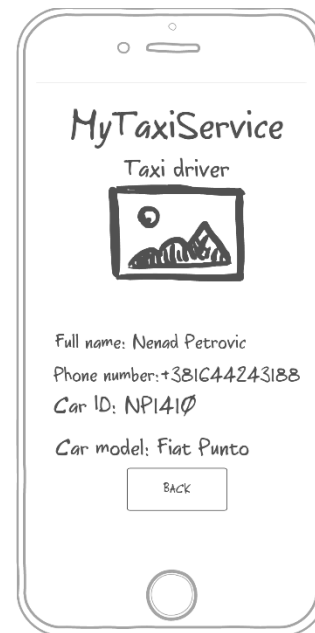
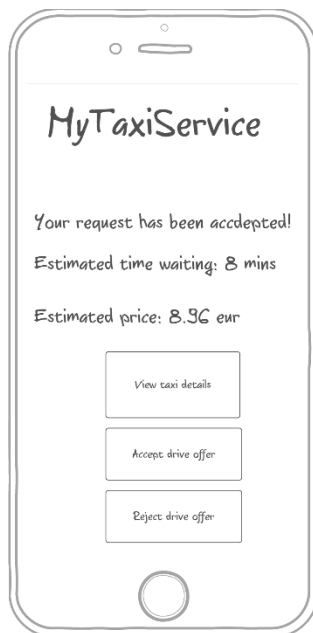
3.1.1.5 Profile modification - This mock up shows the form where user can edit his/her that or change picture, but can't change fiscal code.



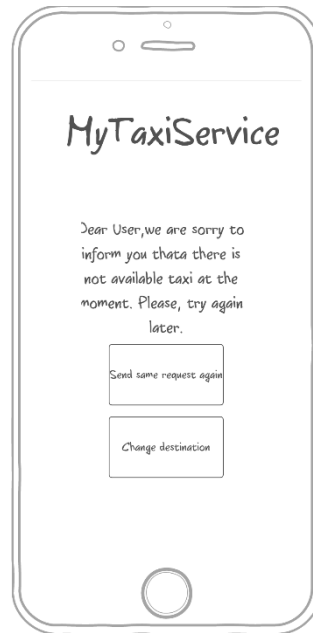
3.1.1.6 Waiting for response screen -This mock up shows what happens when the user submits the request for a drive. System responds within 180 seconds, but user can press quit waiting before that and cancel request and return to menu.



3.1.1.7 System response - This mock up shows what user sees when request is accepted by taxi. User can view taxi details (displayed on the right picture), accept or reject the offer. Estimated waiting time and estimated price are shown, so they can help user decide whether to accept the offer or not.



3.1.1.8 No taxi available – This mockup shows the screen displayed to user when there is no taxi available.

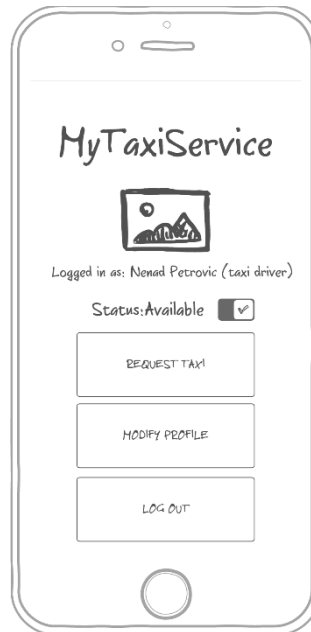


3.1.1.9 Drive progress and report form – This mockup shows the screen displayed to user after accepting the drive offer. User can see the drive progress, estimated time remaining, but can also report driver or send an S.O.S signal. After clicking the Report driver button, the screen on the right side shows up. User has to enter the reason of reporting the driver and confirm report in order to make it valid.

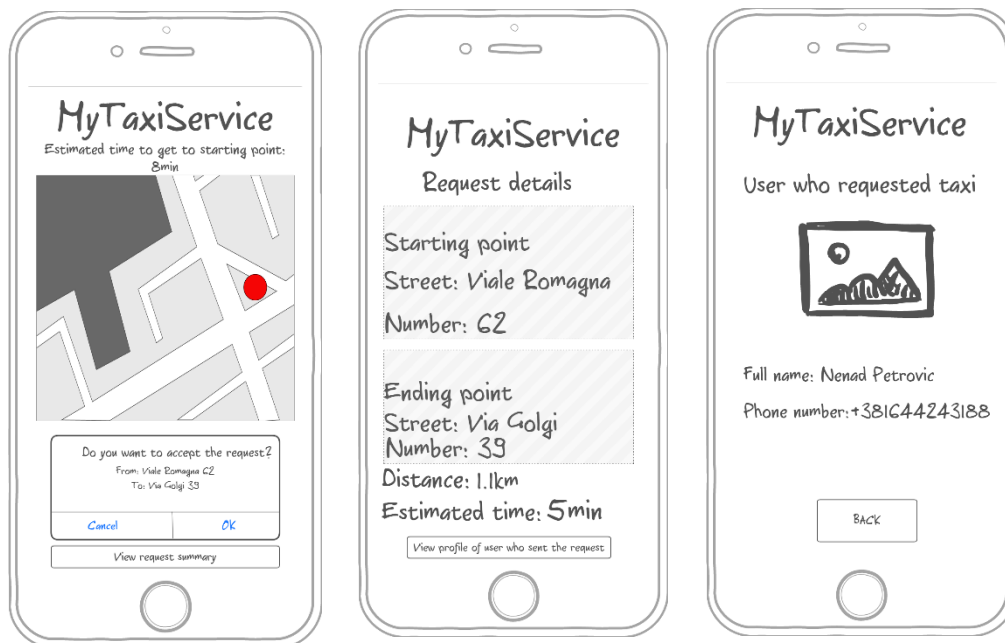


Driver's point of view

3.1.1.10 Driver menu – this mockup is similar to user's menu, but includes availability status button.



3.1.1.11 User request accept/reject page – Driver can accept or reject user's request for a taxi drive. It is possible to optionally view request summary, as shown on the middle picture. After that, it is possible to view user profile of the person who sent the request (right side).



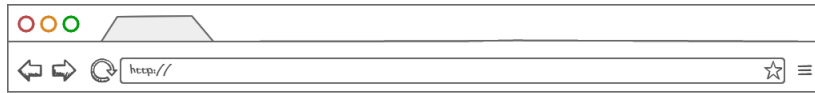
3.1.1.12 Waiting user response – Within 60 seconds, user who requested drive must accept or reject the drive offer and taxi driver has to wait (the left side). After accepting the offer, the similar screen is displayed to a taxi driver like in 3.1.1.9 (the right side). Driver can also report user the same way as in 3.1.1.9.



Administrator's point of view

Administrator's application is optimized for web browser, but will also have a mobile counterpart. In what follows, the web browser version is going to be presented.

3.1.1.13 Login page – In case of using the web browser version, user logs in by entering data in form like shown bellow.

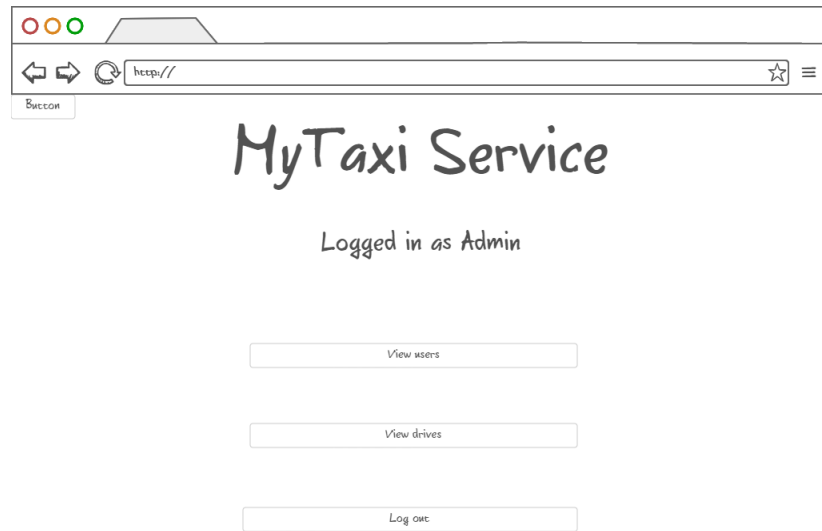


Welcome to MyTaxi Service!

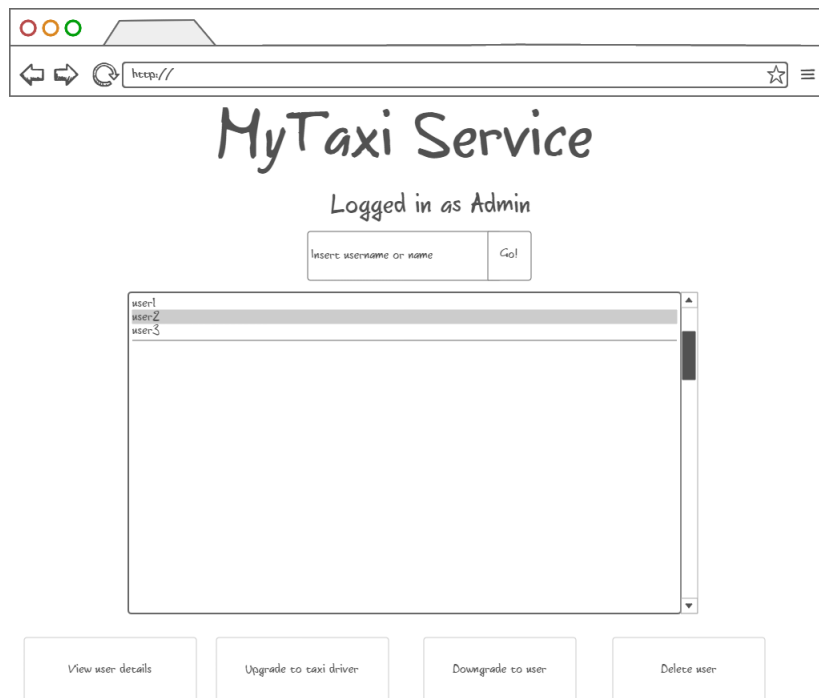
Dear guest, please log in or register in order to use our service.

Email	<input type="text"/>
Password	<input type="password"/>

3.1.1.14 Administrator menu – Administrator can view users (both typical users and taxi drivers) and their reports in order to decide their removal from system or view drives. Of course, there is also a „Log out“ button.



3.1.1.15 Browsing users – Administrator can view users and their reports in order to decide if they deserved removal from system, promote users to taxi drivers, downgrade taxi drivers to users and edit user's data.



3.1.1.16 Promotion to taxi driver – After user selection, administrator can promote user to taxi driver by entering valid driving license number, car identification number and car model (optional). Data is checked by the system. If data is valid, promotion can be confirmed.

MyTaxi Service

Promote user: Nenad Petrovic to taxi driver

Driving license number:

Car id:

Car model:

MyTaxi Service

Promote user: Nenad Petrovic to taxi driver

Driving license number: 010935634 ✓

Car id: NP1410 ✓

Car model: Fiat Punto

Are you sure?
Are you sure you want to promote this user?

3.1.1.17 Viewing user info, reports and drive event – After user selection, administrator can view user details and all reports and drive events related to a particular user (left picture). It also possible that administrator wants to delete the user after reading the reports. The reports of selected user are shown in a form of table as on the right-side picture. Similar form appears when system administrator is notified about new report which needs an action.

MyTaxi Service

Full name: Nenad Petrovic
Phone: +381644243188

drive	report
drive2	report2

MyTaxi Service

Reports for: Nenad Petrovic

Report number	From user	Reason	Startpoint	Endpoint	Timestamp

Drive events related to selected user could also be viewed in a form of a short summary with option of viewing details of actors involved in the selected drive event.

A screenshot of a web browser window displaying the 'MyTaxi Service' user summary page. The browser's address bar shows 'http://'. The page title is 'MyTaxi Service'. The user information is as follows:

- User: Alice Jackson (with a 'View user' button)
- Taxi driver: Bob Eastwood (with a 'View user' button)
- Startpoint: Viale Romagna, 62
- Endpoint: Via Golgi, 39
- Date: 02-11-2015

At the bottom of the page, there are two buttons: 'View reports' and 'Go back'.

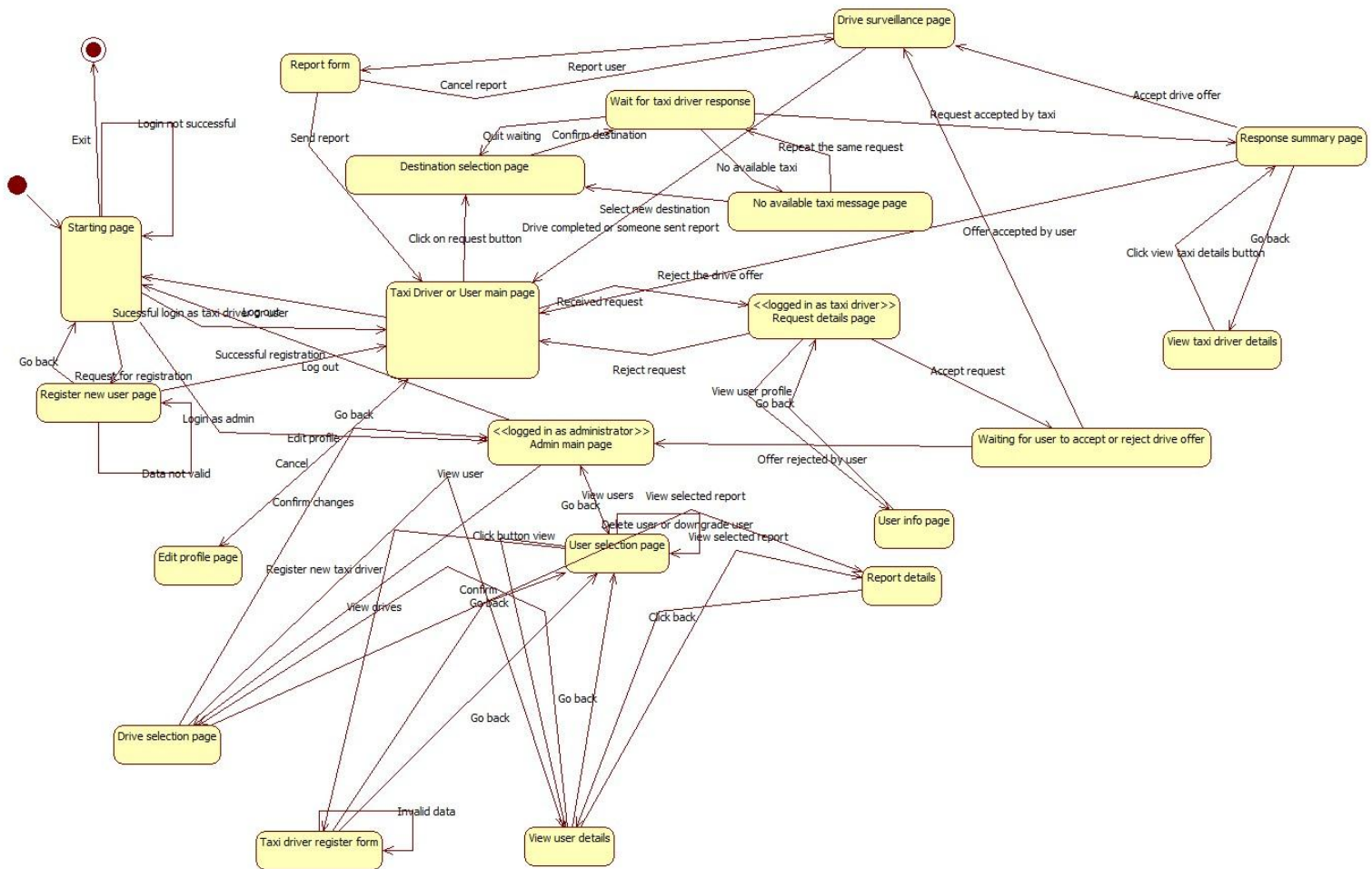
3.1.1.17 Deleting selected user – After user selection, administrator can also delete the selected user if he/she deserved. The deleted users can't login nor register anymore. Their fiscal code is blacklisted.

A screenshot of a web browser window displaying the 'MyTaxi Service' user management page. The browser's address bar shows 'http://'. The page title is 'MyTaxi Service'. The user is logged in as 'Admin'. Below the title, there is a search bar with the placeholder text 'Insert username or name' and a 'Go!' button. A list of users is displayed in a table:

user1
user2
user3

A confirmation dialog box is open in the center of the page, asking 'Are you sure?' and 'Delete user2?'. The dialog has two buttons: 'Yes' and 'No'. Below the dialog, there are four buttons: 'View user details', 'Upgrade to taxi driver', 'Downgrade to user', and 'Delete user'.

The following state machine diagram gives a simplified vision of the entire application.



Together with mockups from 3.1.1, this diagram gives a complete overview of the user interface design.

3.1.2 API – Application Programming Interface

Software interface to other applications will be in a form of Application Programming Interface.

MyTaxiService will offer API that targets developers as possible users. Developers will have programmatic interface that will allow them to call MyTaxiService functions inside their application, so they can easily integrate with their apps and extend the usage of MyTaxiService.

API would contain objects and functions that encapsulate MyTaxiService functionalities.

Developers will be able to connect their social network-alike applications and easily migrate data to register using already created profiles that have necessary information for registration. For example, in notation of Java-alike object – oriented pseudo-language, it would look like:

```
MyTaxiServiceObject.register(username:      String,      password:String,  
fiscalCode: String, phoneNumber: String, gender: Char, picturePath: String)
```

Developers would also have ability to request a taxi inside their application if they have a valid session (previously logged in):

```
MyTaxiSession      session1=MyTaxiServiceObject.login(username:      String,  
password:String)  
  
Session1.RequestTaxi( )
```

Functions like this would give developers ability to make more complex systems using MyTaxiService functions as primitives.

5 Requirements traceability

In this part, it is going to be considered how the requirements previously defined in RASD map to design decisions, in other words, how the requirements affect the software design.

Functional requirements

Registration/Taxi driver promotion data check – Considering the requirement that system must find out whether the provided fiscal code/car identification/driving license number is valid or not, and belongs to corresponding person, MyTaxiService needs to be able to communicate with some external database that could provide the necessary information. So, MyTaxiService communicates via REST service with external city government database which would check data validity related to citizens.

GPS coordinates determination and calculation –As it is necessary to use calculations based on GPS coordinates and current location, it is also needed to use external API here (outside of the application). When it comes to taxi zone determination – it is necessary to calculate distances between two locations, for example. When user sends a request, his/her current location is taken as a starting point. As it can be seen, it is necessary to have full API that deals with GPS coordinates and calculations between locations. As this is not implemented as a part of this application, external API is used, such as Google Maps API.

Reports – One of the requirements is to give possibility that users can report each other in case of bad behaviour. Administrator can later see the reports, read their reasons and decide to ban users or not. So, reports must be stored in database and there is REPORT table in database related to this system.

Blacklist – after ban, users are not able to log in, nor to register again using same fiscal code. So, it is necessary to keep track of blacklisted users. There are two ways, as it is going to be explained in database design part. They could be stored in a separate table (their fiscal code) or we can add additional attribute to each user which is set to 1 after ban („Blacklisted“ attribute).

Non-functional requirements

Maintainability – as it is necessary to estimate the cost of a drive from one place to another, it is necessary to use some kind of pricing policy. It is also needed to make algorithm as flexible as possible, because this policy could be related to only one city, and it could be changed during time. So, the algorithm implementation takes many factors into account – distance, time of the day, is it weekend or not, and is it a holiday or special day. These considerations are typical and common to any pricing policy. The algorithm has many parameters that could be easily changed, so it could be shaped according to pricing policy. Putting coefficient 0 means that the parameter is not taken into account, or putting negative parameter, it means that it decreases price (discounts). This is one of the non-functional requirements - maintainability requirement. It was needed to make easily changeable and flexible algorithm that could be customized according to pricing policy and changed in future. When this software is customized for another city, only algorithm parameters are going to be changed, but not the algorithm itself.

Security – as it has been told in RASD document, server side will use separated web, application and database part, with firewall between each of them to prevent unauthorized users from access. Taking this consideration into account, it is found obvious that Java EE is solution that would be suitable for this kind of separation, as it offers the separate client, web, business and database tier, which is exactly what is needed. So, this will give ability to place firewalls between each two parts and make application secure and meet the security requirements.

Considering the other non-functional requirements, it could be said that decision to use Java EE reflects many of the non-functional requirements previously defined in RASD, because Java EE is, generally based on n-tier architectural style which brings many benefits when it comes to non-functional requirements.

Java Enterprise Edition is designed to develop large-scale and multi-tiered applications that are scalable and meet reliability conditions and are secure at the same time.

The main benefits of the N-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.

So, the decision to use N-tier and Java EE implementation satisfies most of the non-functional requirements and is related to them.

Interface

In the end, a few words about developer API and how it affected the design.

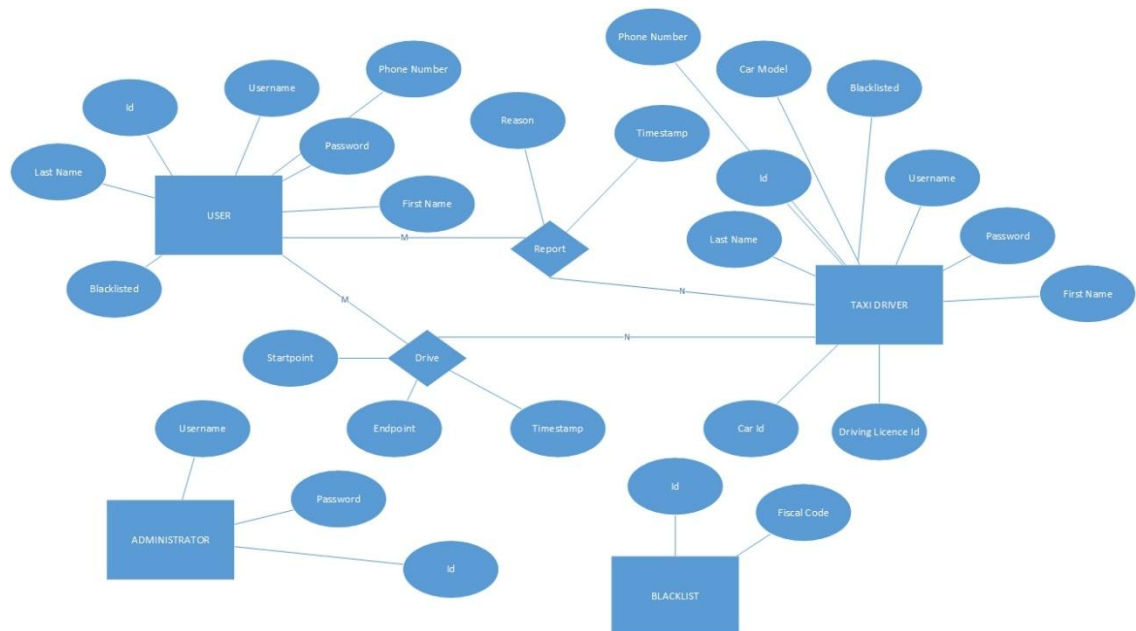
API for developers gives them ability to use any of the MyTaxiService functions in order to make their own application based on MyTaxiService operations as a primitives. So, it is decided to make full access to developers to MyTaxiService via REST service, but developers must be registered in city government office in order to get such privileges.

6 Database design

This system uses simple database that has several purposes:

- to store registered users and their information, which would be necessary in order to give them ability to log in and use the service
- to store drive events summary, which would give overview to the administrator
- to store report, which would help administrators to bring decisions about deleting users that show bad behaviour
- to keep track of users that are banned

In what follows, ER diagram of this simple database is presented:



We can see that it stores necessary information for user, taxi driver and administrator which would allow them to log in – username and password.

Taxi driver is different from user only in several attributes – it has all the attributes the users have, but also car model, car id and driving license id.

Reports are stored in database, so they can be later read by administrators. Report reason is necessary in bringing decision to ban user or not, so it is stored as an attribute in this database.

Blacklist could be a separate table, where user fiscal codes are stored after their ban by admin. There is also alternative way – to store attribute „Blacklisted“ and set it to 1 after ban, and not delete the banned users from database.

Let's take a look at a translation of this ER model into database:

User(Id,Username,Password,FirstName,LastName,PhoneNumber,Blacklisted,isTaxiDriver,CarModel,CarId,LicenseId)
Administrator(Id,Username,Password)

As it can be seen, taxi driver is not translated as a separate table. Attribute “isTaxiDriver” is added, instead. When this attribute is set to 1, attributes CarModel,CarId and LicenseId are considered valid. When Blacklist attribute is set to 1, user can't log in or register again.

Report(Id,*Sender*,*Receiver*,Reason,Timestamp)

As it can be seen on ER diagram, we need M-N relationship, so Report is translated as a separate table with foreign keys corresponding to the sender of the report (the one who reports another user) and receiver of the report (user who is showing bad behaviour) . In fact, their Ids are used as a foreign keys.

Similarly, Drive is translated:

Drive(Id,*User*,*Driver*,Startpoint,Endpoint,Timestamp)

Startpoint is the starting location, while endpoint is the desired destination. User is foreign key which corresponds to the user, while Driver is foreign key which corresponds to the Taxi Driver participating in this drive event.

7 Software and tools used

- Microsoft Office Word 2013: To create and redact this document
- StarUML: to create component view, runtime view, deployment view, sequence diagrams in runtime view.
- Edraw Max 7.9: to draw component interfaces
- Microsoft Visio 2016: to draw high-level architecture, algorithms and database design.

- NinjaMock (<https://ninjamock.com/>): to create mockups for both mobile and web variants of the application.
- Github (repository: <https://github.com/penenadpi/Software-Engineering-2-Project/Deliveries>)

7.1 Hours of works

The time spent for constructing this document and Alloy model:

- Nenad Petrovic: ~ 50 hours.

8 References

1. Software Design and Software Architecture (Software Engineering 2 course, Politecnico di Milano)-6. Design I –II.pdf
2. Java EE7 introduction (Software Engineering 2 course, Politecnico di Milano)- Java EE.pdf
3. Java API for restful services, https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services
4. Java Persistence API, https://en.wikipedia.org/wiki/Java_Persistence_API
5. Session Facade, <http://www.oracle.com/technetwork/java/sessionfacade-141285.html>