

Politecnico di Milano

2015-2016

Software Engineering 2: “MyTaxiService”

Design Document

version 1.0

Author: Nenad Petrovic

22nd November 2015

Contents

[1 Introduction 3](#_Toc435104491)

[1.1 Purpose 3](#_Toc435104492)

[1.2 Actual System 3](#_Toc435104493)

[1.3 Scope 3](#_Toc435104494)

[1.4 Actors 4](#_Toc435104495)

[1.5 Goals 5](#_Toc435104496)

[1.6 Definitions, Acronyms, Abbreviations 5](#_Toc435104497)

[1.7 Reference Documents 7](#_Toc435104498)

[1.8 Document overview 7](#_Toc435104499)

[2 Overall Description 8](#_Toc435104500)

[2.1 Product perspective 8](#_Toc435104501)

[2.2 User and stakeholder characteristics 8](#_Toc435104502)

[2.3 Constraints 9](#_Toc435104503)

[2.4 Assumptions and Dependencies 9](#_Toc435104504)

[2.5 Future possible implementation 11](#_Toc435104505)

[3 Specific Requirements 12](#_Toc435104506)

[3.1 External Interface Requirements 12](#_Toc435104507)

[3.2 Functional Requirements 24](#_Toc435104508)

[3.3 The world and the machine 28](#_Toc435104509)

[3.4 Scenarios 30](#_Toc435104510)

[3.5 UML Diagrams 32](#_Toc435104511)

[3.6 Non Functional Requirements 52](#_Toc435104512)

[4 Appendix 54](#_Toc435104513)

[4.1 Alloy 54](#_Toc435104514)

[4.2 Software and tools used 66](#_Toc435104515)

[4.3 Hours of works 66](#_Toc435104516)

[5 Revision 66](#_Toc435104517)

# Introduction

## 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.

## Scope and document overview

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

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 themself 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.

Data design: This part of the document will present the ideas related to data design of the data used in this system, more concrete – the database design is going to be defined and described by database schema, supported by graphical diagrams and interpretations (ER diagram).

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.

## 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 constains 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: aggresion, avoiding payment, offensive beahviour etc.

1.3.2 Acronyms

* API: Application Programming Interface.
* DD: Design Document.
* DBMS: DataBase management system.
* DB: DataBase.
* GUI: Graphical User Interface
* JEE: Java Enterprise Solution
* JVM: Java Virtual Machine
* OS: Operating System.

## Reference documents

This document referecnces 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]

# Architectural design

## 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 accpeting/rejecting drive requests.

-Scheduler subsystem – determines the taxi zones and forwards requests and responses to corresponfing 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.

-Developer subsystem – operations related to external API offered to developers. They can virtuallz 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 to deal with these requests, create responses according to the enviornment 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.

## High level components and their interaction

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

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.

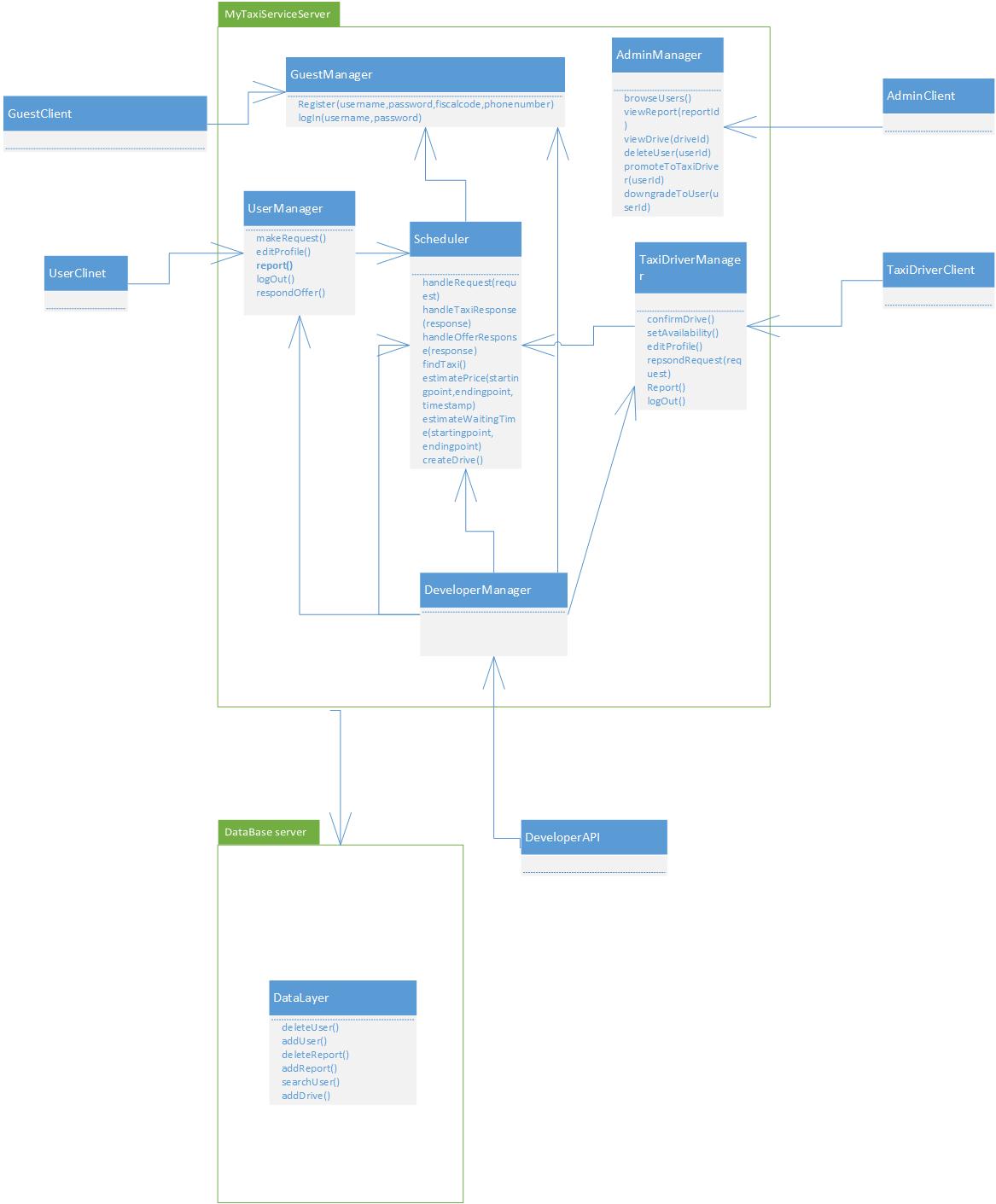
Developers have programmatic interface that lets them call any function from MyTaxiService, except the functions related to administration. Component that handles the API calls is DeveloperManager and belongs to server part of the application.

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.

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. Parameters are going to be explained later in component interfaces part.



## Component view

The system follows client-server style for what concerns the management of the interaction with the users (including system administrators, taxi driversn 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 tfour kinds of clients – RegisteredUserClient, TaxiDriverClient, AdminClient and UnregisteredClient. Moreover, the part of the server that is related to data that needs to be check 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 folowing parts:

-UnregisteredManager is a component which gives ability 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 users location when necessary. In order to acomplish 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.

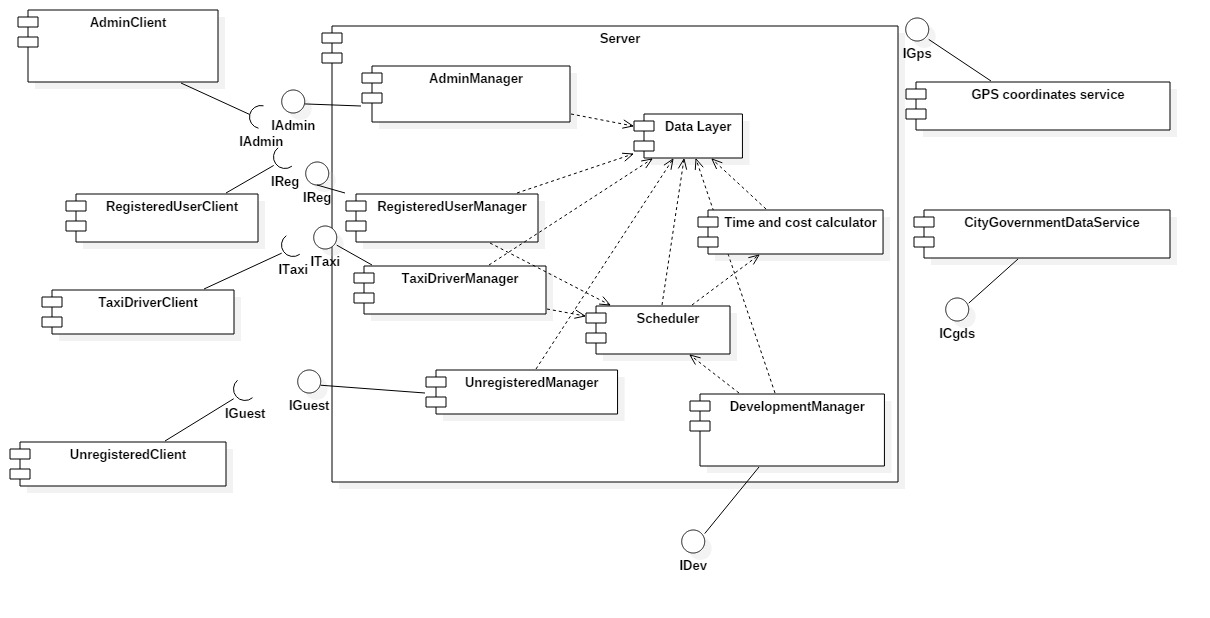
-TaxiDriverManager is similar to previous one, but also includes some additional options, like accepting or rejecting a drive request and changing the availability status, in order to support TaxiDriver clients.

-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 uses 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 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 rejecet the offer if the price is too high.

-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 serivce (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.

## Deployment view

## Runtime view

## Component interface

## 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 acrhitectural style.

In what follows, the advantages of this architecure and reasons for the selection aregoing 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:

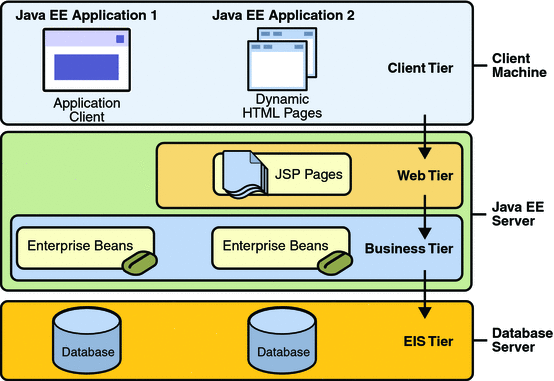
-Client tier that is running on the client machine. It contains Application Clients and Web Browsers and it is the layer that interacts directly with the actors. The client machine could be either mobile phone running application or web browser or personal computer running web browser in this case. Taxi 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 Paged. 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.

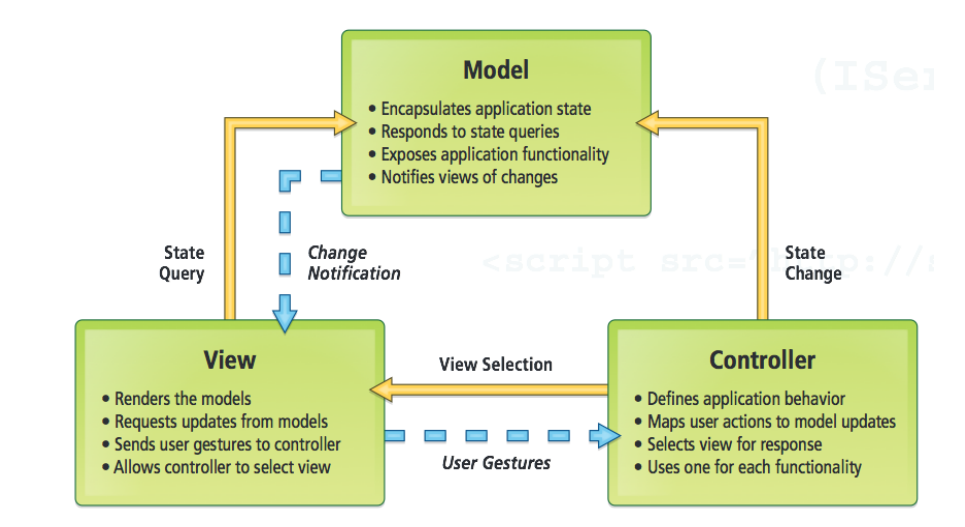
-Enterprise information system (EIS) running on the database server, consisting of data sources, to be more precise, databases and stores the data that needs to be retrieved and manipulated.

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.

2.7.2 Model-View-Controller Pattern

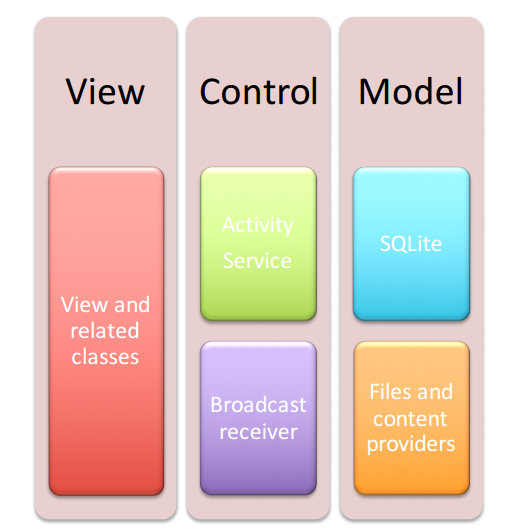
MVC is a standard pattern that separates the user interface (View) and the business rules and data (Model) using a mediator (Controller) to connect model to the view.

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.

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. In this case, model will deal with sending requests to a server and getting the data from the server. 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.

When it comes to Android application, which is going to available for mobile phones, MVC implementation in Android is illustrated below:



The view consists of classes related to user interface, such as buttons, text fields etc. When user takes some action, controller takes its role. Controller requests model state change, and in this case, model classes communicate with server side.

# Algorithm desgin

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.)

## 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).

## D:\swe2\Software-Engineering-2-Project\Deliveries\DD\ALGORITHMS\algorithm_schedule.jpg

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.

## 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 divsion 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 approxiamtely 2km2 each. So, it would be necessary to find a model for this problem.

Let’s consider each taxi zone as a square of approximately 2km2 with center point defined in advance.

So knowing that:

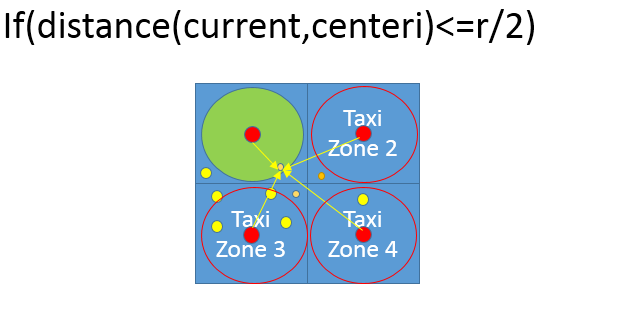
=2 [km2]

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 (outter circle radius) ,while we have:

[km]

=2 [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.

Current- user’s currentlocation

Small ornage circles – users

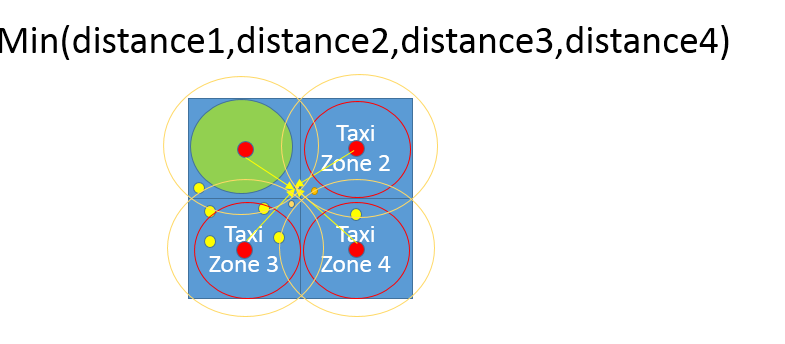
Yellow circles – taxi cars

Red circles – taxi zone center

Green circle – inner circle related to the corresponding taxi zone

r – radius of the inner circle

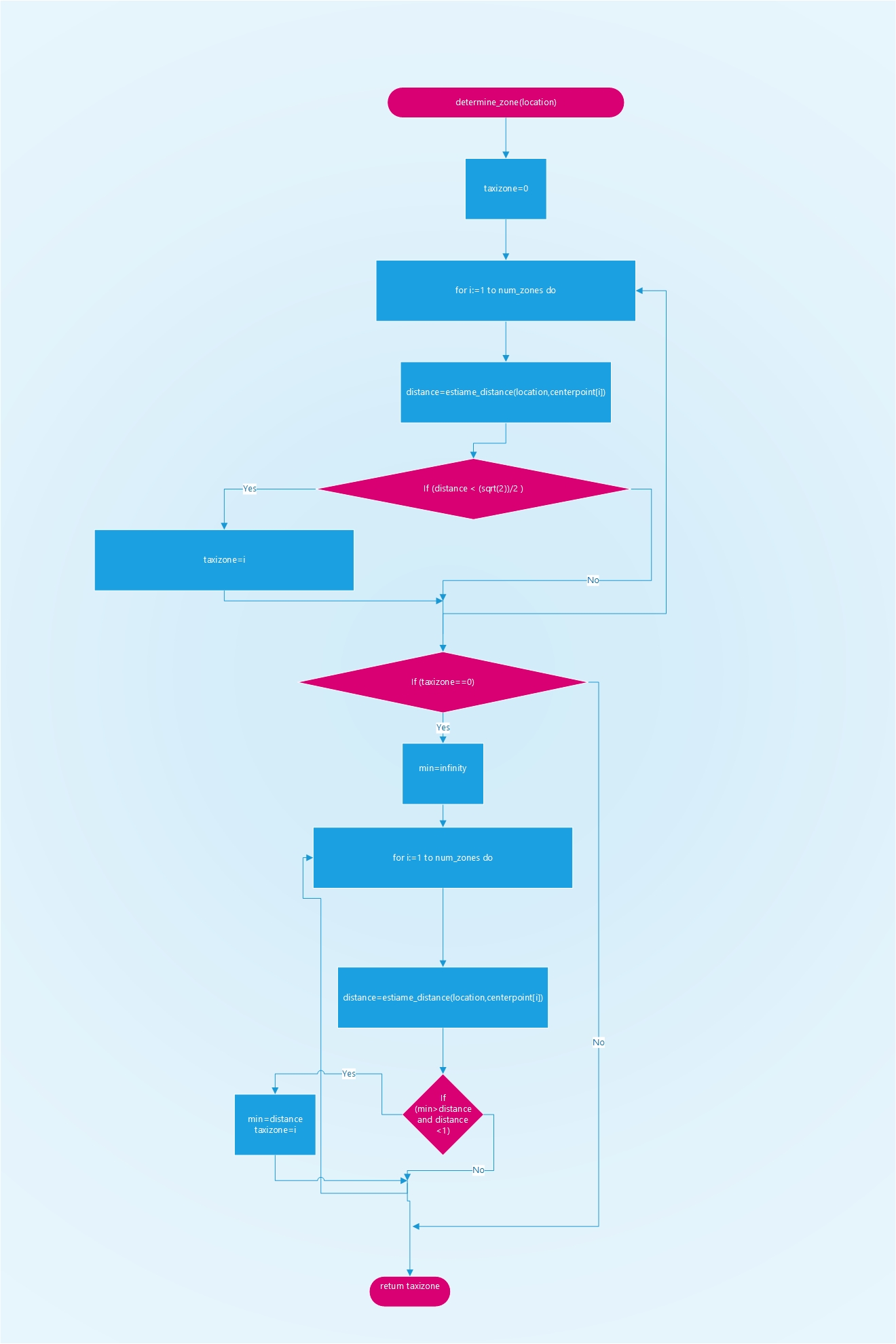
centeri – center corresponding to the ith taxi zone

1. Now, lets consider situatiion when the tested point doesn’t belong to inner circle. In this case, the outter circle is considered

After that, to find the corresaponding 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.

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

In what follows, the high-level algorithm interprettation 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/taxizone.jpg or corresponding visio project file.



## Cost estimation algorithm

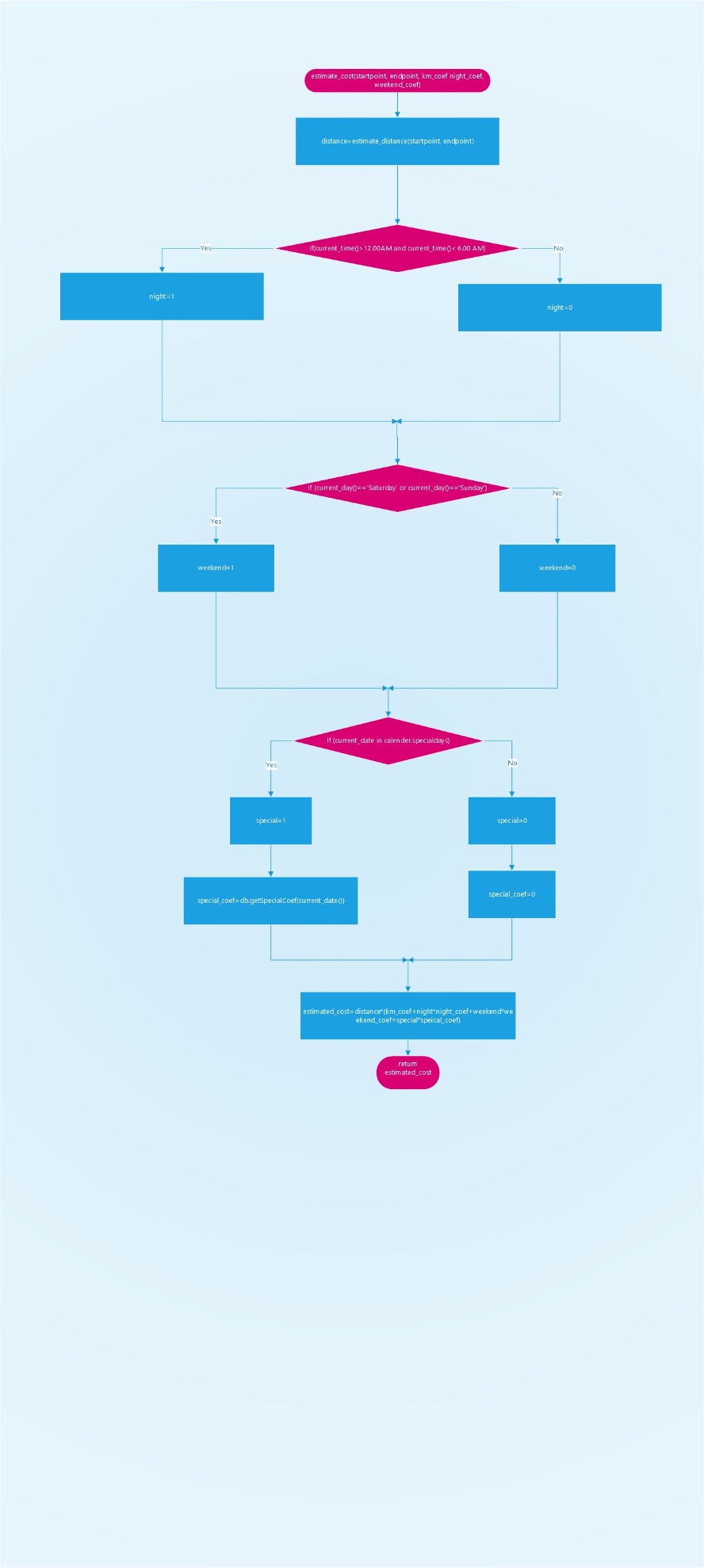
As a part of the drive negotitation protocol, cost estimation plays important role, as it gives overview to users what would be the price for a taxi service to the desired destination, so the user can either accept or reject the drive offer.

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, pricecould be lower or higher, depending on geovernment’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 interprettation 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.



## Functional Requirements

The following requirements are determined according to the Jackson and Zave analysis, so assuming that the domain properties, which are described above, hold for mentioned purposes, we construct the requirements in order to satisfy goals.

3.2.1 [G1] Only registered users can use the service

[R1] Visitor must fill in the form with his/her data

[R2] System checks fiscal code and password for correctness

[R3] Unregistered visitors can only see login page

[D1] Email address used for registration data must be formally correct

[D2] There is no more than one person with same fiscal code

[D3] User is not using someone else’s identity

3.2.2 [G2] Registered users can use service if and only if they are logged in

[R1] User has to fill form with credentials

[R2] Username and password entered during login process must be correct and validated by system

[R3] Service becomes available to users once they log in

3.2.3 [G3] User can request taxi to a desired destination using the service

[R1] User selects the desired destination

[R2] User can optionally enter maximum waiting time that he/she is able to wait for taxi to come

[D1] Taxi drive’s starting point is user’s current location

[D2] User can’t request another destination as a starting point

[D3] If user loses connection before the time runs out, the request is discarded from system

3.2.4 [G4] Allow user to modify his/her profile – high priority

[R1] User enters data using the form

[R2] Data validity is checked

[D1] Users can’t edit their fiscal code

[D2] Users can’t change their gender

3.2.5 [G5] Taxi drive negotiation – high priority

[R1] Taxi driver can see the request information

[R2] Taxi driver can accept or reject request

[R3] User can see the estimated price and time

[R4] User can accpet or reject drive offer

[R5] Request is forwarded to another taxi in same taxi zone if rejected

[R6] Possibility to determine the taxi zone using the current location

[R7] Queue of taxi drivers that can get requests are from same taxi zone as the user who requested a drive

[D1] User’s taxi zone corresponds to his/her current location

[D2] Taxi zone boundaries are regulated in advance

[D3] Taxi zones are related to one city only

[D4] Only available taxi drivers are considered in queue

[D5] User has 60 seconds to accept or reject the offer

[D6] One request is valid for 180 seconds. Within 60 seconds,user will get the repsonse if there is taxi available or not.

[D7] Taxi driver has 60 seconds to accept or reject the request

3.2.6 [G6] Taxi driver’s availability can be changed – high priority

[R1] Taxi drivers can click on availability slider to change their availability

[R2] System observes drive events and changes the availability according to real word changes

[D1] If driver clicks button and was available before, then becomes unavailable.

[D2] If driver clicks button and was unavailable before, then becomes available.

[D3] When drive starts, driver becomes unavailable.

[D4] When drive finishes, driver becomes available.

[D5] When report is submitted, taxi drive becomes available

[d6] When SOS is requested, taxi driver becomes available

3.2.7 [G7] Allow user and taxi driver to report the other side (user against taxi driver or taxi driver against the user) in case of bad behaviour – medium priority

[R1] Users and taxi drivers can initiate report clicking on “report” button

[R2] Users and taxi drivers must write provide text (reason) in which they describe the situation

[D1] In case of report, it is assumed that bad behavior occurred during drive event

3.2.8 [G8] Users that show frequent behaviour can be removed from system by administration – medium priority

[R1] Administrator can read reports

[R2] Administrator can see the users related to report

[R3] Administrator can delete user from system

[D1] Administrator decides is report is serious enough to be considered

[D2] Administrator cannot remove any other administrator user from system in any case

[D3] Administrators are employed by city government and have special credentials used for logging in.

3.2.9 [G9] Only eligible users approved by administrationcan become taxi drivers – high priority

[R1] Administrator can promote users in order to become taxi driver clicking on button

[R2] Administrator must find the desired user either scrolling the list or browsing by keywords

[R3] Administrator must enter driver’s car ID number and driving license number

[R4] System must check validity of entered driving license and car ID number

[R5] Administrator can downgrade driver to user again

[R6] Administrators can change driver’s info (for example, when driver changes car or license number)

[D1] User visits the administrator’s office to show real documents

[D2] People are using their own driving licenses

[D3] There are no people with same licenses

[D4] Each taxi driver can be registered with one car at the moment

[D5] Administrator decides to promote or downgrade user

3.2.10 [G10] Users and taxi drivers can request emergency service in case of an accident during drive – low priority

[R1] User or taxi driver sends an S.O.S signal clicking on the S.O.S button

[R2] System dispatches the closest possible emergency vehicle

[D1] Accident occured during drive event

[D2] Emergency vehicle is coming to sender’s location

[D3] Driver that responds to emergency group is dispatched, not a regular taxi driver

## The world and the machine

In what follows, will be given an overview of „The world and the machine“ (M.Jackson and P. Zove) approach application to MyTaxiService as an example.

I would clearly mention that the diagram bellow has only illustrative purposes and is not detailed overview, so only the most relevant parts are included, while some parts are omitted in orded to keep picture as understandable as possible.

As it can be seen, there are three main categories:

The World – portion of real world affected by machine

The Machine – portions of system to be developed

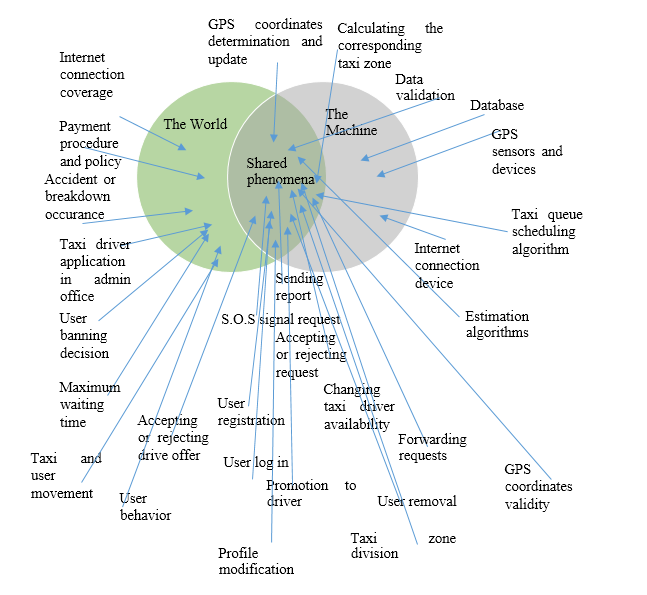
Shared phenomena- 1.can be controlled by world and observed by machine or 2. controlled by machine and observed by world and are, in practice, portions of system that have to be implemented.

Considering the results presented in 3.2, we can apply The world and the machine approuch to our system.

The World - As previously said, in set „The world“ are placed all these things that are related only to environment and out of machine range. For example, internet connection coverage is related to many environment factors. Payment procedure isn’t covered by this system yet and is performed only in world domain. User banning decision is brought by the administrator and is not based on machine algorithm. Accident or breakdown are events that are out of control of the machine and are result of the environment conditions. Maximum waiting time is not a result of any machine calculation and depends only on user’s decision and patience. User’s behavior is also in this part. Taxi and user movement are in this part. Taxi driver must come to administration office in real world to show driving license and car registration document in order to become eligible.

Shared phenomena – Mainly, here are the actions that users can take that will affect the real world. Taxi requests, accepting and rejecting taxi requests and offer – are used taken actions that are visible by machine and cause changes in real world. Sending a report or S.O.S signal also belongs to this category. When accident or bad behavior occurs in real world, the users react by sending a corresponding message to the system. Taxi drivers can change their availability, which is observed by the machine and affects the scheduling and queues, but is controlled by “The world” and “The Machine”, so it is shared phenomenon. Forwarding requests is also in this part, because it is controlled by “The Machine”, but affects the real world. Movement of cars corresponds to real world, but is observed by the machine which determines the corresponding taxi zone. GPS coordinates determination is observed by machine, but the coordinates themselves are changed by real world events (user and taxi movement). Registration belongs to this part, because user enters the data, but the process is observed by the machine and affects the database. After registration, user can use the service (the real world phenomena), but it is controlled by the machine itself, which has its own databases where it stores the registered users, so the machine could add new users. Profile modification also belongs to this part – the same reason. GPS coordinates validation (belonging to city or not, or particular taxi zone) also corresponds to this part, along with scheduling algorithm which needs to be implemented, because both of them are done and controlled by machine, but do affect the state and the situation of the real world. Estimation algorithms are also part of the system that has to be developed, and depends on both real world and machine. Pricing regulatory comes from business owners, but, in practice, does affect the algorithm performed by the machine itself. Promotion to a taxi driver or downgrading a taxi driver to user is also shared phenomenon because – in first case, when taxi driver needs to come to admin’s office in order to register, but the admin needs to enter his data in order to promote him to a taxi driver so it will affect the database and the system itself, but the real world also, because the particular user gets taxi driver role. User removal is also shared phenomenon, because it is performed by system on a database level, but it depends on administrative decisions and affects the real world at the same time – users removed from system cannot use the service anymore nor register again. Taxi zone division is done by system but takes some real world but takes into account some government decisions and affects the whole taxi scheduling and dispatching mechanism.

The Machine – This category contains only entities strictly related to computers, programs and communication devices that are necessary along with database with users and other data belongs here to ensure that system can perform its role.

In what follows, the diagram presents the facts previously stated.

## Scenarios

In this part of the document, 5 different real-life scenarios are described in order to illustrate how MyTaxiService could be used in practice and which situations can be covered by this system. Only the most interesting and complex scenarios are selected and presented, because the number of possible scenarios is quite large, but many of them are really trivial, so they are not described here.

3.4.1 Scenario 1

Alice wants to use MyTaxiService. She hasn’t previously used the system. First time she started application , she has figured out that she has to register in order to use MyTaxiService. She clicks the „Register“ button on the home page and then she is transfered to a registration form. On this form, she enters the required data (fiscal code, username, password, phone number confirms the password, then first name, last name, gender and even selects here profile picture from a mobile phone) and clicks on submit button. But, she gets message that states that her desired username already exists in database, so she needs to select another username. After long consideration, she finds another appropriate username and submits the data. Everything is valid, the database is updated, so she is now a registered user. She is back to a home page and can now enter her credentials, as she is so excited to use the service as soon as possible.

3.4.2 Scenario 2

Alice has entered her credentials and is logged in. From user menu, she selects „Taxi request“ button. After that, map is displayed to her. She lives in „Cassa dello Studente – Leonardo da Vinci“ and wants to go to Expo. She finds Expo and map, and selects it as a destination. She is so excited and confirms the request, but she forgot to enter maximum waiting time. She is in a hurry and hits the quit button after 2 seconds waiting for response. The system didn’t send her response, because she cancelled it before her request was accepted by taxi driver. She still wants to try out this new service, but this time she selects much closer destination - „Via Golgi 39“. She enters that she can wait for taxi only 2 minutes After that, she is transfered to response waiting screen. System transfers her request to the first taxi from a queue belonging to her taxi zone and is available. Bob is the taxi driver who received her request, but he rejected it, because he finished his working time and forgot to change his availability status. After that, the queue is checked. The request is forwarded to next taxi driver from her taxi zone. John is polled from the queue, because he needs only 1 minute to come to her place. He accepts Alice’s request. Alice gets notification that she has a taxi drive offer with details – the drive would cost 4.21 euros and she will have to wait 1 minute. She accepts the offer. John is coming to her location.

3.4.3 Scenario 3

Alice is in John’s car, on the way to the desired destination. John has a bit crush on Alice and starts to behave in a way that Alice didn’t like. She told him that she has a boyfriend, but he is still behaving the same way. They are about 400m from the destination. Alice is angry and clicks the „Report“ button. She describes the situation into reason field and submits the report. Report and drive event details are inserted into system. Drive vent ends and system destroys the object. John stops car. Alice exits the car. John’s taxi is available again and is available in queue belonging to his current taxi zone.

3.4.4 Scenario 4

Nenad is one of the administrators. He is notified about the report and views it. He decides to see John’s profile. Nenad sees that this is not John’s first time to be reported for such thing, so the administrator chooses the most severe measure – to delete John from system. He clicks on „Delete“ button. John’s fiscal code is transferred to a black list and he is no longer able to register, log in or work as a taxi driver using this service.

3.4.5 Scenario 5

One of Nenad’s friends comes to MyTaxiService administration office. Friend told him that he had bought a car. Nenad is amazed a bit, because he knew that his friend had difficulties passing driving license exam. Friend told him that he wants to work as a taxi driver. Nenad finds his friend in database and clicks on „Promote“ button. After that the form for taxi driver details appears.

* Which car do you have? - Nenad asks.
* Zastava 101 ! – friend responds as quick as possible
* Car registration number? –Nenad asks.

His friend politely reads him car number from a mobile phone. Nenad immediately wants to check the validity of car registration number. It turns out that the number is not valid. Nenad asks again. The friend honestly told him that he tried to lie to him, but didn’t know that system checks the data validity using the government database. Nenad is angry and explains his friend that he has luck because the system checks data validity.

* I suppose that It’s better not to ask you if you have a driving license! You could risk your life driving without license. But you don’t even have a proper car registration number.

The friend lost a good opportunity to find a job, and leaves the office.

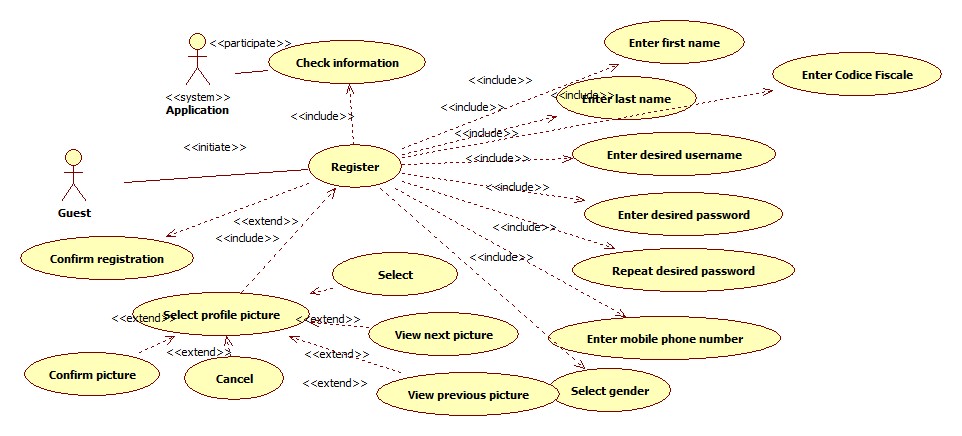
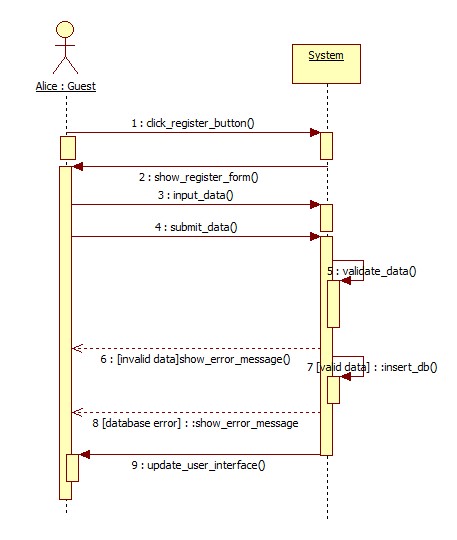
## UML Diagrams

3.5.1 Use Case and Sequence Diagrams

In this part, use cases are given in order to describe what can system do, which are cases when it can be used and how it can be used in these cases.

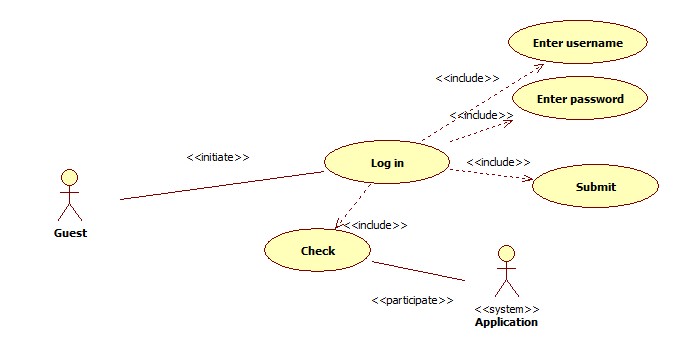
3.5.1.1 Register

|  |  |
| --- | --- |
| Actor | Guest |
| Goal | Users are added to database (in order to access the service) |
| Input Condition | NULL |
| Event Flow | 1. Guest on the home page clicks on “Register”. button to start the registration process. 2. Guest fills in at least all mandatory fields and selects picture (optional). 3. Guest clicks on “confirm” button. |
| Output Condition | Visitor succesfully ends registration process and becomes a Registeref User. From now on he/she can log in to the application using his/her credential and start using MyTaxiService and is added to database. |
| Exception | 1. The visitor is already a user. 2. One or more mandatory fields are not valid. 3. Username choosen is already used by another user. 4. Password isn’t according to security policy 5. Fiscal code is not valid, already in use or is blacklisted (banned)   System participates in this use case by checking the data validity after pressing the „confirm button“. All exception are handle alerting the visitor of the problem and application goes back to point 2 of Event Flow. |

The following sequence diagram is based on Scenario 1.

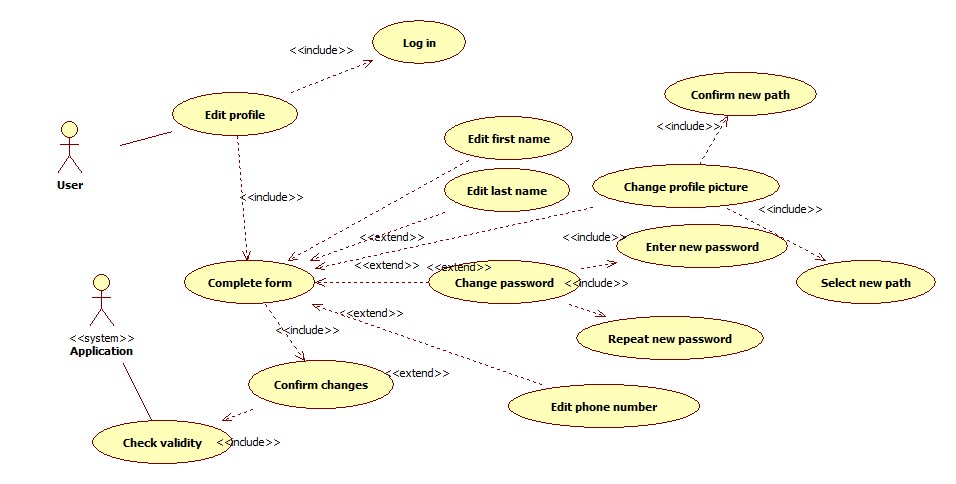
3.5.1.2 Login

|  |  |
| --- | --- |
| Actor | Guest |
| Goal | Guest that already have profiles log in, so they can access service |
| Input Condition | Guest is previously registered into the system. |
| Event Flow | 1. MyTaxiService shows the login page to visitor. 2. Visitor completes the form inserting correct username and password. |
| Output Condition | 1. MyTaxiService verifies the credential of visitor and if correct show the menu page with enabled functions (depending on which type of user is logged in). 2. Guest becomes User and can acccess the service. |
| Exception | 1.If username and/or password are incorrect or not exist, MyTaxiService notifies that to guest and shows the login page again.  2. If user is blacklisted, he/she will also get a message that can’t use the service.  System actively participates in this part, by checking the data validity. |



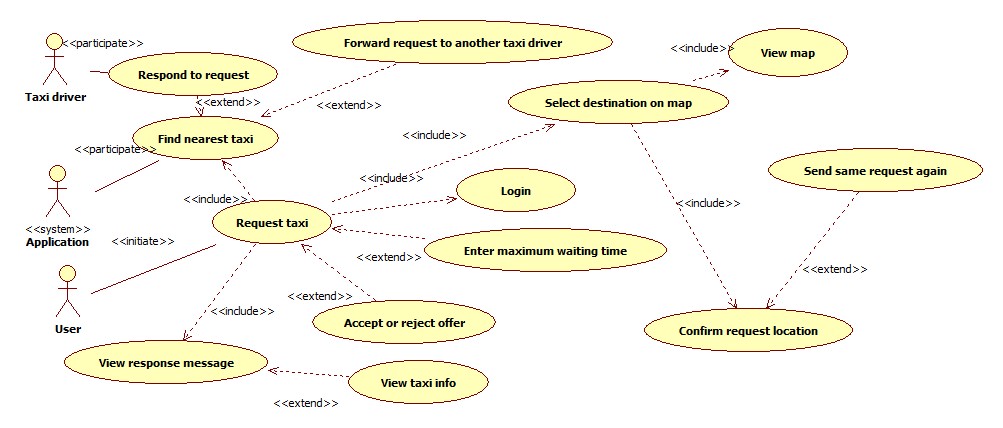
3.5.1.3 Profile modification

|  |  |
| --- | --- |
| Actor | User |
| Goal | User changes his/her data after registration |
| Input Condition | Registered User is already logged in into MyTaxiService. |
| Event Flow | 1. Registered User click on „Modify“ button 2. User completes form with new data 3. User confirms changes |
| Output Condition | User’s data is changed by new data entered by himself/herself. Database is updated. |
| Exception | 1. New data not is not valid. System participates in this use case by checking the new data entered by user. If it is not valid, user has to do steps 2. and 3. again. |

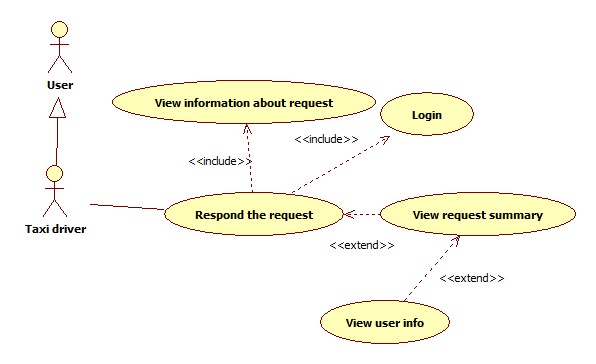


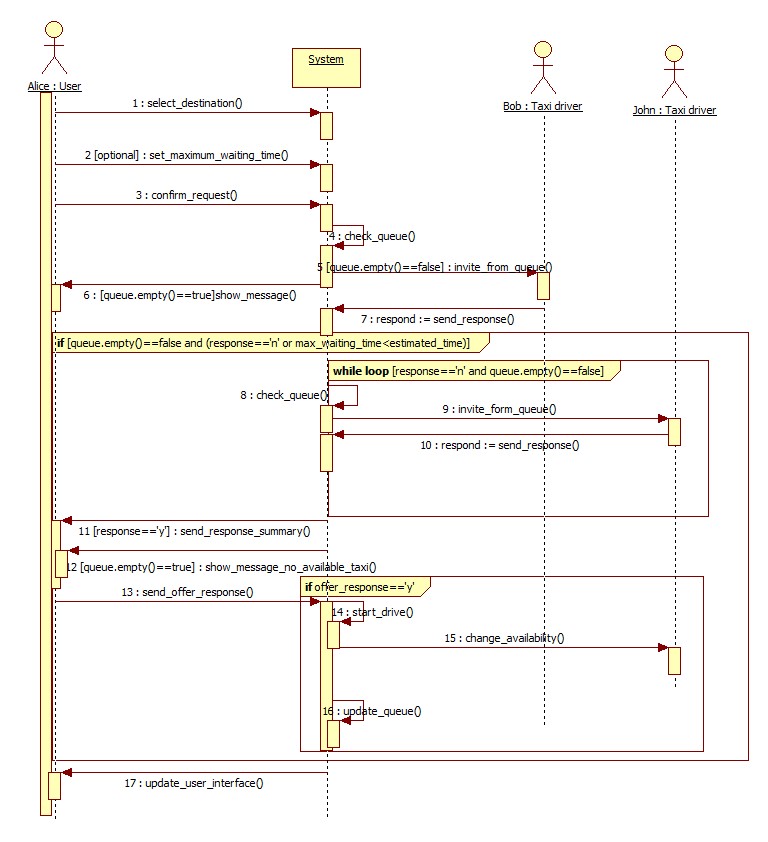
3.5.1.4 User negotiation for taxi request to a desired destination .

|  |  |
| --- | --- |
| Actor | User, Taxi Driver |
| Goal | Users and taxi drivers negotiate about taxi drive for a desired destination |
| Input Condition | 1. User and taxi driver are already logged in into MyTaxiService.  2. Taxi driver is using mobile application. |
| Event Flow | 1. User triggers „Taxi request“ button 2. User selects destination on the map 3. User optionally enters maximum waiting time 4. User confirms request 5. System finds taxi from queue 6. Taxi driver responds to a request by rejecting or accepting it 7. User gets response with offer if there is available taxi 8. User accepts or rejects the offer. |
| Output Condition | User gets response and has scheduled taxi drive |
| Exception | 1. Taxi rejects the request – System forwards it to next taxi in queue 2. User rejects the offer – System cancels the request 3. No taxi available – System sends message to user that there is no taxi available 4. User loses internet connection before confirming the offer – Request is deleted from system 5. User loses internet conncetion after confirming offer – Request is considered as still valid |



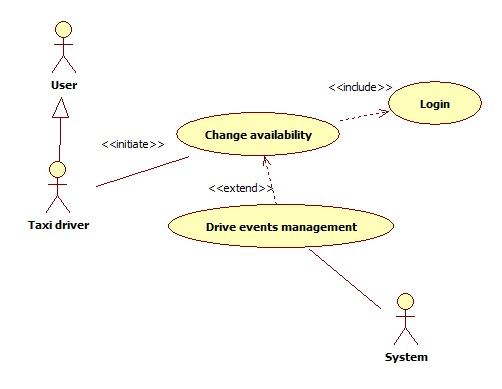
Taxi driver can respond or reject the request. It includes a smaller user case where taxi driver views the request data or, optionally, user info, and decides to accept or reject it.



The following sequence diagram is based on Scenario 2.

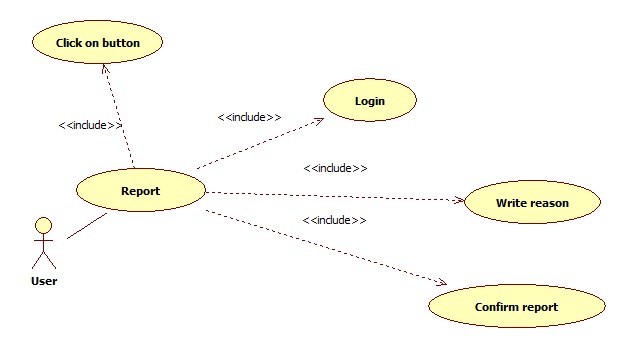
3.5.1.5 Taxi driver availability change.

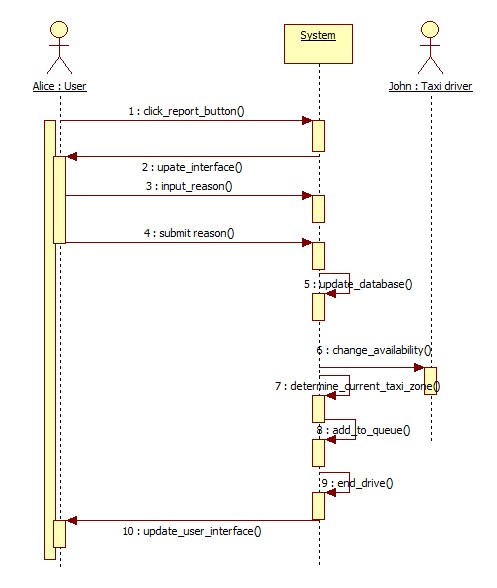
|  |  |  |
| --- | --- | --- |
| Actor | Taxi driver |  |
| Goal | Availability status of taxi driver can be changed | |
| Input Condition |  |  |
|  | 1. | Taxi driver must be registered. |
|  | 2. | Taxi driver must be logged in into mobile application. |
|  |  |  |
| Event Flow | 1.Taxi driver clicks on change availability button. | |
|  |
|  |
|  |
|  |
| Output Condition | Previous availability status is changed or corrected by system | |
| Exception | 1. Drive event is started or ended   System can react and change availability in this cases. | |



3.5.1.6 Report user

|  |  |  |
| --- | --- | --- |
| Actor | User |  |
| Goal | Report other user for bad behaviour during taxi drive | |
| Input Condition |  |  |
|  | 1. | User must be registred. |
|  | 2. | User must be logged in into application. |
|  | 3. | User is in drive event. |
| Event Flow | 1.User pushes report button.  2. User writes reason for reporting.  3. User confirms the report. | |
|  |
|  |
| Output Condition | Report is added into database, so administration can later review it. | |
| Exception | 1. Empty report reson field 2. Internal database error   User is returned to report page again. | |



Sequence diagram based on Scenario 3.

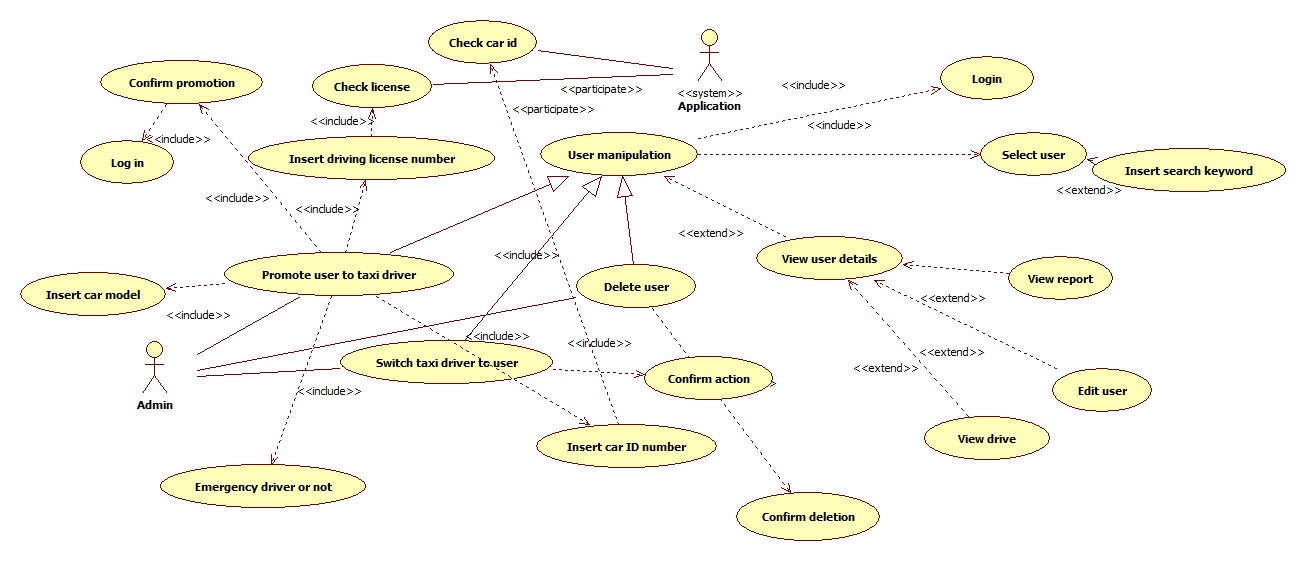
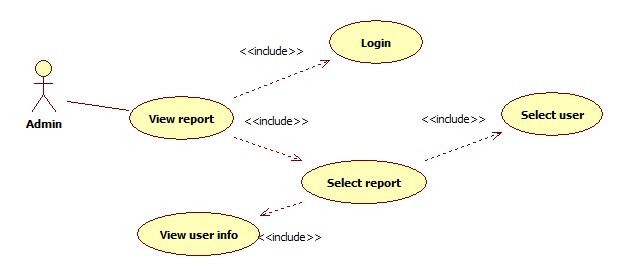
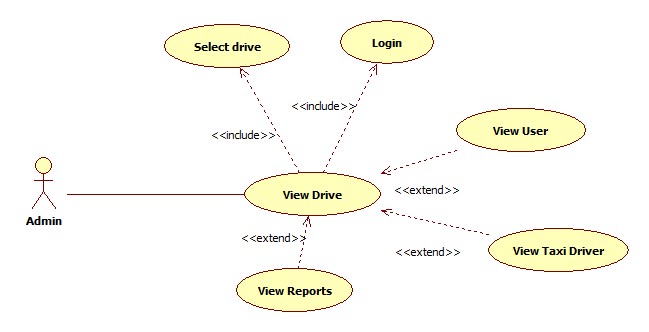
3.5.1.7 Administrator deletes reported users

|  |  |
| --- | --- |
| Actor | Administrator |
| Goal | Remove users who don’t take MyTaxiService serious enough from system |
| Input Condition | 1. Administrator must be registered by government. 2. Administrator must be logged in into application. |
| Event Flow | 1. Administrator browses user database. 2. Administrator selects desired user. 3. Administrator views reports related to user. 4. Administrator views drive events related to user. 5. Administrator clicks on delete burron. 6. Administrator confirms operation. |
| Output Condition | Database is updated and selected user is prevented from using MyTaxiService. |
| Exception | 1. Internal database error.   Exception is handeled alerting the administrator of the problem and application goes back to point 1. of Event Flow |

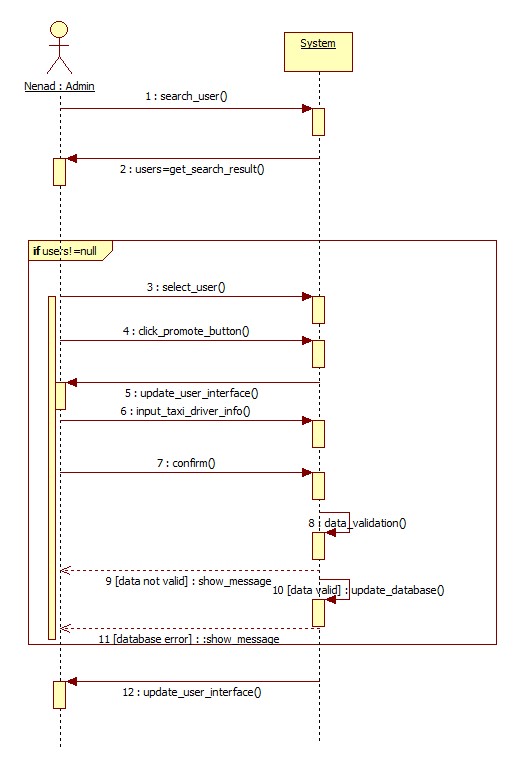
Use case diagram is included as a part of more complex diagram presented in 3.5.1.9., in order to better explain how this case is used as apart of more complex use case.

3.5.1.9 Administrator promotes user to taxi driver

|  |  |
| --- | --- |
| Actor | Administrator |
| Goal | User becomes taxi driver if eligible |
| Input Condition | 1. Administrator must be registred. 2. Administrator must be logged in. |
| Event Flow | 1. Administrator selects the desired user. 2. Administrator clicks on „Promote to driver“ button. 3. Administrator enters the necessary data using form. 4. System checks data. 5. Administrator confirms taxi driver promotion. |
| Output Condition | Previously selected user is now also a taxi driver, database is updated. |
| Exception | 1. Driving license invalid. 2. Taxi driver with same license already exists. 3. Car identification number invalid. 4. Car with same identification exists. 5. Internal database error.   System participates by checking the data. All exception are handled alerting the administrator of the problem and application goes back to step 3 from event flow. |

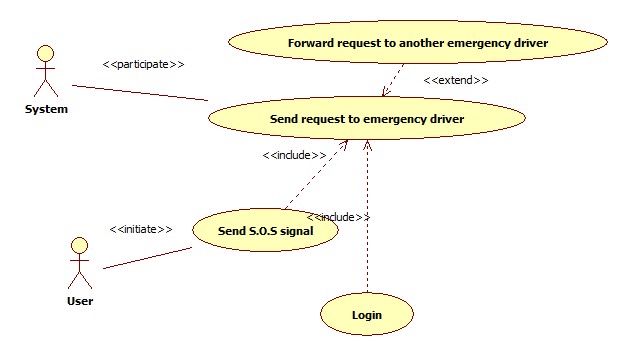


Sequence diagram that explains how the realization in practice is based on Scenario 5 and presented below.



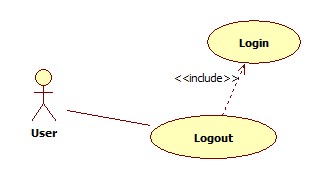
3.5.1.10 User sends S.O.S signal in case of an accident .

|  |  |
| --- | --- |
| Actor | User |
| Goal | Request emergency vehicle to a destination where the accident is reported. |
| Imput Condition | 1. User is registered and logged in. 2. User is into drive event. |
| Event Flow | 1. User reports energency by clicking on S.O.S button. |
| Output Condition | Emergency vehicle is dispatched from queue. |
| Exception | 1. No vehicle for taxi zone available. System searches for a vehicle in other taxi zones until a vehicle is available |



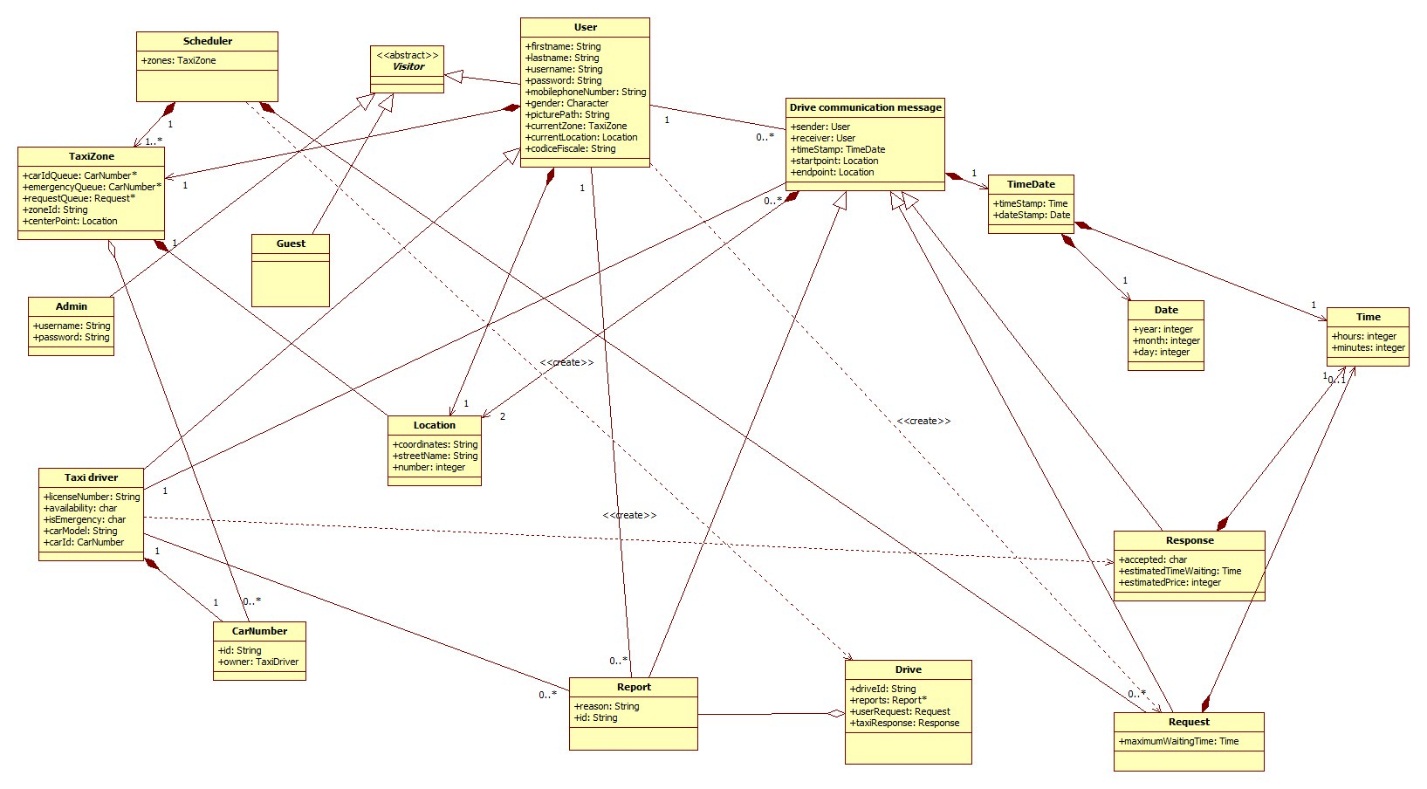
3.5.1.11 Log out

|  |  |
| --- | --- |
| Actor | User |
| Goal | Log out from system and lose access to service features |
| Imput Condition | 1. User already registered and logged in. |
| Event Flow | 1. User click on „Log out“ button |
| Output Condition | User is back to home page and must log in again in order to use MyTaxiService. |
| Exception | NULL |



3.5.2 Class Diagram

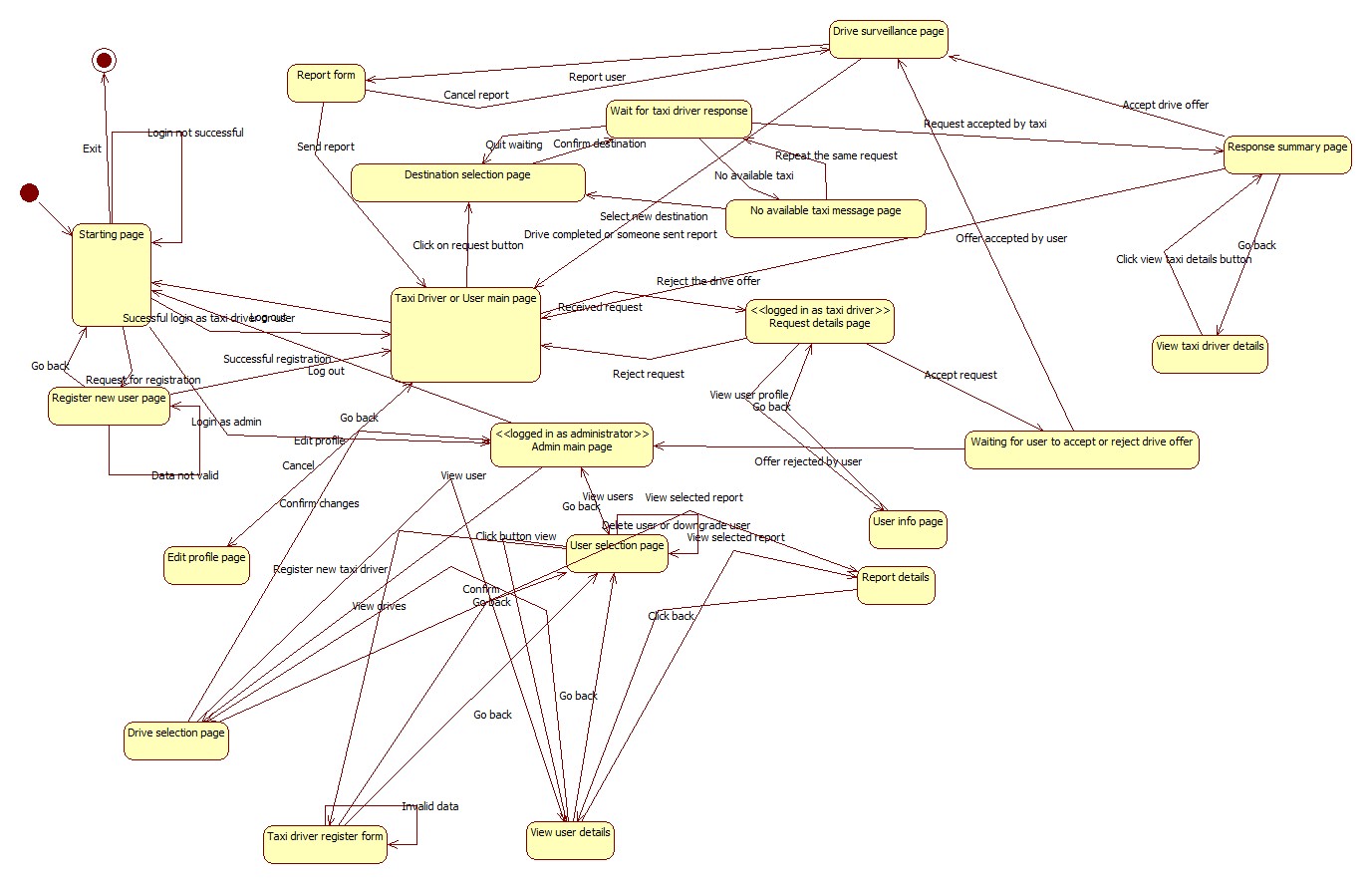
In this part, the class diagram is presented. This diagram will be updated during the design and development process, because all the operations are missing at the moment, as it can be seen. Mostly, the requirements-level related entities as a part of the application are presented.



3.5.3 State Machine Diagram

Application state diagram

In this part, the state machine diagram is presented. The following state machine diagram gives a simplified vision and overview of the entire application. Together with Mockups from 3.1.1, this diagram gives a better introduction to how system resonds to user action in different cases, corresponding to different events from real world and user actions.



Availability change state diagram

This state diagram shows in which cases the taxi driver is considered available and in which cases is considered as not available. So, this are the facts presented here and the situations in which the change occurs are described.

As it can be seen, availability could be changed manually, but a part of the system monitors the drive events and changes the availability in case of some events.

In order to be a part of the queue, taxi driver needs to be „available“.

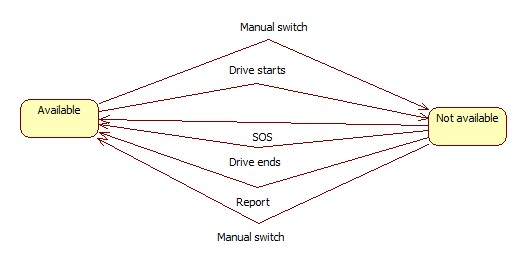
After the beginning of drive event, taxi driver becomes automatically „unavailable“.

When the drive ends, the taxi driver becomes „available“ again.

After the report of any side, the drive ends and taxi becomes „available“.

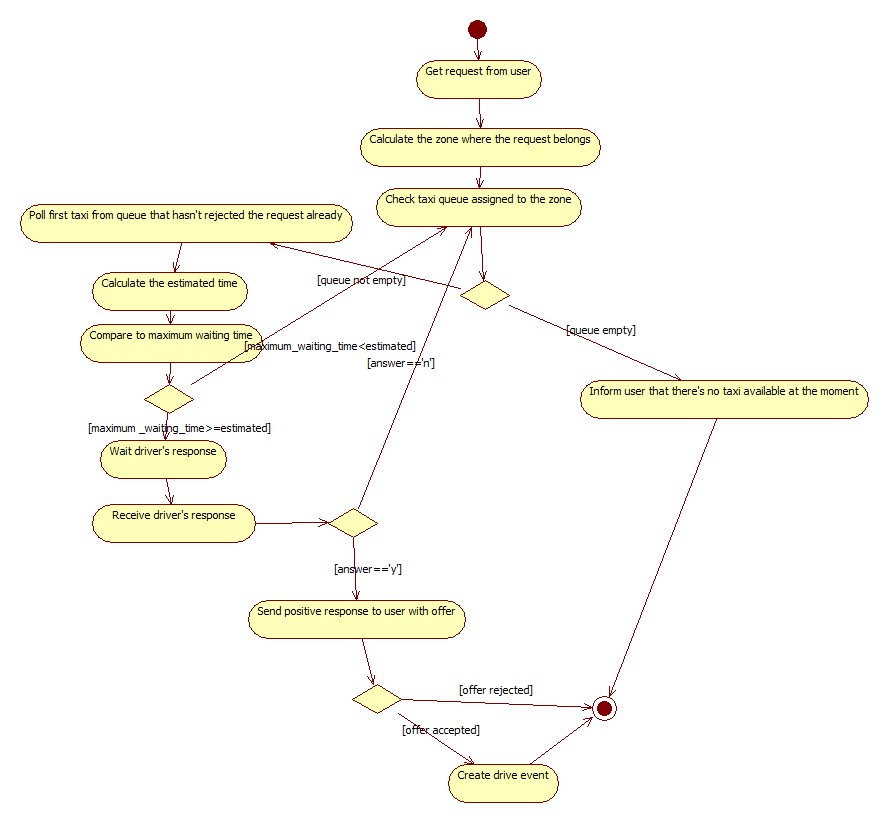
When SOS signal is sent, the drive ends and taxi becomes available, but taxi driver can still manually set availability. This could be very useful in case when the drive is ended and taxi is not damaged and user can continue drive.

But, in case when taxi driver can’t countinue his/her job, he/she is given ability to manually change the availability to „unavailable“.



3.5.3 Activity Diagram

In this part, the activity diagram for taxi scheduling is presented. The following activity diagram gives an overview which activities are performed by the system, and the idea how the algorithm for scheduling works and behaves in certain conditions – and will support the development of the algorithm in design phase.



## Non Functional Requirements

3.6.1 Performance Requirements

We assumed that user will get answer within maximum 180 seconds (searching for taxi is limited to maximum 180 seconds per request in order to load balance), but, if taxi chages availability during that period, system needs to respond in real time to that changes. So, the response time of the system, to be usable in real time should be less than 1 second per action, because unstability of internet connection in certain areas is assumed.

In terms of maximum active users at certain point in time, considering that this application will be used in a city environment, the maximum number of simultaneus users should be at least 5 000.

3.6.2 Minimum software, hardware and communication interface requirements

3.6.2.1 Server-side application

Software

|  |  |
| --- | --- |
| OS | Any OS that supports Java Virtual Machine, DBMS and Application Server |
| DBMS | MySQL 5.6 or newer |
| Java Virtual Machine | 7.0 or greater |
| Application server | GlassFish 4.1 or greater |

Hardware

|  |  |
| --- | --- |
| Memory | 8 GB or more |
| Free storage space | 20 GB or more |

Communication

|  |  |  |
| --- | --- | --- |
| Protocol | Application | Port |
| TCP | HTTPS | 443 |
| TCP | HTTP | 80 |
| TCP | DBMS | 3306 ( default ) |

3.6.2.2 Client-side web-browser application

Software

|  |  |
| --- | --- |
| Web browser | Any |

Hardware

|  |  |
| --- | --- |
| Display resolution | 1024 x 600 or greater |

Communication

Active Internet connection.

3.6.2.2 Client-side mobile application

Software

|  |  |
| --- | --- |
| OS | Android 4.0 or greater |

Hardware

Any GPS-enabled Android 4.0 or greater device, with resolution at least 800x480.

Connection

Active Internet connection and GPS turned on.

3.6.3 Software System Attributes

3.6.3.1 Availability: The application would be accessible online 24/7. To achieve this it is necessary to use a dedicated server, but, in order to guarantee even better availability, the whole system could be hosted on already available cloud platform. System downtime should be less than 30 minutes per month.

This solution give more scalability to performance required by the system and could reduce the cost for a dedicated server, mantaining a high level of performance especially in case of full load (advanced load balancing).

3.6.3.2 Maintainability: Regular updates are planned, and the change of princing estimation should be as flexible as possible, because each city’s government could change pricing policy any time and each city could have different taxi policy. API is provided that could be used to extend system by other developers and system will be documented in details. So, in case of critical situations or other cases, system could be modified or extended.

3.6.3.3 Scalability: The system should be expandable to handle greater number of active users and also will be extendable – so the new functionalities could be added in future aas described in 2.5.

3.6.3.4 Design constraints: The system will inherite all the design constraints of the programming languages (Java) , frameworks used (Glassfish,Java Eclipse, Android Studio) and DBMS (MySQL). The part of the system that deals with maps and GPS coordinates will rely on Google Maps API as an external API developed by Google and API provided in Android Studio that deals with coordinates and maps.

3.6.4 Security

3.6.4.1 External interface side of MyTaxiService application implements a login authentication to protect the information of users.

There are no special requirements for the user side of the application, but user password has to be at least 8 characters long, containing at least one capital letter, one special symbol and one number. User can change password anytime using the profile modification page. In future, captcha verification could be added in order to prevent attacks.

In future, face recognition based authentication system will be developed in order to identify administrators, together with password. Combination of these two technologies will defend system from being managed by unauthorized people and make system more durable to attacks.

3.6.4.2 Application side would implement defense from SQL injection. Also, the application would use HTTPS instead of HTTP connection, in order to guarantee communication integrity and confidentiality.

3.6.4.3 Server Side – will use separated web, application and database part, with firewall between each of them to prevent unauthorized users from access. Because cloud platform use is still considered (Amazon Services), there might be different solution, offered by Amazon.

One of the possible solutions is to use Server-Side Encryption with Amazon S3-Managed Keys (SSE-S3) , where each object is encrypted with a unique key employing strong multi-factor encryption. As an additional safeguard, it encrypts the key itself with a master key that it regularly rotates. Amazon S3 server-side encryption uses one of the strongest block ciphers available, 256-

bit Advanced Encryption Standard (AES-256), to encrypt data.

# Appendix

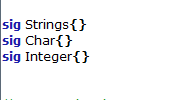
## Alloy

In this part of the document, Alloy model is presented – its signatures, facts, asserts, predicates and along with its results. The whole model file (Alloy.als) could be found under Deliveries folder in my github repository.

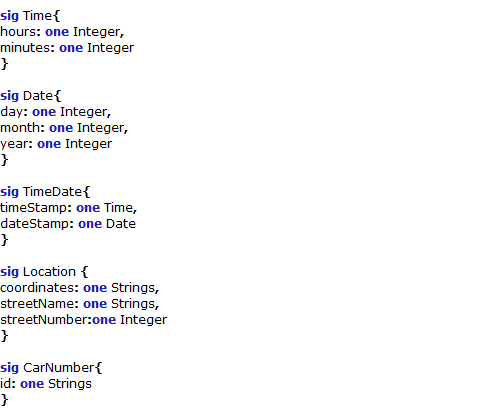
4.1.1 Signatures

In this part, definition of datatypes are given – from common types to domain entities.

Common data types found in almost every programming language, necessary for this model



Auxiliary data types used in this problem – related to space and time.

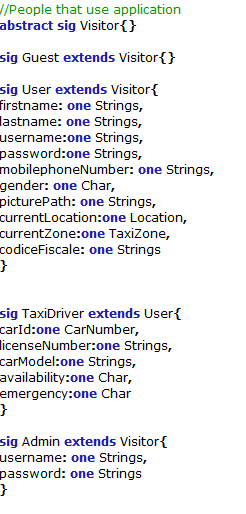
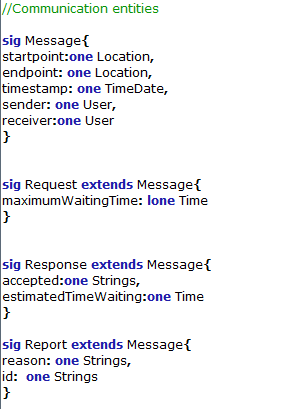
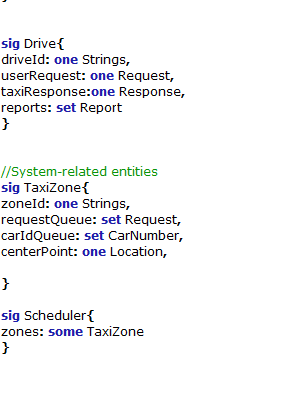


Entity signatures

Signatures of all the entities that appear in this problem.

First, the signatures of actors are given (left). These, are in fact, abstractions of a people who use the application.

Then, communication entity signatures, of that are used in protocol are given (right).



And finally, the last tree signatures are written :

-Drive – an abstraction of a drive event

-TaxiZone – an abstraction of taxi zone

-Scheduler – part of the system which deals with taxi zones and their queues

4.1.2 Facts

Fact is used to *force* something to be true of the model. Facts are always „on“.

A trivially false fact results in there being no solutions to the model no matter what the rest of the model looks like. This is an extreme version of [overconstraining](http://alloy.mit.edu/alloy/tutorials/online/sidenote-overconstraint.html) your model. Overconstraint is dangerous, since it means that any other assert statement that you check will trivially have no solutions, thus potentially masking both errors in your model and (worse) failings in the subject matter you are modeling.

In what follows, facts about MyTaxiService are written, after consideration of domain rules, constraints and properties.

//noEmpty-alike facts

fact noEmptyTime{

all t:Time | (#t.hours=1) and (#t.minutes=1)

}

fact noEmptyDate{

all d: Date | (#d.day=1) and (#d.month=1) and (#d.year=1)

}

fact noEmptyTimeDate{

all td:TimeDate | (#td.timeStamp=1) and (#td.dateStamp=1)

}

fact noEmptyLocation{

all l:Location | (#l.coordinates=1) and (#l.streetName=1) and (#l.streetNumber=1)

}

//User mandatory fields

fact noUserMissingInfo{

all u:User | (#u.username=1) and (#u.password=1)

}

//User mandatory fields

fact noSenderMissing{

all m:Message | (#m.sender=1)

}

//Admin mandatory fields

fact noUsernameMissingAdmin{

all a:Admin | (#a.password=1)

}

fact noPasswordMissingAdmin{

all a:Admin | (#a.password=1)

}

//

//Scheduler must have at least one zone

fact minZones{

all s:Scheduler| (#s.zones>1)

}

//Taxi driver is eligible if and only if has a car and license number

fact noUnlicensedAndCarlessTaxiDriver{

all t:TaxiDriver | (#t.carId=1) and (#t.licenseNumber=1) and (#t.carModel=1)

}

//No duplicate and overlap-alike facts

//No duplicate usernames

fact noDuplicateUser{

no u1,u2: User |(u1!=u2) and (u1.username=u2.username)

}

//No possibility for one user to register more than once with a same fiscal code

fact noFakeProfiles{

no u1,u2: User | (u1!=u2) and (u1.codiceFiscale=u2.codiceFiscale)

}

//

//No self-communication allowed - all the requests,responses and report are towards differnt users

fact noSelfCommunication{

all u:User, m:Message | not ( (m.sender=u) and (m.receiver=u) )

}

//No sending requests, responses or reports to same location

fact noSameStartAsEnd{

all m:Message | not ( m.startpoint=m.endpoint )

}

//No taxi zones with same center points

fact noSameZones{

no z1,z2: TaxiZone | (z1!=z2) and (z1.centerPoint=z2.centerPoint)

}

//No taxi zones with same ids

fact noSameIdZones{

no z1,z2: TaxiZone | (z1!=z2) and (z1.zoneId=z2.zoneId)

}

//No car number belonging to different owners at the same time

//fact noSameCarOwner{

//no cn1, cn2: CarNumber | (cn1!=cn2) and (cn1.owner=cn2.owner)

//}

fact noSameCarNumbers{

no cn1, cn2: CarNumber | (cn1!=cn2) and (cn1.id=cn2.id)

}

//No two taxi drivers with same car

fact noSameDriverCars{

no td1, td2: TaxiDriver | (td1!=td2) and (td1.carId=td2.carId)

}

//No taxi drivers with same license id allowed

fact noSameLicense{

no td1,td2: TaxiDriver | (td1!=td2) and (td1.licenseNumber=td2.licenseNumber)

}

//No two drives with same ids

fact noSameDriveId{

no d1,d2: Drive | (d1!=d2) and (d1.driveId=d2.driveId)

}

//A taxi drivers can respond to requests that are from users in that taxi zone

fact responseCondition1{

no r:Request | r.sender. currentZone!=r.receiver.currentZone

}

fact responseCondition2{

no r:Response | r.sender. currentZone!=r.receiver.currentZone

}

//No two requests from same sender can be in one queue of request per taxi zone

fact noTwoZonesSameSender{

no r1,r2:Request | some z:TaxiZone|

r1!=r2 and (r1 in z.requestQueue) and

(r2 in z.requestQueue) and (r1.sender=r2.receiver)

}

//One request can't belong to queues of two different taxi zones at the same time

fact noTwoZonesSameRequest{

no r: Request | some z1, z2:TaxiZone |

z1!=z2 and (r in z1.requestQueue) and

(r in z2.requestQueue)

}

//Are always the right actors reported in a drive

fact driveReportsRule{

all d:Drive,r:Report| (r in d.reports) and ( (r.sender =d.userRequest.sender) or (r.sender=d.userRequest.receiver) )

}

//The condition when a drive is considered correct

fact correctDrive{

all d:Drive| (d.userRequest.sender = d.taxiResponse.receiver)

}

//One taxi can't belong to queues of two different taxi zones

fact oneTaxiCanOnlyBeInOneTaxiZone {

no t: TaxiDriver | some z1, z2:TaxiZone |

z1!=z2 and (t.carId in z1.carIdQueue) and

(t.carId in z2.carIdQueue)

}

//All zones belong to one scheduler per city

fact allBelong{

all t:TaxiZone| some s:Scheduler| (t in s.zones)

}

4.1.3 Asserts

While a fact is used to force something to be true of the model, an assert is a claim that something must already be true due to the rest of the model.

//Checking if there are same requests in two zones

assert NoTwoZonesSameRequest{

no r: Request | some z1, z2:TaxiZone |

z1!=z2 and (r in z1.requestQueue) and

(r in z2.requestQueue)

}

check NoTwoZonesSameRequest for 5

//Checking if there is no self-message

assert NoSelfMessage{

no m: Message | m.sender=m.receiver

}

check NoSelfMessage for 5

//Checking add request

assert addRequest{

all r:Request, t1:TaxiZone,t2:TaxiZone | (r not in t1.requestQueue) and addRequestToTaxiZone[r,t1,t2] implies (r in t2.requestQueue)

}

check addRequest for 5

assert deleteRequest{

all r:Request, t1:TaxiZone,t2:TaxiZone | (r in t1.requestQueue) and removeRequestFromTaxiZone[r,t1,t2] implies (r not in t2.requestQueue)

}

check addRequest for 5

//Are always the right actors reported in a drive

assert driveReports{

all d:Drive,r:Report| (r in d.reports) and ( (r.sender =d.userRequest.sender) or (r.sender=d.userRequest.receiver))

}

check driveReports for 5

//DriveRule

assert driveRule{

all d:Drive| (d.userRequest.sender = d.taxiResponse.receiver)

}

check driveRule for 5

//Should find counterexample!- Simulating a situation where two taxi drivers register with same car

//A trivially false assert will result in counterexamples being found when it is checked.

//Since anything makes such an assertion false, any example of the system will be a counterexample.

assert assignSame1{

all cn:CarNumber, t1,t2,t3,t4:TaxiDriver| AssignCarNumber [cn, t1, t2] and AssignCarNumber [cn,t3,t4]

}

check assignSame1 for 10

//Shouldn't find a counterexample this time

assert assignSame2{

all cn:CarNumber, t1,t2,t3,t4:TaxiDriver| AssignCarNumber [cn, t1, t2] and AssignCarNumber [cn, t3, t4] implies (t4=t2)

}

check assignSame2 for 10

//Should illustrate that it is not possible to send message with same sender and receiver

//It prevents respond the request of yourself

assert selfMessage{

all m1,m2,m3:Message,u:User | AssignReceiver[m1,m2,u] and AssignSender[m2,m3,u]

}

check selfMessage for 5

assert RegisterUser1{

all u1,u2,u3,u4: User,name:Strings | RegisterUser[u1,u2,name] and RegisterUser [u3,u4,name]

}

check RegisterUser1 for 5

assert RegisterUser2{

all u1,u2,u3,u4: User,name:Strings | RegisterUser[u1,u2,name] and RegisterUser [u3,u4,name] implies (u4=u2)

}

check RegisterUser2 for 5

4.1.4 Predicates

This is the part where the predicates that are used are presented with the previous assert to verify the model.

//To genrate world example

pred show(){

#User=2

#TaxiDriver=2

#Guest=2

#Drive=2

#Strings=2

#Char=2

#Integer=2

#Time=2

#Date=2

#TimeDate=2

#Request=2

#Response=2

#Location=3

#Scheduler=1

#TaxiZone=2

#CarNumber=2

}

run show for 5

//Adding request to a taxi zone

pred addRequestToTaxiZone(r:Request, t1,t2:TaxiZone)

{

r not in t1.requestQueue implies t2.requestQueue=t1.requestQueue+r

}

run addRequestToTaxiZone for 5

//Removing request from a taxi zone

pred removeRequestFromTaxiZone(r:Request, t1,t2:TaxiZone)

{

t2.requestQueue=t1.requestQueue-r

}

run removeRequestFromTaxiZone for 5

//Assigning taxi driver a car number

pred AssignCarNumber(cn:CarNumber, td1,td2:TaxiDriver)

{

td2.carId=td1.carId+cn

}

run AssignCarNumber for 5

//Simulating how system assign recevier from schedule

pred AssignReceiver(m1,m2:Message,u:User)

{

m2.receiver=m1.receiver+u

}

run AssignReceiver for 5

//Simulate sending of the message

pred AssignSender(m1,m2:Message,u:User)

{

m2.sender=m1.sender+u

}

run AssignReceiver for 5

//Registration of the user

pred RegisterUser(u1,u2:User,name:Strings)

{

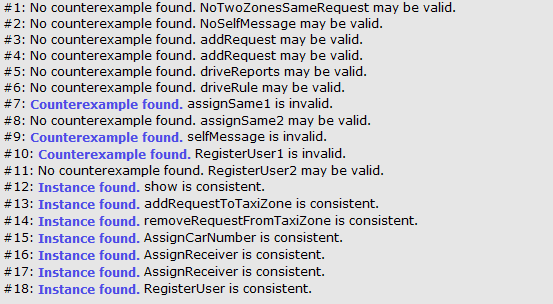
u2.username=u1.username+name

}

run RegisterUser for 5

4.1.5 Results

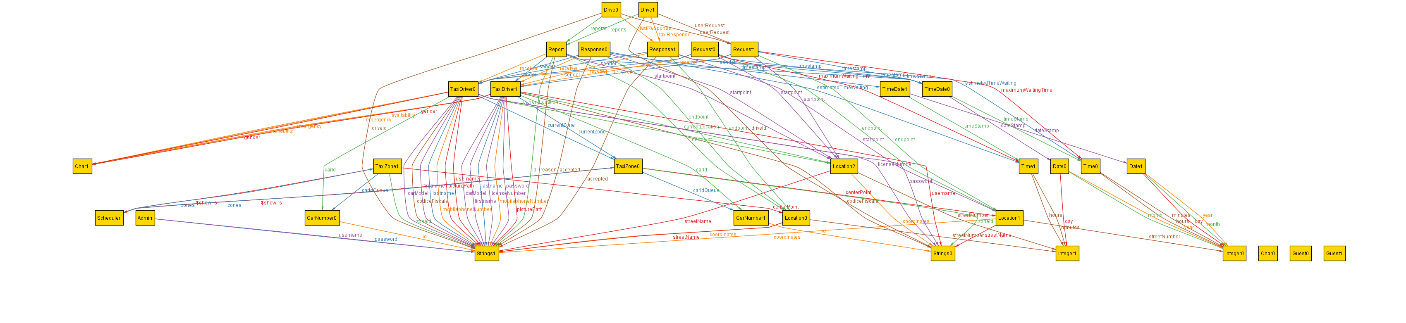
The screenshot of the Alloy Analizer software is presented.



As it can be seen, three counterexamples are found, as it was expected in asserts.

A trivially false assert will result in counterexamples being found when it is checked. Since anything makes such an assertion false, any example of the system will be a counterexample. As a result, the solution returned may appear completely unrelated to the assertion it falsifies.

Alloy software generates graphical view of the world generated, as it can be shown for command #12.

+

## Software and tools used

* Microsoft Office Word 2013: To create and redact this document
* StarUML [(http:// http://staruml.io/)](file:///D:\swe2\Software-Engineering-2-Project\Deliveries\(http:\%20http:\staruml.io\)): to create Use Cases Diagrams, Sequence Diagrams, Class Diagrams, State Machine Diagram and Activity Diagram
* Alloy Analyzer 4.2 [(http://alloy.mit.edu/alloy/)](http://alloy.mit.edu/alloy/): to prove the consistency of the presented model.
* NinjaMock [(https://ninjamock.com/)](http://balsamiq.com/products/mockups/): to create mockups for both mobile and web variants of the application.
* Github (reporsitory: https://github.com/penenadpi/Software-Engineering-2-Project/Deliveries)

## Hours of works

The time spent for constructing this document and Alloy model:

* Nenad Petrovic: ~ 52 hours.

# Revision

Version 1.3

1. Statechart diagram is added related to change of taxi driver availability taxi drivers in certain situations in order to describe it better.
2. Minor corrections in The World and the Machine part of the document – both the picture and page.
3. Maintainability – added a fact that price estimation could be changed, so the system should be modifiable in terms of pricing.
4. Goals 5 and 6 are updated with explicitly stated all the domain assumptions.
5. Design constraints and Scalability sections updated.
6. Several typos are corrected and missing words are added to several sentences related to goals.
7. Simpliefied sequence diagrams, more suitable for RASD documents are added. More complex variants from version 1.0 are going to be expanded for next phase and are exchanged with simpler variants.