MyTaxiService
Software Design Document
Version 1.1

Losio Davide Francesco, Luchetti Mauro, Mosca Paolo

January 20, 2016

Contents

1 — Introduction

This documentation will be used to aid in software development by providing further details of how the software should be built. Within the Software Design Document are narrative and graphical documentation of the project software design, including use case models, sequence diagrams, and other supporting requirement information.

1.1 Purpose

The purpose of the Software Design Document is to provide a description of MyTaxiService system design and architecture fully enough to allow software development to proceed with an understanding of what is to be built and how it is expected to be built. To achieve this DD(**D**esign **D**ocument) translates and states more accurately the Requirement Specifications described in the MyTaxyService RASD document. It identifies high-level system architecture and design framework as well as hardware, software, communication and interface components.

1.2 Scope

MyTaxiService application is a server/client combination that will allow a user to handle different type of taxi service, keeping track of all the transactions necessary for the completion of each operation. This will include book a taxi in a specific date, call a taxi, handle the sharing option and, for the taxi drivers, manage the taxi queues. All this functionalities will be guaranteed in the way and in the manner explained in the RASD document. Via a cross platform web environment (by the use of JavaEE), MyTaxiService will be able to run on various platforms, including Unix based OS and Windows systems, and all the portable devices based on Android, iOS and Windows phone. When a network connection to the server will be available, the user will be able to synchronize his PD (**P**ortable **D**evice) or PC with the server. The passenger-user will be allowed to log in or register and to make a request. Otherwise, the taxi-driver user will be allowed to set his availability and to accept or reject requests.

Below are stated some main issues that the system has to be capable to cope with:

- **PD Issues:** because of memory limitations, a PD will only store data and application parts that are strictly necessary for a PD user. Also, PDs have reduced screen size and limited input capability compared to PCs, so we will design PD standalone functionality in manner that can be easily displayed on a typical 240x320 screen.
- **Synchronization:** we will develop server software to work as an interface between the PC or PD and the application logic, by the re-using of already existent services offered by third parties company.
- **Transaction and functionalities:** all the transactions and booking procedure will be handled by the application logic layer. It will be divided from the presentation layer, as well as the queue managing features and algorithms. PDs will implement only presentation layer and connection functionalities.

The architecture will be developed and structured to support the fulfillment of this main issues.

1.3 Definitions, Acronyms, Abbreviations

- **RASD: R**equirements **S**pecification **A**nalysis **D**ocument
- **DD: D**esign **D**ocument
- **PD: P**ortable **D**evice
- **MVC: M**odel **C**ontrol **V**iew
- **FIFO: F**irst **I**n **F**irst **O**ut, it's a policy applied to the queue managing. It means that the first person to be enqueued will be the first to be dequeued
- **PAAS: P**latform **A**s **A** **S**ervice
- **SOA: S**ervice **O**riented **A**rchitecture
- **DB: D**ata**B**ase

1.4 References

- **MyTaxyService RASD**
  - November/6/2015, edited by DadOs & Green-Go & Pol Corporation -

1.5 Document Structure and Overview

- **Architectural Design:** this section is the main topic of the document. It provides an overview of the system major components and architectures, as well as architectural styles, used patterns and other design decisions.

  A detailed analysis of the modules will also describe lower-level classes, components and functions, as well as the interaction between them.

- **Algorithm design:** this section is focused on the definition of the most relevant algorithms of the project.
- **User interface design:** this section provides an overview on how the user interfaces of the system will be. In particular, referring on what is already stated in the RASD, here some further details are specified.
- **Requirements Traceability:** this section explains how the requirements defined in the RASD map into the design elements that are defined in this document.

2 — Architectural Design

2.1 Overview

This section analyze several different part of the design architecture:

- System Components & Interactions: here are individuated and stated the main system software components in which the software is split into, and in the interactions among them. To achieve this latter feature, a high level description of the interfaces will be provided. Within this description are:

  Component;

  Deployment;

  Runtime;

  UML diagrams. These are intended as a support to the better understanding, as well as a more clear and simply specification of our components division.

- Architectural Styles And Patterns here are listed the architectural styles and patterns used to solve the main interactions and functionality problems. They are taken in this first version of the DD more as suggestions and ideas. They will be updated and revised during the developing of the application where requested by the circumstances. If patterns and architectural styles will actually reveal to be unfeasible, different approaches will be evaluated and chosen.

- Technologies design decisions: here are stated choices and motivations taken regarding the development environment.

2.2 System Components & Interactions

The system to be produced implements the common MVC pattern. The components are split in order to represent this logic. For each component the main modules are listed.

2.2.1 Server Model

This module represents the core logic of the whole system. Components contained here are responsible for the correct performing of MyTaxiService features.

2.2.1.1 Queue Handler

This component has to guarantee the right usage of a FIFO (**F**irst **I**n **F**irst **O**ut) queue into the application. Keeping in mind that each zone has his own queue, the operations that this component handles are: add a new driver to the queue, delete a driver from the queue, extract the first driver in the queue and move a driver from top to the bottom of the queue. It has to follow some rules deduced from the RASD document:

- The driver has to be available to be added.
- A new driver has to be added in the right queue, that is related to the geographical location of the taxi.
- The deleting of a driver implies that he has accepted/reject a request or that he is become unavailable.

2.2.1.2 Requests Handler

This component is responsible of the correct forwarding of each request coming from a user. It is closely with the Queue Handler module because of his necessity to send requests to drivers. Furthermore, it embed some specific functionalities with respect to the request type:

- It has to forward the request with the sharing option enabled to the Shareable Ride Finder.
- It has to store the booking in a correct structure that provide to notify the system 10 minutes before the reservation time to perform the effective taxi allocation.

2.2.1.3 Shareable Ride Finder

This is a sub-component of the Request Handler and its aim is to list-out the rides which have the same origin and destination position of the analyzed one. The research is done upon a suitable memorization structure, in which are stated all the pending requests, either they are booking request or immediate request. The immediate request will be stored in that structure for at most three minutes. This is done in order to keep consistent their "immediate" peculiarity. At the end of this time-out, an immediate request with the shareable option enabled will be treated as a non shareable ride and the passenger will have to pay the entire fee.

2.2.1.4 Mapping Features Handler

This component has to deal mainly with the google maps API. On the passenger side, it has to support the origin and destination addresses input. It has to guarantee the correctness of these addresses and to show a graphical map representation in order to enhance the interactivity. Whereas on the taxi driver side, it has to deal with the queue manager by providing the driver positions. This allows the queue manager to assign the taxi to the right queue. Moreover it has to perform some navigation features. It has to provide the taxi driver with the available paths, distance and travel time to arrive at the passenger location.

This can support the accept/reject decision of a request by a taxi driver. In case of acceptance the mapping handler has to connect the driver into his GPS navigation system by the opening of the related view. Here, the shortest path passing through the passenger location and going to his final destination will be displayed.

2.2.1.5 DBMS and Log-In Handler

This component has to deal with the user's personal data. Since the only data needed by the application are the Log-In data related with the personal data, this system also has to handle the log-in procedure. It has to record in the DB the registration information and to query the log-in information provided by the user, finally it has to check if there is a correct match. To perform the log-in procedure with regards of the requirements stated in RASD, this component has to check also the ban state of the user. In this case it has to block the user functionalities.

2.2.1.6 Ban Handler

This component is responsible for the managing of the ban policies. Its task is to store correctly the banned users in a suitable memory structure. It has to return the ban state of a user when another component query it and it also has to periodically check if a banned user has done his "sentence". In this case, the Ban Handler has to restore all the user functionalities and return the not-banned information to the user view.

2.2.2 Server Controller

This module represents the part of the system that is directly in contact with the users. It acts as a glue between the view part, loaded on the user's devices, and the model part, that is loaded on the server. This part too has to be loaded and performed by a dedicated server area.

2.2.2.1 Passenger Communicator

This component is responsible for the correct forwarding information between the server and a user. The operations that this module handles are: receive a message from a user, send a message to a user, handle timeout errors.

2.2.2.2 Driver Communicator

This component is responsible for the correct forwarding information between the server and a driver. The operations that this module handles are: receive a message from a driver (including the availability state and the GPS positioning), send a message to a driver, handle timeout errors.

2.2.3 Views

This module is a collection of graphical views related to the suitable devices and types of user. For graphical views is meant the graphical representation of both the user interface and the data computed by the model. This module has to manage the different input possibilities and to represent the results for the elaboration of them. In any case, the adoption of cross platform technologies allows to develop one unique system interface that will be automatically adapted to a suitable format for both WEB application and mobile application. Several useful mockups that picture a complete framework of all the functionalities requested by the user interfaces are stated in the RASD document. Because of their completeness and clearness they are reasonably assumed as an adequate support to the views development.

Mainly there are two different views to be developed, according to the user types:

2.2.3.1 Passenger-views

This views have to provide an easy access to all and only the passenger functionalities, it has to be mainly structured for the mobile applications, because we can reasonably assume that the most of the users will access to MyTaxiService with this interface. All the functionalities are already sufficiently stated in the RASD.

2.2.3.2 Driver-views

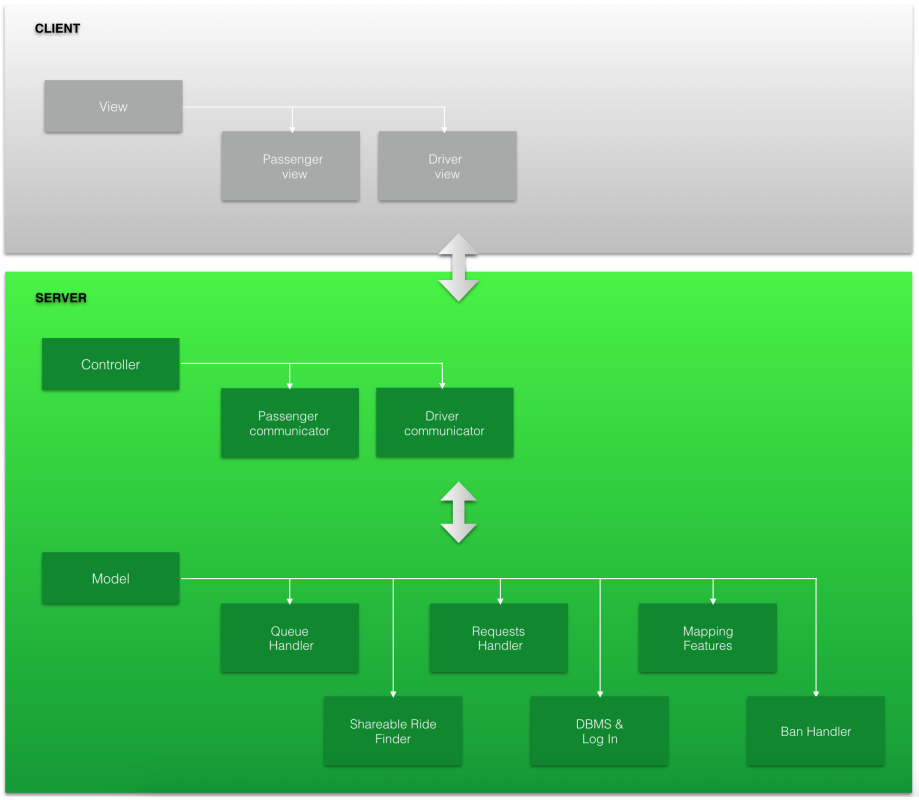
This views have to provide an easy access to all and only the taxi driver functionalities. The same consideration made upon the passenger-view are also valid here.

2.2.3.3 "Yeah, but how to choose which ones!?"

During the log-in procedure the the DBMS and Log-In Handler component performs another little useful feature among the others. When it successfully founds a match of the user credential, it also return the type of the user. This output, taken as input by the view, allows the loading of the right view associated to the passenger type.

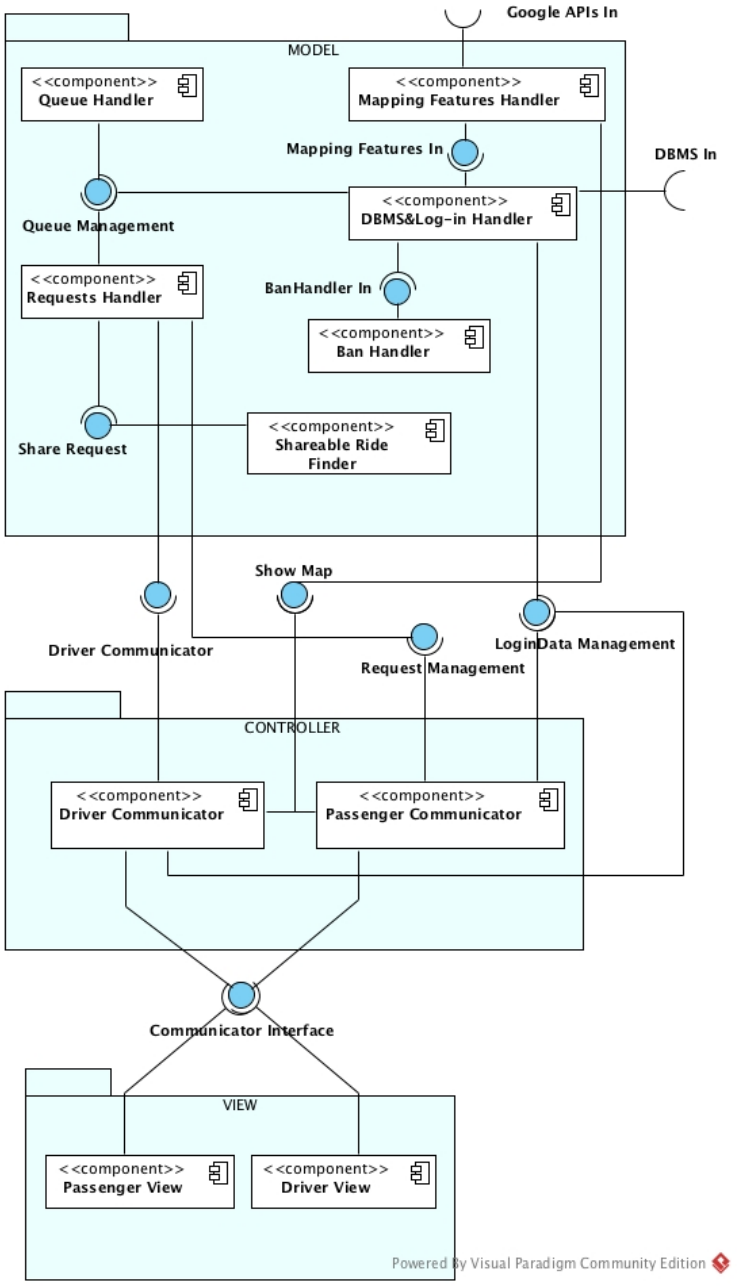
2.3 Components scheme

To have a better overview of the various modules with the relevant components and their connections, it was decided to provide a general outline of the system



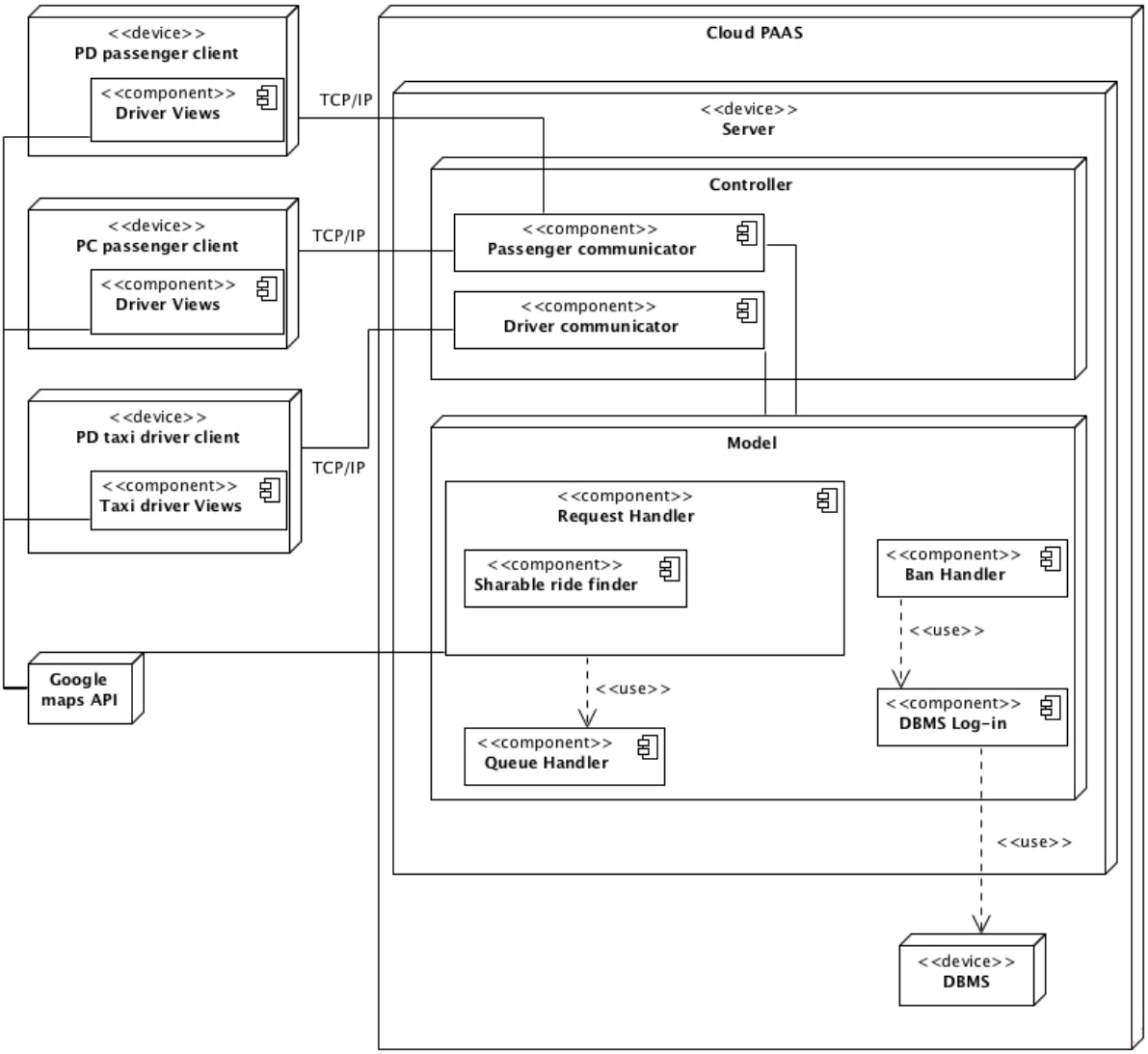
2.4 Component View

This diagram shows how components are wired together to form the software system.



2.5 Deployment View

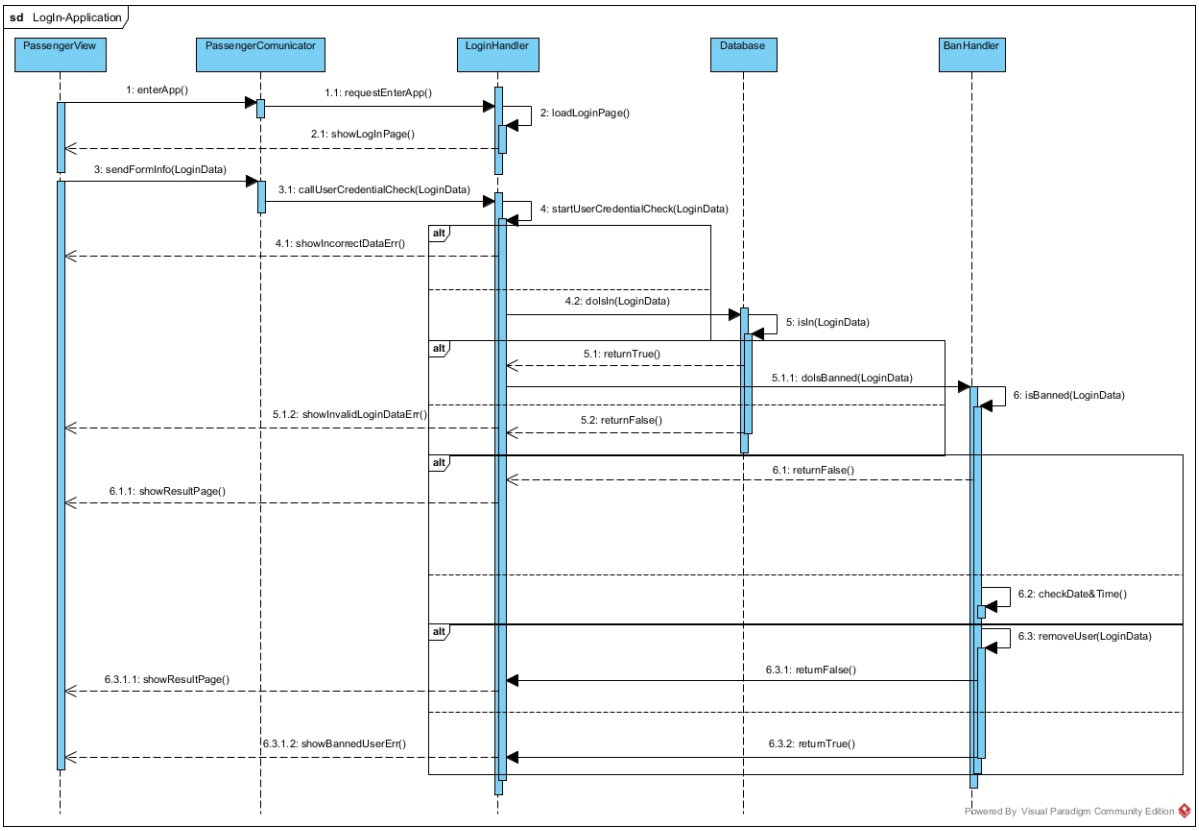
This diagram shows what hardware components exist, what software components run on each node and how the different pieces are connected.



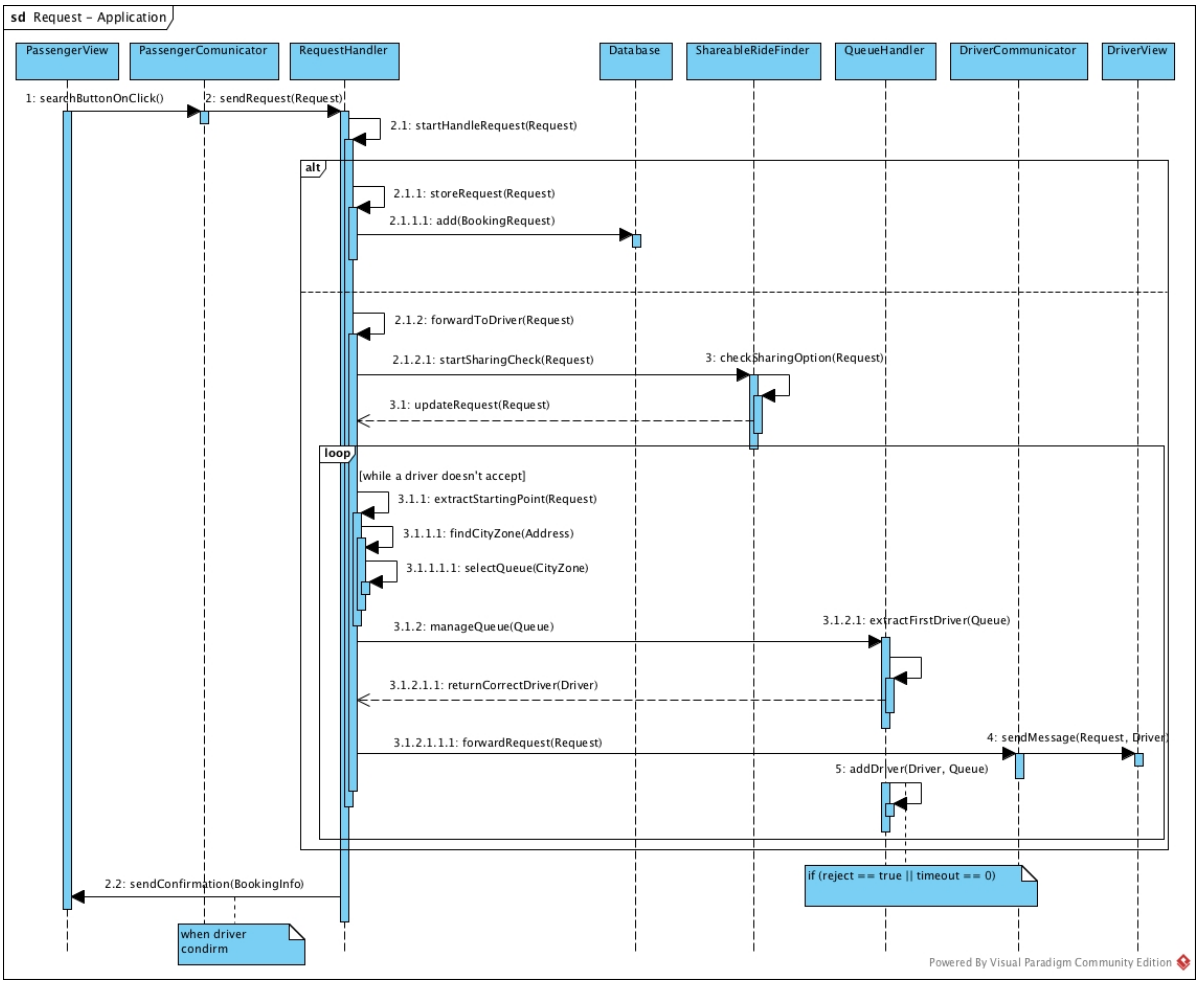
2.6 Run-time View

This section shows the most important features of the system through some sequence diagrams. The interaction between the various component uses methods that will be introduced later on the document.

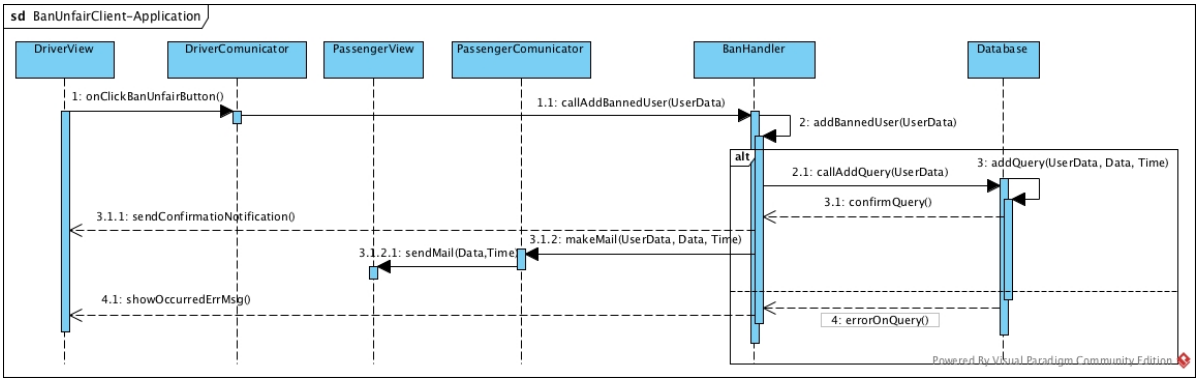
2.6.1 Log In



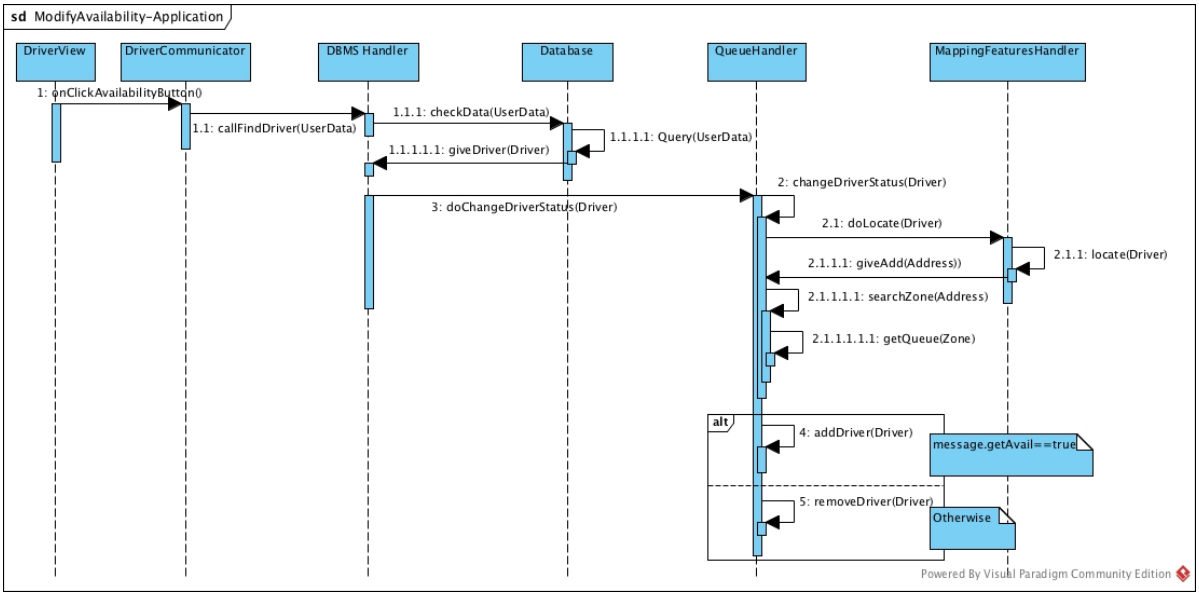
2.6.2 Taxi Request



2.6.3 Ban Client



2.6.4 Modify Availability



2.7 Component Interfaces

Here are stated the main functionalities provided by the component interfaces. For each one is stated the signature (in a java like pseudo-code) and a little explanation of how it works.

2.7.1 Queue Handler

```
1  public void addDriver(Driver d, Queue q)
```

input: It receives a driver and a queue as input.

output: -

what it does: Adds a driver at the end of the queue.

```
1  public Driver extractFirstDriver(Queue queue)
```

input: It receives the queue as input.

output: It returns the first driver of the queue.

what it does: Selects the first driver from the queue, removing and returning him.

```
1  public void removeDriver(Driver d, Queue queue)
```

input: It receives a driver and a queue as input.

output: -

what it does: Removes a specific driver in the queue. This method is used when a driver changes his availability.

2.7.2 Request Handler

```
1  public void handleRequest(Request request)
```

input: It receives the request as input.

output: -

what it does: The method implementation has to perform the correct handling of the various request type. It has to forward the immediate request to a Taxi Driver, accordingly to the constraints stated for this operation in the previous analysis. Otherwise it has to store the booking request in a suitable memory structure.

```
1  public Boolean deleteRequest(Request request, Queue queue)
```

input: It receives the request and the queue to which is addressed as input.

output: It returns a boolean values that state the success or failure of the operation.

what it does: Deletes the stored request after a feasibility checks of the operation according do deleting operation constraints stated in RASD.

2.7.3 Shareable Ride Finder

```
1  public Request sharableCheck(Request request)
```

input: It receives the request as input.

output: It returns a new request object containing the overall number of passenger joining the ride, as the source and destination and all the other information are the same.

what it does: -

2.7.4 Mapping Feature Handler

This component is totally self contained, as it is a third parties service, so all the interfaces to interact with it are already provided by the Google Maps API documentation stated below:

- https://developers.google.com/maps/

2.7.5 DBMS and Log-In Handler

In order to ensure an high level security for this sensible component, it will be designed having an unidirectional communication. This in the sense that the only sensible data are sent directly by the end-user towards our system and the system can only answer with boolean values or not strictly related personal data. Acting in this way, the database system never directly sends personal data on the network by itself.

```
1  public boolean addNewUser(UserData uData)
```

input: It receives the users data as input.

output: It returns a boolean value that state the success or the failure of the adding operation.

what it does: -

```
1  public boolean userCredentialCheck(LoginData lData)
```

input: It receives the users data as input.

output: It returns a first boolean value that state the success or the failure of the login operation.

what it does: -

2.7.6 Ban Handler

```
1  public boolean addBannedUser(LoginData lData)
```

input: It receives the users data as input.

output: It returns a boolean that state the success or the failure of the operation.

what it does: It correctly stores the banned user in a suitable memory structure.

```
1  public boolean isBanned(LoginData lData)
```

input: It receives the users data as input.

output: It returns a boolean value that state if the user is actually banned.

what it does: -

2.7.7 Passenger and Driver Communicators

Actually in all the modern developing technologies controllers are already ensured by the developers tools and frameworks themselves through the use of listeners class. In any case listeners class can be overwrite in any moment forecasting some developing difficulties.

2.7.8 Views

Views have to be capable to receive the model output, and update the graphics feedback of the application. Due to this, view interface has only to provide suitable methods to receive the messages sent by the Model.

2.8 Architectural Styles and Patterns

Here are listed the main architectural styles and patterns choices. Every choice is adequately motivate and is made in order to simplify the design of intricate situations.

2.8.1 SOFTWARE

2.8.1.1 MVC Pattern

MVC is an essential, rock solid choice for each distributed architecture. The pattern is perfectly suitable with the tier architecture model and represent the “best practice” choice since many years. Furthermore, every modern development environment automatically embedded this structure itself, so, in any case, you cannot help avoiding his use.

2.8.1.2 Event Based Pattern

Event Based Pattern is implemented in the managing of the request. The passenger-user could be seen as a publisher in the publish-subscribe paradigm, and the taxi driver as the subscriber. In this pattern, a request plays the role of the event, it is up to the system to ensure that actually not any taxi-driver subscriber can see the request, but only the first taxi driver in the right queue. We made this choice because firstly, the event base pattern, is another well known, reliable pattern. Secondly, in front of some little modifications in his implementation, it can greatly do his job.

2.8.1.3 SOA Architecture mainly

The System adopts a proper service oriented architecture, either to use or offer services.

As stated in the RASD document, the whole "map system" is based upon the Google Map API, so for the architecture is mandatory to allow the useful implementation of this features. As decided with our stakeholders, we are going to release our own application programmatic interfaces. This will provide third party companies with our module, knowledge and algorithm carefully studied on this application case.

Follow the list of the API to be released:

- Queue Handler API,
- Shareable ride finder API,
- Request Handler API.

About the major issues resulting from the adoption of this architecture:

- Security is guaranteed by a careful separation between the sensible data stored and managed only by the DBMS module and the data used by the services.
- The optimal reliability of the message exchanged among the various services shall be guaranteed by the handling of the sending/receiving failure.
- Quality of service, with regards of the service adopted, is guaranteed by the choice of well known, widely used and reliable services. Whereas the quality of services released by us, is guaranteed by a careful planning and the adoption of all the necessary precaution.
- Service maintainability will be facilitated by the simplicity of the services offered.

2.8.2 HARDWARE

The Hardware Architecture will use the latest technology in the field of Cloud Computing. In particular it will be adopted the PAAS model.

The Platform as a service (PAAS) adoption will provide a platform allowing us to develop, run and manage web applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching of an app. So our company will controls software deployment and configuration settings, and the provider will give the networks, servers, storage and all the necessary services to host the application. This translates into:

- Significant cost reduction.
- Significant developing time reduction.

Obviously with respect to the other architecture solution allowable (Hardware infrastructure of our own, IAAS cloud architecture etc). The implementation of a large hardware structure such as two tier o three tier structure indeed, includes huge and useless effort both to buy the necessary equipment and to set them up. Furthermore there are the maintenance and personal cost to take in account. Nowadays several different companies definitely offers a PAAS cloud system, capable of offer good and efficient performance for the right price.

2.9 Technology design decisions

Here are explained the motivations that bring us to chose JavaEE. Mainly, because it helps in create large scale, multiered, scalable, reliable and secure networks application. This is the case of MTS. It substantially enables developers to focus on business logic by reducing development time, introducing convention over configuration, allowing the use of powerful annotations. It alse definitely reduces application complexity thorough the use and implementation of multitiered model based on the containers/components paradigm to model functionalities and their interactions. It follows the philosophy "Write Once Run Anywhere", and defines a contract that makes possible to use platforms from various vendors introducing a huge flexibility. Developers are allowed to abstract from low level problems such as transaction management, multi-threading, connection and pool management. It provides mechanisms to support interoperability with non-Java systems, and this is an essential characteristic since we have to operate with several different systems either towards the user side or towards the business logic side. We will substantially make use of JSF technologies for the web client application, and ad-hoc applets will be implemented to allowing a user-friendly eye-candy GUI for the mobile-device.

3 — Algorithm Design

The goal of this section is to provide the most relevant algorithmic part of the project. In order to do this some code is stated relating to the most important modules analyzed in the chapter 2. Moreover the code is not complete. It can be considered as a first implementation which aims to give a general idea of the system core logic. The code is written in a Java-like pseudocode.

3.1 Login

This algorithm receives the user data inserted in the login form as input.

1. The system receives the login data
2. The system calls the userCredentialCheck procedure in the Login Handler, using the login data as a parameter
3. The userCredentialCheck procedure checks the correctness of the data
4. If the given data are wrong a related error is shown and the procedure returns the value “false”
5. If the given data are correct the procedure searches for a correspondence in the DB
6. if the correspondence is not found between id and password, a related error is shown and the procedure returns the value “false”
7. If the correspondence is found the procedure calls the Ban Handler and checks the ban status of the user
8. If the user is banned a related error is shown and the procedure returns the value “false”
9. If the user is not banned the procedure returns the value “true”

A possible implementation of the algorithm is the following:

```
1  public static boolean userCredentialCheck(LoginData lData){
2    if(isCorrect(lData)){
3      if(!DataBase.isIn(lData))
4        return false; //"id or password wrong" message must be sent
5      else{
6        if(BanHandler.isBanned(lData))
7          return false; //banned info must be sent to the user
8        else
9          return true;
10     }
11   }
12   return false; //incorrect data message must be sent
13 }
14
15 private boolean isCorrect(LoginData lData){
16   //list of data constraints
17   if(lData.getId().contains('@') &&
18      lData.getId().contains('.') &&
19      lData.getPsw().contains('[A..Z]')) &&
20      lData.getPsw().contains('[0..9]')))
21     return true;
22   else
23     return false;
24 }
```

3.2 Ban Management

The algorithms stated below are responsible of adding a user in the Banned User set and of checking for a user ban status.

- **Add an user to the Banned User Set**

  1. The system calls the addBannedUser procedure using the user data as a parameter
  2. The procedure creates a new element in the ban set
  3. If error occurs the procedure returns the value “false”
  4. The procedure adds the user data in the created element
  5. The procedure adds current data and time in the created element
  6. The procedure returns the value “true”

A possible implementation of the algorithm is the following:

```
1  public boolean addBannedUser(UserData uData){
2    try{
3      BannedUser newBannedUser = new BannedUser(uData);
4      bannedUserSet.add(newBannedUser);
5    } catch(InstanceError e) {
6      return false;
7    }
8    newBannedUser.addData('currentData');
9    newBannedUser.addTime('currentTime');
10   return true;
11 }
```

- **Check for ban status**

  1. The system calls the isBanned procedure using the user data as a parameter
  2. The procedure checks for a correspondence in the banned user set
  3. If it is not found the procedure return the value “false”
  4. If it is found the procedure compares the data and time of the ban with the current ones
  5. If the ban period is out the procedure remove the user from the set and return the value “false”
  6. Else the procedure return the value “true”

A possible implementation of the algorithm is the following:

```
1  public boolean isBanned(UserData uData){
2    for(BannedUser user: bannedUserSet){
3      if(uData.getId() == user.getId()){
4        //if the ban period is out
5        if('currentData' - user.getData() >= MAX_BAN_DAYS){
6          bannedUserSet.remove(user);
7          return false;
8        }
9        else
10         return true; //the user is still banned
11     }
12   }
13   return false; //the user is not banned
14 }
```

3.3 Request Management

This algorithm, starting from an user request, shows how this can be handled by the system.

1. The system loads the Request Handler
2. The system calls the Handle Request procedure of the Request Handler, passing the request as a parameter
3. The method, relying on the type of request, decides how to manage it by calling the specific procedures
4. If the request is a "Booking request" the system properly stores it in the database
5. If the request is a "Taxi now request" the system checks for the sharing option status
6. If the sharing option is "true" the system starts the Shared Request Matcher Algorithm
7. The system sends the request to the first driver available. In order to do this, the following procedure is done:
    - Extract the starting point from the request
    - Find the correct city zone associated to the starting point address
    - Extract the queue associated to the city zone
    - Extract the driver at the head of the queue
    - Forward the request to the driver

A possible implementation of the methods that can handle the procedures stated above is the following:

```
1  //handle a generic request
2  public void handleRequest(Request request){
3    if (request instanceof TaxiNowRequest)
4      forwardToDriver(request);
5    else
6      storeRequest((BookingRequest) request);
7  }
8
9  //add a booking request to the catalog
10 private void storeRequest(BookingRequest request) {
11   requestsCatalog.add(request);
12 }
13
14 //send the request to first queued driver in the corresponding
     city zone
15 private void forwardToDriver (Request request){
16   Request fwRequest = checkSharingOption(request);
17   for (CityZone zone: City.getCity()){
18     if (zone.containsAddress(fwRequest.getStartingPoint())){
                       QueueHandler.selectFirstDriver(zone.getQueue
   ()).sendMessage(fwRequest);
19     }
20   }
21 }
22
23 //check for sharing option and return an handled request
24 private Request checkSharingOption (Request request){
25   if (request.isShareable())
26     return ShareableRideFinder.generateShareableRequest(request);
27 }
```

3.4 Queue Management

The input for this algorithm is a Queue element on which the operations are executed.

1. The system finds the correct queue using the Request Handler component
2. The system extracts the first driver in the queue. The extraction must be done in order to guarantee that two or more incoming requests are forwarded to distinct drivers.
3. The system forwards the request
4. The system starts a timeout and waits for the driver answer
5. If the time is up the driver must be re-added to the queue
6. Else, if the driver answer "accept", the system sets his status to unavailable, sends him all the informations such as the route and forwards a confirmation to the user
7. Else, if the driver answer "reject", the system re-added him to the queue

A possible implementation of the methods that can handle the procedures stated above is the following:

```
1  //add a driver at the end of the queue
2  public void addDriver(Driver d, Queue queue){
3    queue.add(d);
4  }
5
6  //extract a driver
7  public Driver extractFirstDriver(Queue queue){
8    if (!queue.isEmpty())
9      Driver d = queue.getFirst();
10     queue.removeFirst();
11     return d;
12   else
13     return null; //here an error message can be sent
14 }
15
16 //remove a specific driver
17 public void removeDriver(Driver d, Queue queue){
18   for (Driver driverToDelete: queue){
19     if (d == driverToDelete)
20       queue.remove(d);
21   }
22 }
```

3.5 Driver Status Management

This algorithm, starting from a message received by a driver view, has to manage a change in the driver status.

1. The system receives a message from a driver view through the driver communicator
2. The system, relying on the message, found the driver in the DB
3. The system calls the changeDriverStatus procedure using the driver as a parameter
4. The system extracts the position of the driver using the Mapping features handler
5. The systems finds the correct city zone associated to the position address
6. The system extracts the queue associated to the city zone
7. The system extract the availability information from the message
8. If the information is "available" the system has to add the driver using the Queue Handler component
9. If the information is "unavailable" the system has to delete the driver from the queue

A possible implementation of the algorithm is the following:

```
1  public void changeDriverStatus(Driver d){
2    Address address = MappingFeaturesHandler.locate(d);
3    for (CityZone zone: City.getCity()){
4      if (zone.containsAddress(address)){
5        Queue queue = zone.getQueue();
6      }
7    }
8    if (message.getAvailability() == 'true')
9      QueueHandler.addDriver(d, queue);
10   else
11     QueueHandler.removeDriver(d, queue);
12 }
```

3.6 Shared Request Matcher

The input of this algorithm is a Request element. The goal is to merge shareable requests into one.

1. The system loads the Shareable Ride Finder component and use the request as a parameter
2. The Shareable Ride Finder checks if there's a pending request whit the same starting point and the same destination.
3. If some request is found, the number of passenger is updated in a new request
4. The new request is returned

A possible implementation of the algorithm is the following:

```
1  //generate a new request merging the shareable ones
2  public Request generateShareableRequest(Request request){
3    for (Request r: pendingRequests){
4      if (canBeMerged(request,r) && r.isShereable()){
5        request.psgNum() += r.psgNum();
6      }
7    }
8  return request;
9  }
10
11 private boolean canBeMerged(Request r1, Request r2){
12   if (r1.getStartingPoint() == r2.getStartingPoint()
13       && r1.getDestination() == r2.getDestibnation())
14     return true;
15   return false;
16 }
```

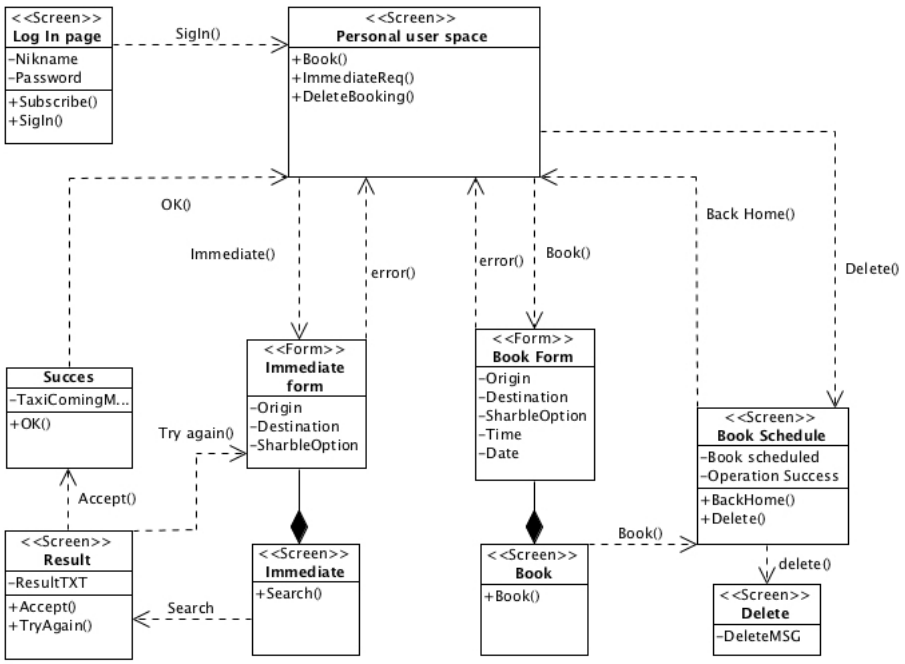
4 — User interface design

This part is already stated in the RASD document and is only reported here for the sake of completeness, and to have a reference. You can find this part in:

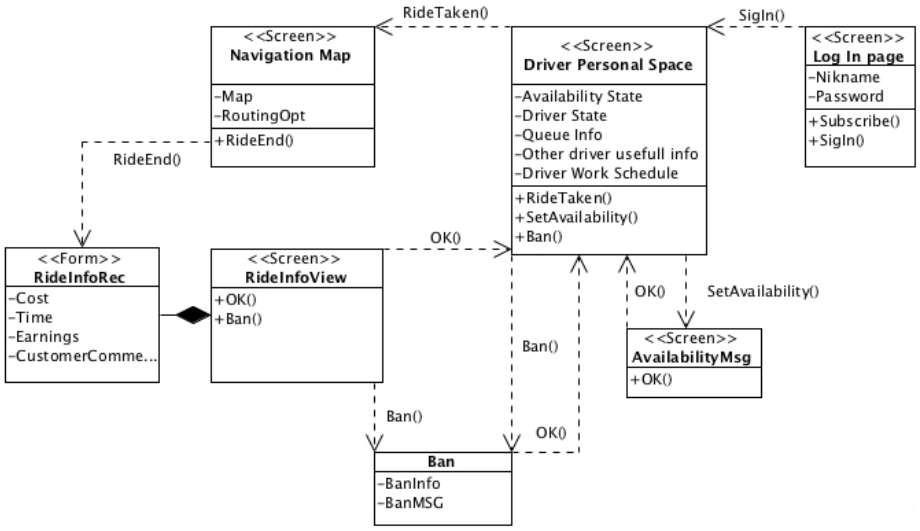
- RASD - Chapter 3.1.1, pages 8-15

Instead of producing some other mockups we think that report a briefly and simplified UX Diagram representation can be more useful to the development team. These diagrams clearly specify the transition between the views and their sequences. Below are stated the main UX Diagrams related to the main functionalities, leaving other more common view sequence (like the log-in ones) to the developer own interpretation. There is no reason to re-design them because many models and more suitable best practices are already available and commonly used.

4.1 Passenger UX Diagram



4.2 Driver UX Diagram



5 — Requirements Traceability

It is explained here how the requirements defined in the RASD map into the design elements defined in this document.

5.1 Passenger User Requirements

• Taxi now/Book a taxi:

R1 The system shall check at eventual user's ban before allowing the user to do reservations.

- This is ensured By the Ban Handler in collaboration with the Log-In Handler component. The Ban is checked at the log-in moment. If a user is banned the system automatically blocks all the user functionalities.

R2 The system shall contact the first queued taxi driver in the specific city-zone requested by the user.

- This is performed by the Queue Handler and the Request Handler. The Request Handler will contact a Queue Handler, which takes care of forward the request, to the first driver in the zone queue.

R3 The system shall create a private session for each reservation, situation with concurrent requests for the same driver must be avoided.

- This is handled by the Queue Handler. The driver is moved out the queue in the same time it receives a request. This is acceptable because if the driver either accept or reject the request it has to be moved out the queue.

R4 The system shall grant the correct priority in queuing taxis. A driver who does not accept a reservation request must be moved in the bottom of the queue.

- This is handled by the Queue Handler. The correctness of the operation performed by the handler is ensured by the queue structure of the memory structure adopted.

R5 The system shall avoid wrong inputs such as same start and destination address.

- Checks are performed by the Request Handler in collaboration with the mapping system, during the handling of the request.

R6 The system shall notify to the requesting user that a taxi driver has been found.

- The outcome of the Request Handler in the model, as well as all the model outcomes, generates a view change, in this case a pop-up. This is ensured and embedded in MVC pattern adopted.

R7 The system shall notify to the requesting user the waiting time of the first taxi, the fee and the taxi code number.

- The outcome of the Request Handler in the model, as well as all the model outcomes, generates a view change, in this case a pop-up. This is ensured and embedded in MVC pattern adopted.

R8 The system shall guarantee that a reservation has to occur at least two hours before the ride.

- This kind of checks are performed by the Request Handler class.

R9 The system shall guarantee that each request corresponds an answer message which also brings a taxi confirmation.

- This is ensured by the logical order in which the procedure is performed.

R10 The system shall give a confirmation about the booking taken in charge. inputs starting and ending position; date; time; sharing option. outputs booking acknowledged or denied

- The outcome of the Request Handler in the model, as well as all the model outcomes, generates a view change, in this case a pop-up. This is ensured and embedded in MVC pattern adopted.

- **Taxi sharing:**

R1 The system shall check if another user has already reserved a taxi for the same destination with the share option activated. If this happens, the best route is calculated and sent to the driver.

- This feature is performed by the Shareable Ride Finder component. Whereas route calculating is performed by the Mapping Features Component.

• **Reservation deleting**:

R1 The system shall grant the possibility to delete a reservation for a user. This must be true only two hours before, whereupon this possibility is no longer available.

- This functionalities is performed by the related Queue Handler method.

5.2 Taxi Driver requirements

• **Accept/reject a request:**

R1 The system shall grant the access to the driver GUI only for those accounts that have the correct credentials.

- This checks is performed by the Log-In Component.

R2 The system shall control that all the drivers waiting in the queue are eventually available.

- Since the Queue Handler add only available driver, the requirements is implicitly fulfilled.

R3 The system shall grant that only one driver at a time have access to a reservation request.

- Since the Queue Handler forward a request only to the first driver in queue, the requirements is implicitly fulfilled.

R4 The system shall notify at the first queued driver the request.

- Already stated above, in R3 requirements.

R5 The system shall update the taxi driver user's ride schedule for each confirmation sent.

- The functionality is performed by the Mapping Features Handler component, in collaboration with the Request Handler.

• **Change of the availability state**:

R1 The system shall provide a "not available" option that allows taxi drivers to exit from the waiting queue in case they pick up a person without using the service.

- Functionalities is provided by the taxi driver view, as stated in RASD in chapter 3.1.1, inside the user interface description and mockups. The use of this functionalities apply a change performed by the Queue Handler and in particular by his Remove/add Driver in queue related methods.

R2 The system shall automatically switch to "not available" a driver who takes charge of a reservation. This driver is removed from the queue.

- This functionalities are ensured by the Queue Handler and The Request Handler.

R3 The system shall automatically switch to "not available" a driver who is not enqueued and switch to "available" an enqueued one.

- This functionalities are embedded in the Queue Handler procedure and methods.

• Ban for unfair clients:

R1 The system shall provide an "unfair client" option that allows taxi drivers to communicate an inappropriate client behavior (e.g. false reservation).

- Functionalities are provided by the taxi driver view, as stated in RASD in chapter 3.1.1, inside the user interface description and mockups. Therefore in the model is performed by the Ban Handler related methods.

R2 The system shall recognize a banned client and prevents any reservation for a limited chosen time (e.g. 1 week).

- Functionalities are performed by the Ban Handler related methods.

R3 The system shall notify to a banned client his condition when he receives a ban.

- This is ensured and embedded in MVC pattern adopted.

R4 The system shall display "ban information", such as time left, for banned clients that logged-in.

- This is performed by a suitable view, as stated in RASD in chapter 3.1.

R5 The system shall notify to a banned client that he is no longer banned when ban time is over.

- This is ensured and embedded in MVC pattern adopted as the outcome of Ban handler periodically check procedure. A suitable view, as stated in RASD in chapter 3.1, is related to the representation of this features.

• Queue Managing:

This feature are strictly embedded in the realization of the Queue Handler component and are not further analyzed here.

6 — Appendix

6.1 Used Tools

We used the following tools to make the DD document:

- LYX 2.1: to redact and format the document;
- Visual Paradigm 10 Community edition: to draw the Use Cases Diagram and the Sequence Diagrams.

6.2 Time Spent

- For redact, correct and review this Document we spent almost 16 hours per person.

7 — Revisions

7.1 Version 1.1

In this revision the following elements have been updated:

- Component View: several changes in the interaction between components.
- Technology design decisions: different choices about this point have been preferred.