Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

M.Sc. in Computer Science and Engineering

myTaxiService

Software Design Document

Authors:

Angelo Gallarello

Edoardo Longo

Giacomo Locci

December 20, 2015

Version 1.1

Contents

List of Figures

Abstract

The current document represents the *Software Design Document* (SDD) of *myTaxiService* system. It provides a representation of the system's framework, based upon the description of the system illustrated in the *Requirement and Analysis Specification Document* (RASD). Throughout the document a comprehensive study is conducted, providing many different views of the architecture, in order to offer as many perspectives as possible. This is to ensure that any detail of the architecture is thoroughly analyzed.

The main stakeholders of this document are the teams in charge of the implementation and the software maintenance, for this reason a wide gamut of diagrams is offered, ranging from *class diagrams* to *ER diagrams* and *architecture diagrams*.

1 Overview

myTaxiService is a taxi service that will operate in a big city; the main purpose is to simplify the access of passengers to the service and to guarantee a fair management of the taxi queues.

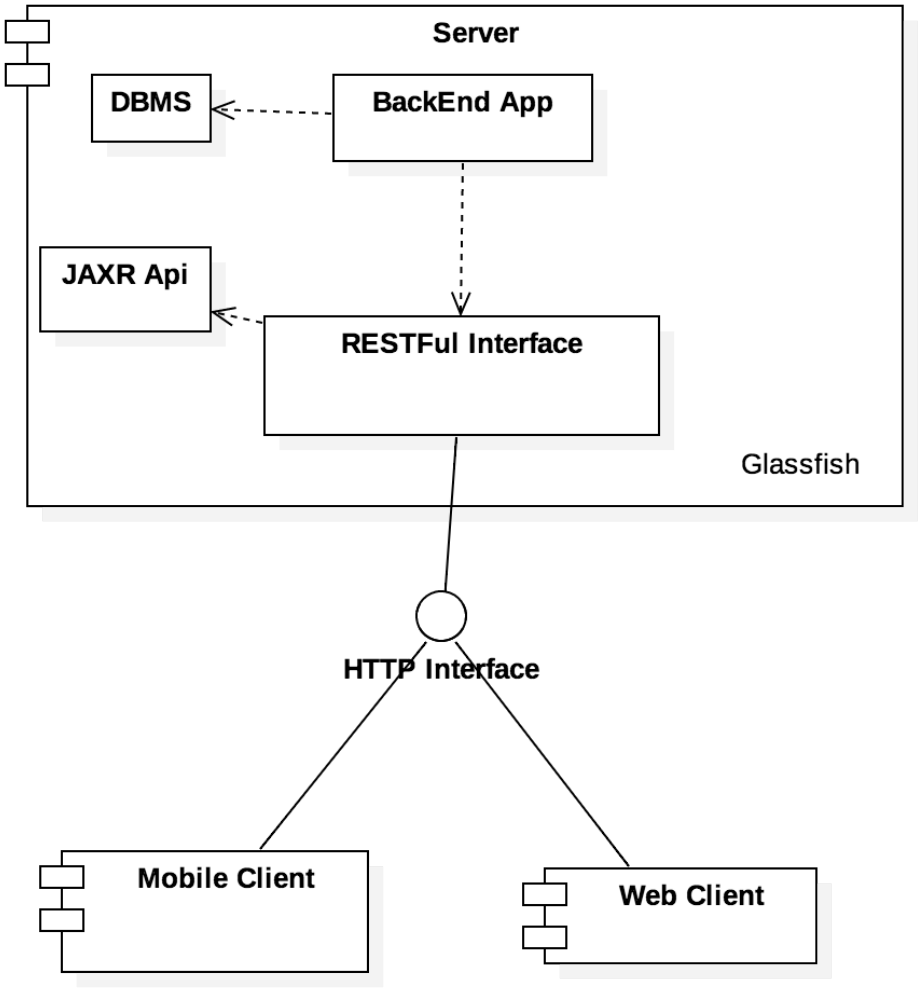
The main stakeholders of the system are the *Users*, the *Taxi Drivers* and the *Operators* as highlighted in *section 1.3* of the *RASD*.

The system is composed of four main core applications :

- Mobile Application (User)
- Web Application
- Mobile Application (Taxi Driver)
- Back-End Application

as stated in *section 1.2.* of the *RASD*.
It's important to highlight that in this document the design of mobile application is based on the Android platform.
Below is shown a diagram representing a general overview of the system.

Figure 1: Overview Diagram



2 High Level Components

The system could be divided into three main high level components that do not necessarily correspond only to one real application:

Server

The Server component is the kernel of the service we want to provide, it incorporates most of the *business logic*, it stores most of the *data* and it provides programmatic interfaces to the clients.

User Client

The User Client components is an high level representation of the real clients available to the users of our service. It's modeled as a *thin client* and it relies on the *Server* to fulfill its tasks.

Taxi Driver Client

The Taxi Driver Client component is an high level representation of the real clients available to the taxi drivers registered to the service. It's modeled as a *thin client* and it relies on the *Server* to fulfill its tasks.

2.1 Components Interaction

From a high level perspective the system is designed following the well known *client-server* paradigm.
The interaction between the components is handled by the Server that provides a programmatic interface that is able to receive remote call from the clients.
The clients never communicate directly with one another.

3 Component View

This section highlights the main features and roles of every component of the system. Moreover it describes the internal interfaces between different classes of every component.
External interfaces between components are described in *section 6*.

3.1 Server

The Server is composed of:

Back-End Application

As stated in *section 1.2.2* of the *RASD*, the *Back-End Application* is the system component that handles most of the business logic.
The application is written in *Java EE* and to fulfill its tasks (see *section 3.5.3* of the *RASD*) it needs to interface with the Internet network using the *HTTPS protocol* and the *JAVA API for RESTful Web Service*[1], with a *MySQL database* and with external Google Maps API.

Back-End Internal Interfaces

The *Back-End Application* is built to be very modular and to grant *interchangeability* between components.
There are four main classes that constitute the kernel of the application :

- **QueueManager** Handles queue policies.
- **RideManager** Creates and manage rides. Is connected to the *RequestManager* via the *RideManagerInterface* and directly depends on *QueueManager*
- **ActorManager** Create and update data about users an taxi drivers. Is connected to the *RequestManager* via the *ActorManagerInterface*
- **PositionManager** Update taxi drivers position. Is connected to the *RequestManager* via the *PositionManagerInterface* and directly depends on *QueueManager*
- **RequestManager** Get and build *Request* object from the requests received via *HTTP*

[1] See `https://jax-rs-spec.java.net/`

MySQL Database

The MySQL database fulfills the task off storing and granting access to all the data generated and used by the service.
A *database dump* is performed daily during the period of minor activity of the service [2].
The connection between the *Java EE* application and the databased is supported by the *JDBC connector*[3].

3.2 User Client

Different real clients are available to the end users of the system.
As stated in *section 1.2.2* of the *RASD* a native mobile application is developed for Android, iOS, Blackberry and WP.
Moreover a Web Application is also available.
To fulfill the requirements expressed in *section 3.5.1* and *section 3.5.2* of the *RASD*, all the clients need to communicate with the Server making calls to the REST API using platform specific API for REST HTTP calls.

3.3 Taxi Driver Client

Different real clients are available to the taxi driver registered to *myTaxiService*.
As stated in *section 1.2.2* of the *RASD* a native mobile application is developed for Android, iOS, Blackberry and WP.
To fulfill the requirements expressed in *section 3.5.1* and *section 3.5.2* of the *RASD*, all the clients need to communicate with the Server making calls to the REST API using platform specific API for REST HTTP calls.

3.4 Clients Internal Interfaces

Mobile clients are composed mainly of subclass of platform specific components.
Interfaces between components are therefore specified in the SDK of each platform. However it's important to highlight that every mobile application has to interface with a *Network Component* that handles *HTTP* requests.

[2] At first, when no activity data is available, the dump will be performed at 04:00 A.M
[3] See `http://dev.mysql.com/downloads/connector/j/`

Figure 2: BackEnd Class Diagram

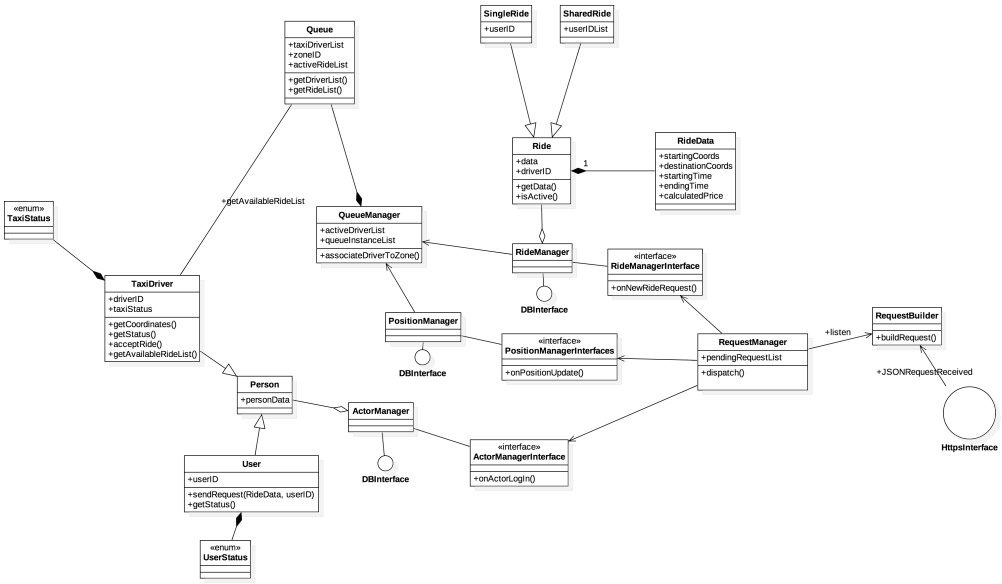


Figure 3: User App Class Diagram

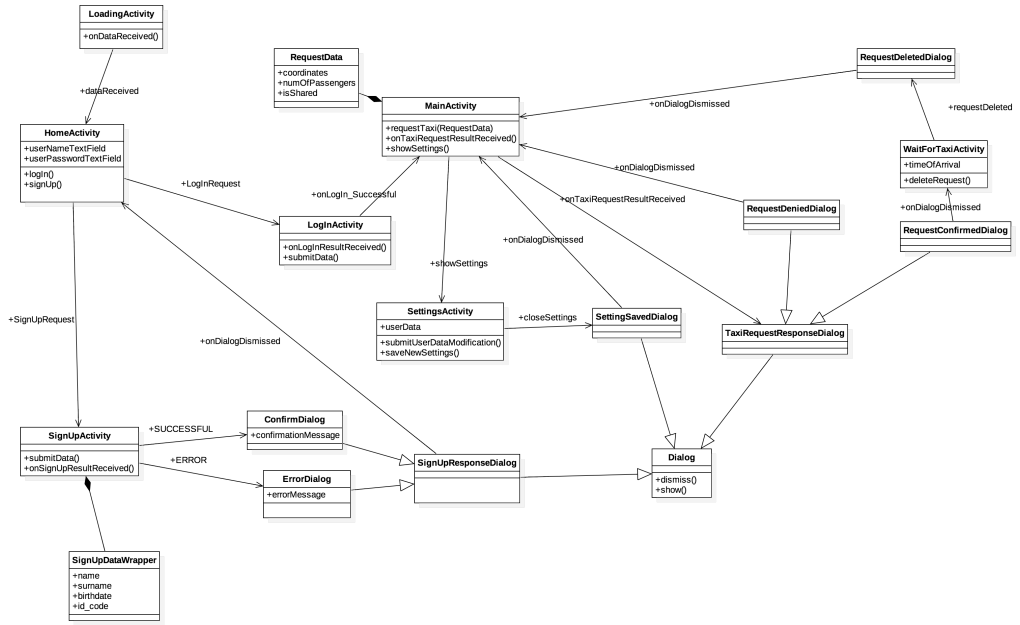
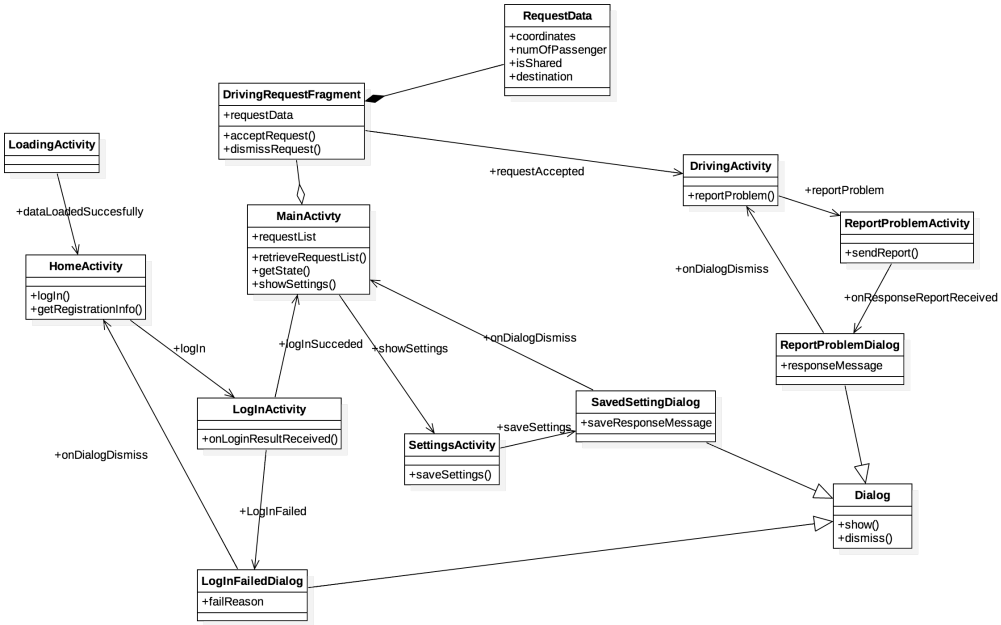


Figure 4: Driver Class Diagram



4 Deployment View

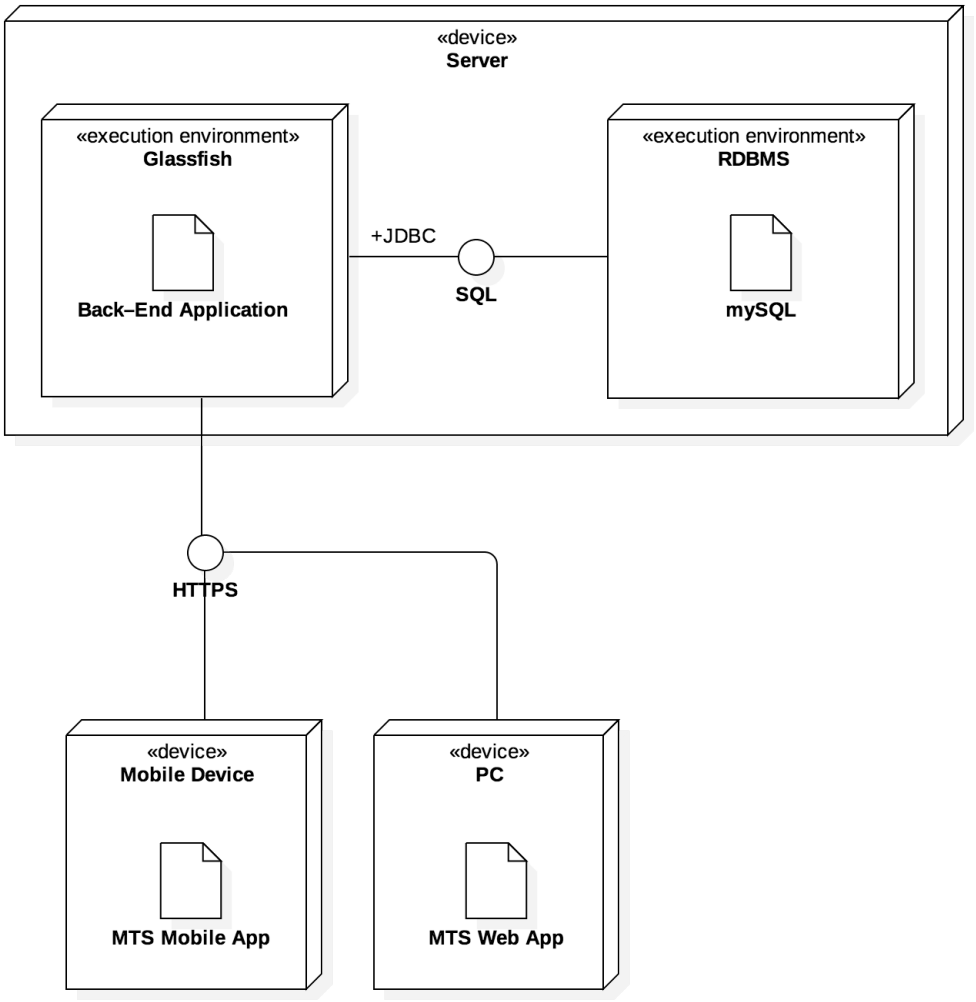
In this section we analyze the *Deployment View*, meaning that a presentation of the deployment point of view is provided.

4.1 Diagram

In order to have a successful deployment an in depth analysis of the main components that will have to be deployed is required.

For this reason here is a diagram showing the *Deployment View*:

Figure 5: Deployment Diagram



4.2 Diagram Analysis

This diagram shows a two-tier architecture.

A main server is deployed: in this node the *mySQL DataBase* is executed. This is also where the *Back-End Application* will be deployed.

Both mobile applications, the *User* one and the *Taxi Driver* one, interface with the *Back-End Application* through *HTTP* protocol. Also the PC application, namely the *Web Application*, connects to *Back-End Application*.

5 Runtime View

5.1 Server View

In this section we focus on the *Runtime View* of the system.
While the system is up and running, the *Server* receives many *HTTP* requests from different clients that are handled by a *Load Balancing* component that distributes the calls uniformly to every real machine.
Every request from the users are registered by the server and saved in the *MySQL* database.
From the *internal* point of view of the *Back-End* application, request are at first parsed by the *Request Manager* and the dispatched to Java object that is in charge of computing the result of that request. Three diagrams are provided to better exemplify the flow of events.

5.2 Client View

From the client point of view, when a *User* or a *Taxi Driver* open his app, the client starts a first *"handshake"* to check for basic authentication data and if it's successful, the client can proceed with requests.
The flow of a request start from a *UI* component (like a *button*) and is finally handled by the class that implements the *HTTP* interface.

Figure 6: Modify Personal Data

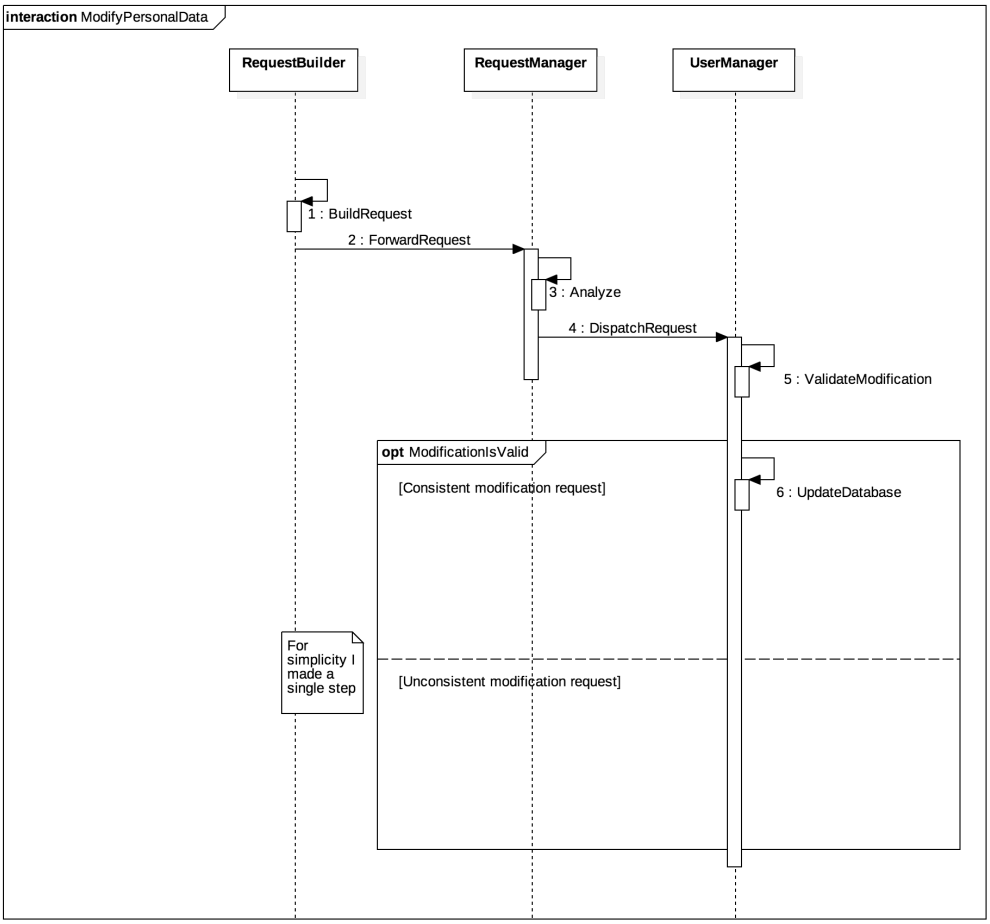


Figure 7: Taxi Request

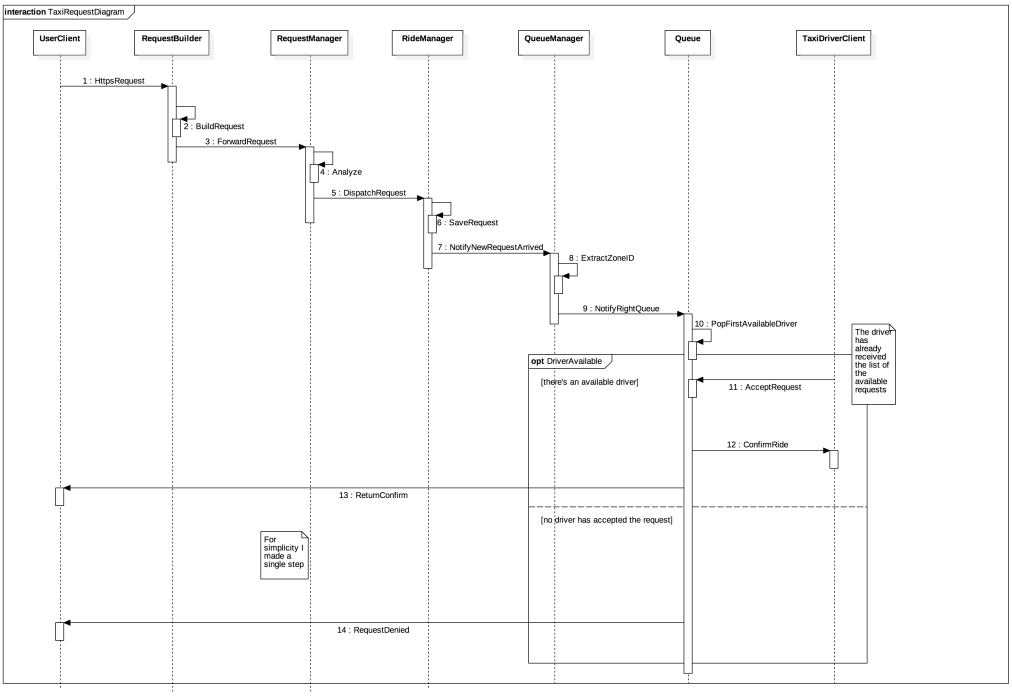
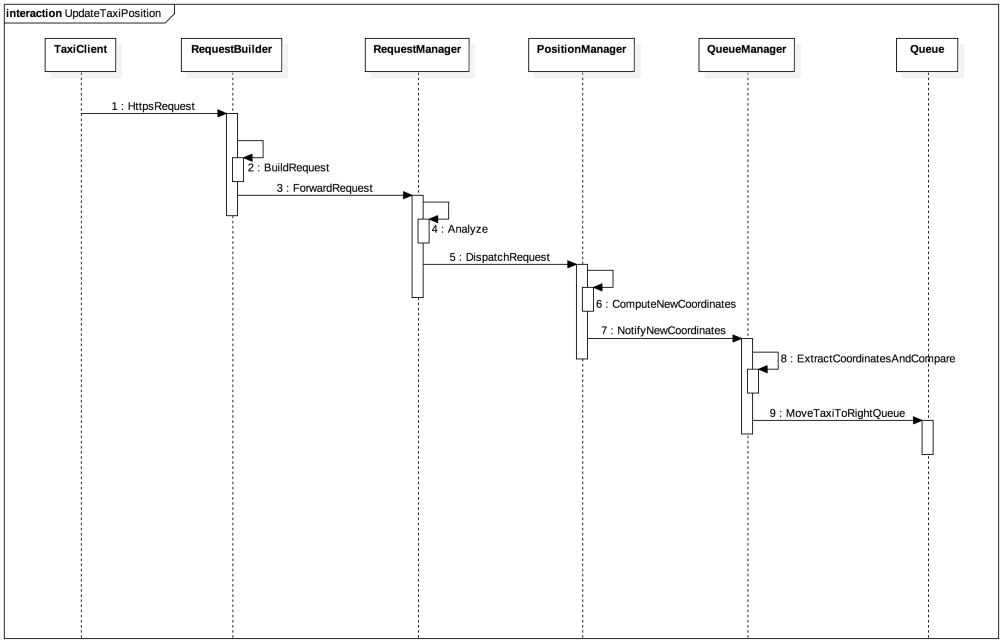


Figure 8: Update Taxi Position



6 Components Interfaces

This section provides a description of the interfaces between the main components of the system.
Internal interfaces between different objects of every component are described in *section 3*

6.1 Back-End Application - Database

The *Back-End Application* uses *SQL* language to query the *Database*. Queries from *Java* are supported by the *Java Database Connectivity (JDBC) API* which is the *API is the industry standard for database-independent connectivity between the Java programming language and a wide range of SQL databases*[4].
This choice is made in order to exploit the *"Write Once, Run Anywhere"* feature of the *JDBC*.

6.2 Back-End Application - Client User

The connection between the *Back-End Application* and the *Client User* is provided by the Internet network and based on the *HTTP* protocol and supported by a *RESTful API* service.
The two components use different internal interfaces to connect to the network[5].

6.3 Back-End Application - Client Taxi Driver

The connection between the *Back-End Application* and the *Client Taxi Driver* is provided by the Internet network and based on the *HTTP* protocol and supported by a *RESTful API* service.
The two components use different internal interfaces to connect to the network.

[4]See `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`
[5]See *section 3*

7 Architectural styles and patterns

This section highlights both hardware and software selected styles and design patterns.
It's important to underline that the rules described below are just a general guideline for the actual developers of the project.
During the development phase it's often required to switch style or rely on other patterns to carry on.
Developers should not recur to *hacks* in order to comply with design rules provided while the *system to be* is not fully specified.

7.1 Software styles and patterns

The overall software system must follow the *Object Oriented Paradigm*.
In particular, developers should follow the principles of *encapsulation*, *composition*, *inheritance*, *delegation* and *polymorphism*.
Developers should promote code reuse and should try to solve programming problems using common *OO Design Patterns*[6].
Code produced by develpers, must be fully commented and documented in order to promote simple refactoring and maintenance.

7.2 Harwdare styles and patterns

As outlined in *section 4*, our hardware system is based on a *2 tier* architecture:

- **First Tier** The first tier is composed of *clients* devices. Such as mobile phone, tablets, and browser enabled computers.
- **Second Tier** The second tier is composed of a rack of *Servers* controlled by a load balancing component.

From the logical point of view, our system is divided in *3 layers*:

- **Presentation Layer** This layer is responsible for displaying data to the users and for transmitting input data to the *business logic layer*. The *presentation layer* is deployed in the *First Tier*.
- **Business Logic Layer** This layer is responsible for receiving data from the *presentation layer*, for computing and transmitting a response

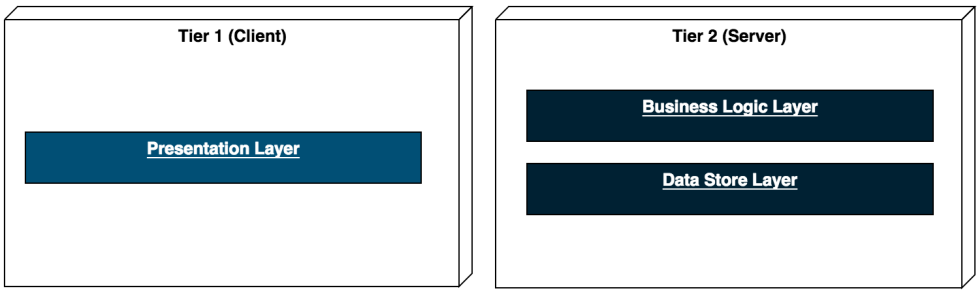
[6] *See Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

(using also data provided by the *data store layer*) to the presentation layer.
This layer implements most of the business logic and is deployed in the *second tier*.

- **Data Store Layer** This layer is responsible for storing all the data meaningful to the system. This layer is deployed in the *second tier*.

A schema of the overall architecture is provided below.

Figure 9: Architecture Diagram



8 Data management view

This section focuses on policies and data storing management.

8.1 Data Policy

8.1.1 Automatic data elimination

Data about *Users* and *Taxi Drivers* is never automatically eliminated from the system.
However rides logs are kept for 12 months to save storage space and to speed up queries.

8.1.2 Data caching policy

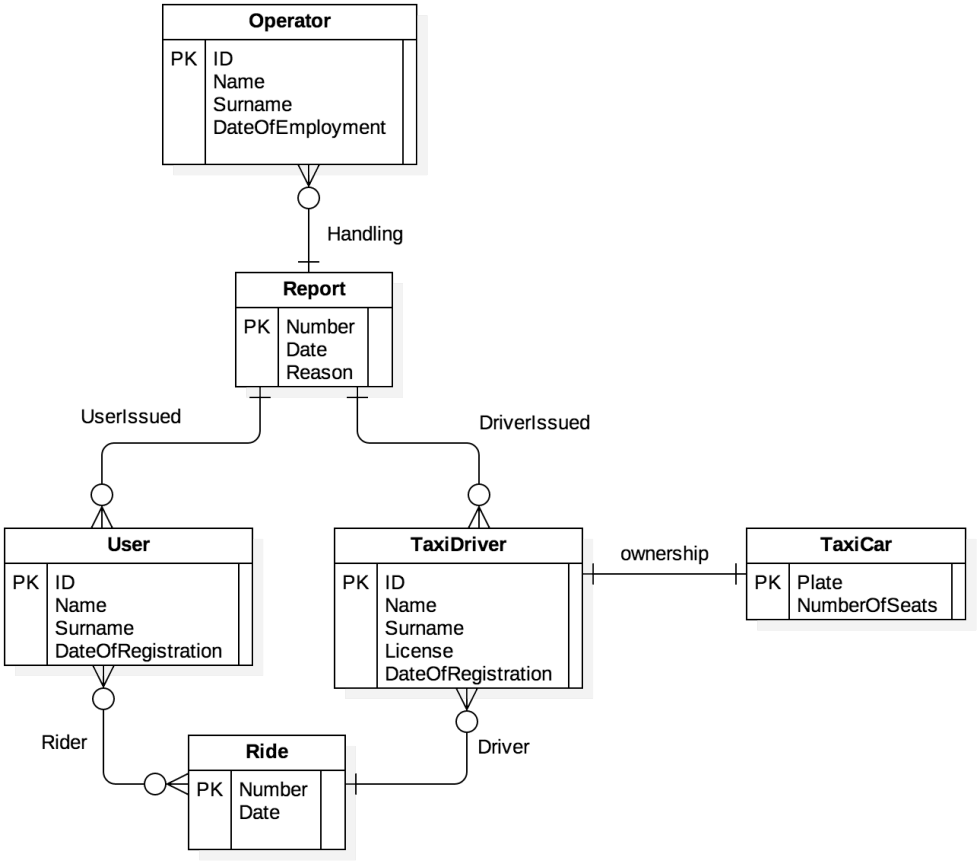
In order to reduce the load on the *Server* and to speed up user's query response, all the data that does not change frequently (user's profile data and user's preferences) is saved locally on the device and reloaded only when a modification of the profile occurs.
The caching systems is implemented in this way :

- At the first access the user enters his personal data and preferences (like name, surname, home address ...)
- This data are sent to the server (with a timestamp parameter) and saved locally on the device.
- If a different client connects to server, the server can tell the client that his data are old and sends it the new data (checking a timestamp parameter sent by the client).

8.2 Data storing

Here is a presentation of the data base schema that will have to be employed:

Figure 10: ER Diagram



8.2.1 ER Diagram analysis

The database schema is relatively simple and slim. Here is an explanation of all the entities and relationships:

Entities
The schema is composed of a total of 6 entities:

- **Operator**
  Its primary key is a unique ID, and it is the number of his/her identity card. Some of his/her personal information is stored, together with the date in wich he/she was employed.

- **User**
  Its primary key is a unique ID, and it is the number of his/her identity card. Some of his/her personal information is stored, together with the date in wich he/she registered to the service.

- **TaxiDriver**
  Its primary key is a unique ID, and it is the number of his/her identity card. Some of his/her personal information is stored, together with the date in wich he/she registered to the service. The taxi license is also stored for this entity.

- **TaxiCar**
  Its primary key is the license plate. Each *TaxiCar* has also the number of passenger seats available.

- **Ride**
  Its primary key is a unique number. In this entity it is also stored the date in which the ride has happened.

- **Report**
  Its primary key is a unique number. The other attributes are the date of the report and the reason of the report.

Relationships

- **Handling**
  This is the relation that associates each report with an *Operator*, that is the one that took care of that specific *Report*.
  Cardinality:
  - **Operator side**: 0..*
    This is because an *Operator* may have never handled any report, or he/she may have handled multiple reports.
  - **Report side**: 1
    This is because a report can be handled by one and only one *Operator*.

- **Issued**
  There are two kinds of this relation: the *User* one and the *TaxiDriver* one. This is to keep track of which one of the two kinds of customers issued the report.

  UserIssued

  This is the relation that associates the reports with a *User*, that is the one that issued that specific report.
  Cardinality:
  - **User side**: 0..*
    This is because a *User* may have never issued any report, or he/she may have issued multiple reports.
  - **Report side**: 1
    This is because a report can be issued by one and only one *Person*.

  DriverIssued

  This is the relation that associates the reports with a *TaxiDriver*, that is the one that issued that specific report.
  Cardinality:
  - **Driver side**: 0..*
    This is because a *Taxi Driver* may have never issued any report, or he/she may have issued multiple reports.
  - **Report side**: 1
    This is because a report can be issued by one and only one *Person*.

- **Rider**
  This is the relation that associates each ride with a *User*.
  Cardinality:
  - **User side**: 0..*
    This is due to the fact that a user may have never taken part to a *Ride* since he has registered to the service.
  - **Ride side**: 1..*
    This is because a *Ride* must have at least one user but may have more than one if it is a *Shared Ride*.

- **Driver**
  This is the relation that associates each ride with a *TaxiDriver*.
  Cardinality:
  - **Driver side**: 0..*
    This is due to the fact that a driver may have never taken part to a *Ride* since he has registered to the service.
  - **Ride side**: 1
    This is because a *Ride* must have exactly one and only one driver, no matter what kind of ride.

- **Ownership**
  This is the relation that associates each *TaxiDriver* with a *TaxiCar* and vice-versa.
  Cardinality:
  - **Driver side**: 1
    This is due to the fact that a driver must have exactly one and only one car in order to register to the service.
  - **Car side**: 1
    This is because a car must have exactly one and only one driver in order to exist in the system.

9 Algorithm design

9.1 Precedence Management Algorithm

This algorithm is created to manage the way taxi drivers are popped out from the queue.
The algorithm is implemented in the **QueueManager** component of the **Back-End application**.
It needs:

- The list of all the available drivers, and in every driver object a reliable coordinate should be included.
- The possibility to interact with the Google Maps API to calculate the time to reach the position of the taxi request.
- A generic math library to compute the precedenceFactor.

```
#this algorithm is created to manage the queue of taxi
   driver with clear and right rules.

from math import *

#the driver class that hold all the data and a
   precendence factor that will be update during the
   algorithm
class TaxiDriver:
        precedenceFactor = 0
        def __init__(self, ID, (x,y), inactiveTime):
                self.ID = ID
                self.coordinates = (x,y)
                self.inactiveTime = inactiveTime

#calculate the euclidean distance, obviously it will be
   replaced by the google maps api to calculate the
   distance
def calculateDistance((x1,y1), (x2, y2)):
        return sqrt(pow(x2-x1,2)+pow(y2-y1,2))

#sample data
taxiDriver = [TaxiDriver("DSIOA", (50,45), 59),
   TaxiDriver("GIAOND", (60,12), 45), TaxiDriver("PINGU"
   , (90,34), 80)]
```

```
requestPosition = (89, 67)
cellDiagonal = 100

bestPrecendenceFactor = 0
selectedDriver = 0

#for every taxi driver in the list it checks which one
   has the best precedence factor that is create
   composing the
#inactive time and the distance from the user with a
   configurable weight (in the sample is 0.2 and 0.8)
for i in taxiDriver:
        i.precedenceFactor = ((cellDiagonal -
           calculateDistance(i.coordinates,
           requestPosition))*20 + i.inactiveTime*80)/100

        if bestPrecendenceFactor > i.precedenceFactor or
            bestPrecendenceFactor == 0:
                bestPrecendenceFactor = i.
                   precedenceFactor
                selectedDriver = i

print(i.ID)
```

9.2 Shared Ride Compatibility Algorithm

This is the algorithm used to check whether two users have the possibility to share the ride.
The algorithm is implemented in the **RideManager** component of the **Back-End application**.
It needs:

- The list of all the users that are waiting for a shared ride.
- The possibility to query the Google Maps API to calculate an estimated time to complete the ride.
- The list of all the zone objects.

Notice that if a user is still waiting in the queue, that means he/she has not found a compatible user yet, as he/she would be immediately popped out of the list in case of matching. A timer of 10 minutes starts as the user requests the shared ride and as soon as it ends a dialog informing that a shared ride has not been found, will be shown.

```
class User:
        ID = "UserID"
        path = [(0,192), (34,90)]
        hasSharingRideEnable = True
        compatibleUserList = [User, User, ...]

usersWithAPendingSharingRequestList = [user1, user2,
   user3]

def calculateEstimatedPrice(path):
        #google gives this kind of API https://
           developers.google.com/maps/documentation/
           directions/intro#traffic-model
        travelTime = GoogleMapsAPI.travelTime(path)
        price = travelTime*costPerMinute
        return price

def pathAreCompatible(path1, path2):
        return getZone(path1[0]) == getZone(path2[0])
           and
         (getZone(path1[0]) in getZoneList(path2) or
                getZone(path2[0]) in getZoneList(path1))

def matchUser(users):
        users[0].compatibleUserList.append(users[1])
        users[1].compatibleUserList.append(users[0])
        sendNotificationOnCompatibilityFound(users)

#supposing that till now no matches has been found for
   the current waiting list
requestingUser = User
requestingUserPrice = calculateEstimatedPrice()

for waitingUser in usersWithAPendingSharingRequestList:
        tempCompositePath = requestingUser.path.append(
           waitingUser.path)
        #starting from the same zone and the end of one
           of them is in the path of the other
        if(pathAreCompatible(requestingUser.path,
           waitingUser.path))
                matchUsers([requestingUser, waitingUser
                   ])
```

9.3 Zone Assignment Algorithm

This algorithm assign the taxi driver to the right zone.
The algorithm is implemented in the **PositionManager** component of the **Back-End application**.
It requires:

- To be periodically triggered to update the membership of every taxi driver.
- The list of all the active taxi drivers.
- The list of all the zone objects.

```
class Point:
        def __init__(self,x,y):
                self.x = x
                self.y = y

class TaxiZone:
        taxiList = 0
        def __init__(self, ID, vertices):
                self.ID = ID
                self.vertices = vertices

class TaxiDriver:
        precedenceFactor = 0
        def __init__(self, ID, (x,y), inactiveTime):
                self.ID = ID
                self.coordinates = (x,y)
                self.inactiveTime = inactiveTime

taxiZones = a list of all the zone

#this function will be periodically called from
   QueueManager
def updateTaxiQueues(listOfAllTheTaxiDrivers):
        for driver in listOfAllTheTaxiDrivers:
                for zone in taxiZones:
                        if zone.region.contains(driver.
                           coordinates):
                                zone.taxiList.append(
                                   driver)
                                break
```

10 User Interface View

This section focuses on the user interface point of view using UX UML diagram to specify the possible interactions and the flow of events.
Layouts for the applications are described and outlined in *section 3.1* of the *RASD*.

Figure 11: User App Login UX

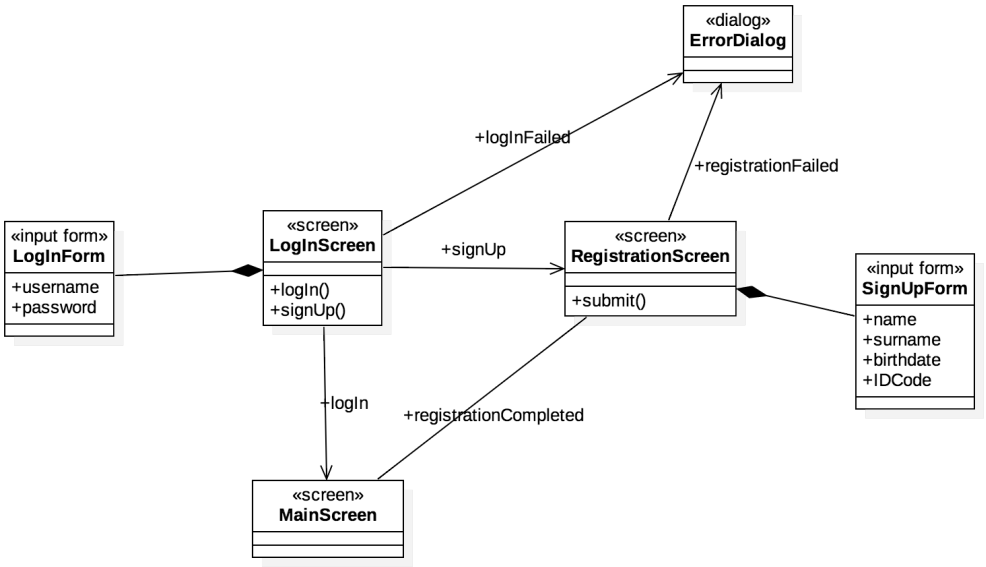


Figure 12: User App MainScreen UX

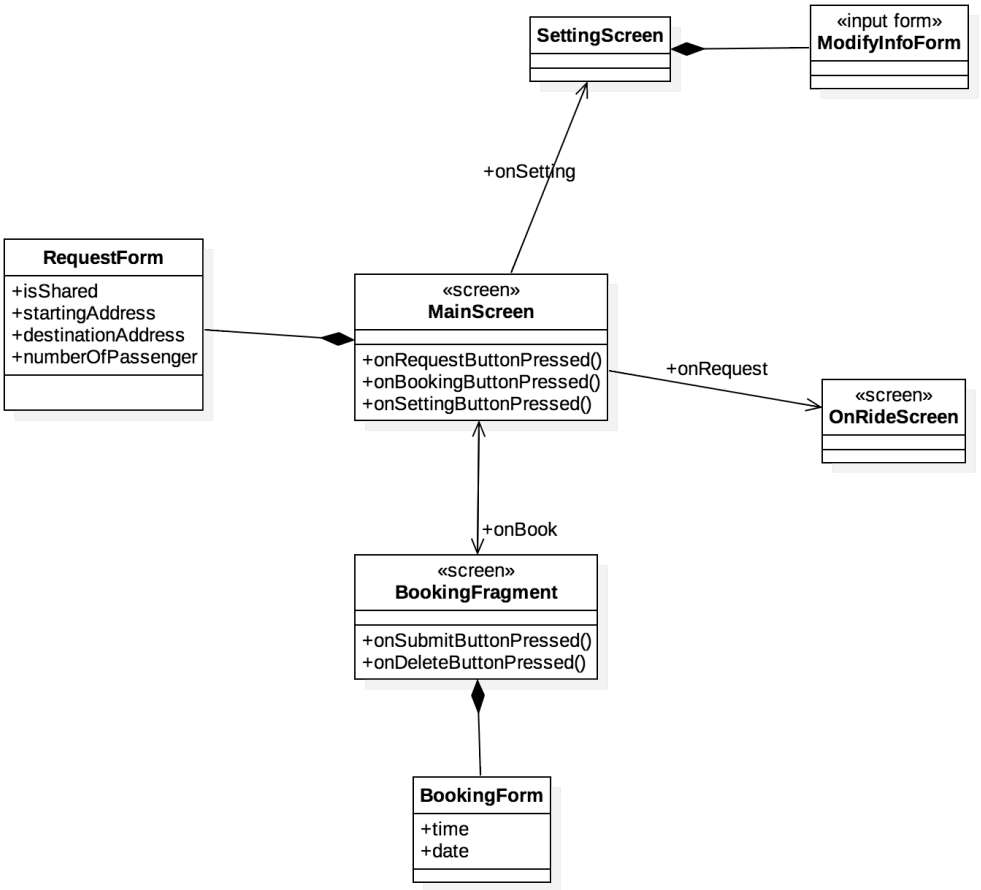


Figure 13: User App On Ride Screen UX

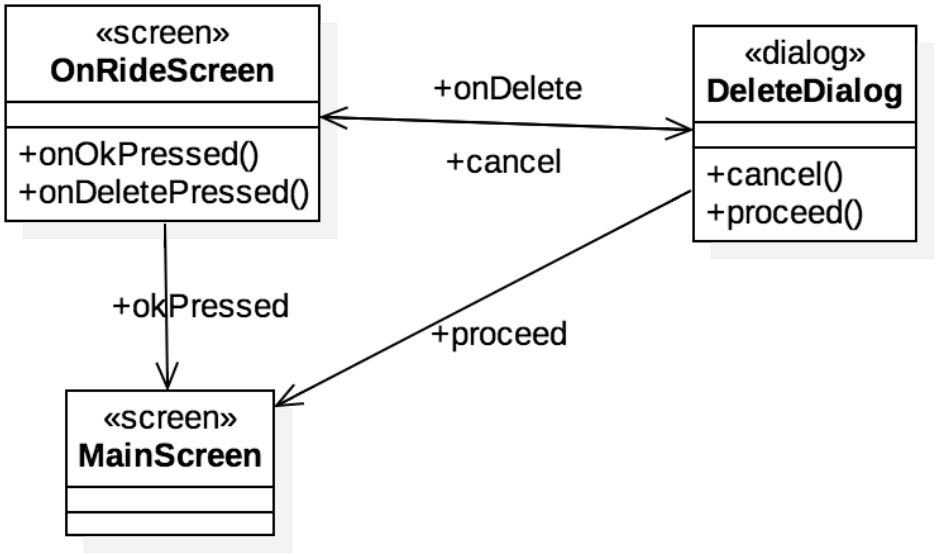


Figure 14: Taxi Driver App Login UX

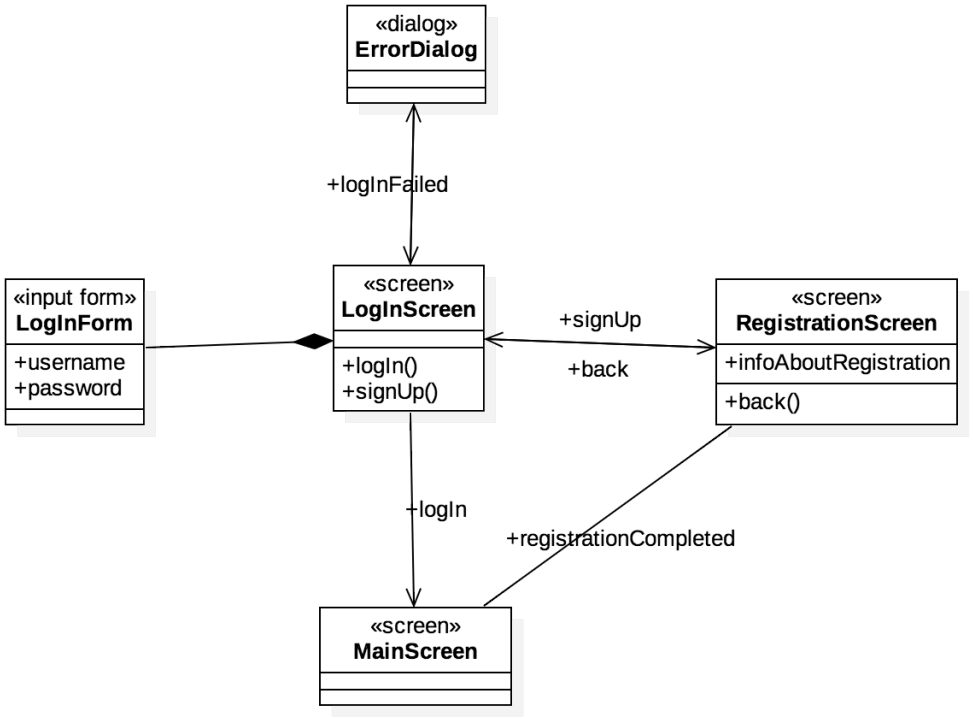
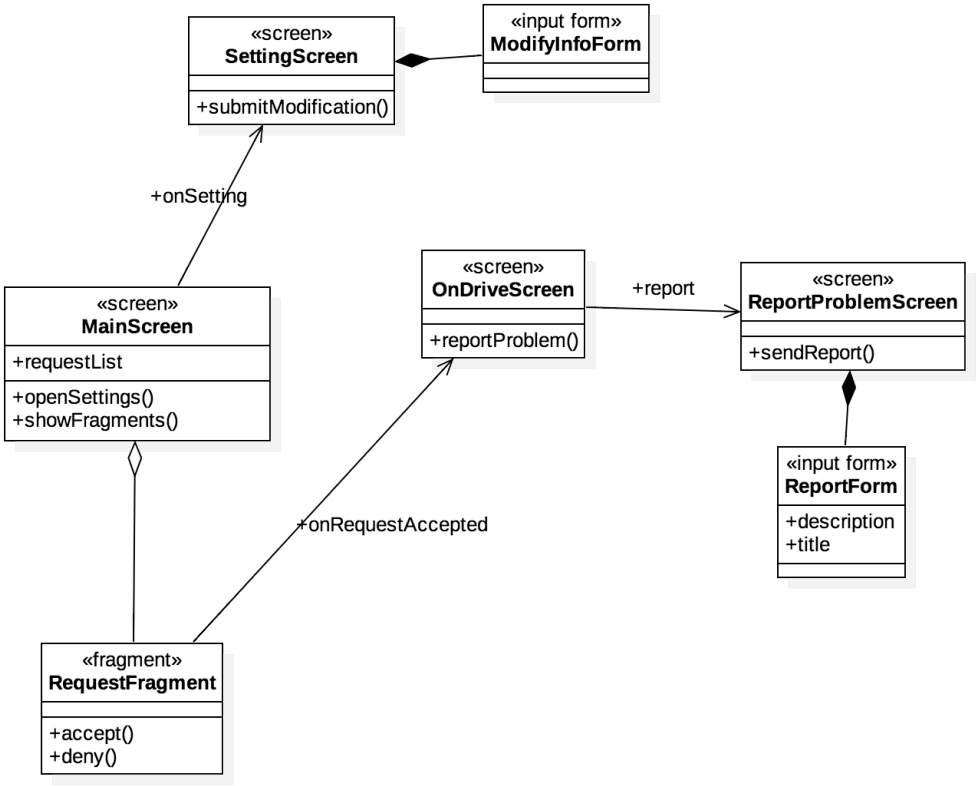


Figure 15: Taxi Driver App MainScreen UX



11 Requirements traceability

This section shows how the implementation design presented in this document satisfies the requirements discussed in the **RASD** document.

Table 1: Requirements

Component	**Requirements**
LoginActivity (User Client)	FR1, FR2, FR13, FR14
MainActivity (User Client)	FR3-FR11, FR15-FR23
SettingsActivity (User Client)	FR12, FR24
LoginActivity (Driver Client)	FR25
MainActivity (Driver Client)	FR26, FR27, FR28, FR31
DrivingActivity (Driver Client)	FR29, FR31, FR32, FR33, FR34
ActorManager (Back-end)	FR35, FR37, FR38, FR39, FR46, FR47, FR49, FR50
RideManager (Back-End)	FR36, FR42, FR45, FR51, FR52, FR53
QueueManager (Back-End)	FR41, FR43, FR44,
DbInterface (Back-End)	FR40, FR46, FR47

Appendices

A Tools

- *Sublime Text 2* as editor
- *LatexTools for Sublime Text 2 + MacTex* to build
- *Trello* for team coordination
- *Git + Git Flow* for version control

B Hours of work

- Angelo Gallarello : 20 hours
- Edoardo Longo : 20 hours
- Giacomo Locci : 20 hours

C Changelog

Version 1.1

- Added *Overview Diagram* on page 6.
- Clarification on the *Data caching policy* section on page 22.
- Added the specific *components* that will implement each algorithm in *Algorithm design* section on page 27.
- Added the *Requirements traceability* section on page 36.
- Fixed the paragraphing issues in sections *High Level Components* and *Deployment View*.