myTaxyService

Design Document

Version 1.0

Matteo Farè, Federico Feresini, Thierry Ebali Essomba

December 4, 2015

Contents

1 Introduction **2**
- 1.1 Purpose . . . 2
- 1.2 Scope . . . 2
- 1.3 Definitions, Acronyms, Abbreviations . . . 2
- 1.4 Reference Documents . . . 3
- 1.5 Document Structure . . . 3

2 Architectural Design **4**
- 2.1 Overview . . . 4
- 2.2 High Level Components And Their Interaction . . . 5
  - 2.2.1 Customer-related Components . . . 5
  - 2.2.2 Driver-related ComponentsDD Diagrams . . . 6
  - 2.2.3 Component Description . . . 6
  - 2.2.4 Database Structure . . . 11
  - 2.2.5 Specific Interactions Between Components . . . 11
- 2.3 Component View . . . 14
  - 2.3.1 Customer Component View . . . 14
  - 2.3.2 Driver Component View . . . 15
  - 2.3.3 External User Component View . . . 15
- 2.4 Deployment view . . . 16
- 2.5 Runtime View . . . 18
  - 2.5.1 Registration . . . 18
  - 2.5.2 Login . . . 18
  - 2.5.3 Ride Request . . . 19
  - 2.5.4 Status Management . . . 20
  - 2.5.5 Sharing . . . 20
  - 2.5.6 Booking . . . 21
- 2.6 Component Interfaces . . . 22
  - 2.6.1 Registration Manager . . . 22
  - 2.6.2 Driver Login Manager . . . 22
  - 2.6.3 Customer Login Manager . . . 22
  - 2.6.4 Immediate Ride Request Manager . . . 22
  - 2.6.5 Booking Ride Request Manager . . . 23

2.6.6 Booking Scanner . . . 23
2.6.7 Sharing Ride Request Manager . . . 23
2.6.8 Rides Sharing Viewer . . . 23
2.6.9 Customer Notification Manager . . . 23
2.6.10 Customer Decision Manager . . . 24
2.6.11 Payment Manager . . . 24
2.6.12 Availability Manager . . . 24
2.6.13 Position Tracking Manager . . . 24
2.6.14 Top Queue Extractor . . . 25
2.6.15 Queues Manager . . . 25
2.6.16 Driver Notification Manager . . . 25
2.6.17 Driver Decision Manager . . . 25
2.7 Selected Architectural Styles And Patterns . . . 26
2.7.1 Architectural Styles . . . 26
2.7.1.1 Message Bus Architectural Style (Communication) . . . 26
2.7.1.2 Object-Oriented (Structure) . . . 26
2.7.2 Architectural Pattern . . . 27
2.7.2.1 Model - View - Controller . . . 27
2.7.3 Design Pattern . . . 28
2.7.3.1 Intercepting Filter Pattern . . . 28
2.7.3.2 Observer Pattern . . . 31

3 Algorithm Design 33
3.1 Security Check . . . 33
3.2 Driver Availability . . . 33
3.3 Remove Driver . . . 34
3.4 Ride Request . . . 34
3.5 Driver Notification . . . 35
3.6 Request Manager . . . 36
3.7 Ride Booking . . . 37

4 User Interface Design 38
4.1 Driver Interface . . . 38
4.2 Customer Interface . . . 39

5 Requirements Traceability 40

6 References 43

Chapter 1

Introduction

1.1 Purpose

The purpose of this document is to provide documentation for design and implementation of the myTaxiService application. Both high-level requirements and implementation details are documented here to ensure successful completion of the project and continuity for future project development.

1.2 Scope

This document will provide detailed specification for designing, implementing and configuring software for myTaxiService. We will show all the design decision that we have taken in order to make myTaxiService application better and better. Thorough the next sections are presented and described all the high-level components of the system, the architectural styles that we have chosen and the pattern selected in order to have a cleaner and more efficient coding. Finally, some of the algorithms that we are going to develop are presented in the specific chapter. All our choices have been made thinking of what are the main functions of myTaxiService and what it needs in order to develop that application in the best possible way.

1.3 Definitions, Acronyms, Abbreviations

- RASD: Requirements Analysis and Specification Document.
- DB: DataBase.
- CML interface: Customer Login interface.
- SRRM interface: Sharing Ride Request interface.
- CDM interface: Customer Decision Manager interface.

- CNM interface: Customer Notification Manager interface.
- PM interface: Payment Manager interface.
- BRRM interface: Booking Ride Request interface.
- IRRM interface: Immediate Ride Request interface.
- RM interface: Registration Manager interface.
- DLM interface: Driver Login Manager interface.
- QM interface: Queues Manager interface.
- DNM interface: Driver Notification Manager interface.
- AM interface: Availability Manager interface.
- TQE interface: Top Queue Extractor interface.
- DDM interface: Driver Decision Manager Interface.

1.4 Reference Documents

- myTaxiServiceRASD.
- Template for the design document - Software Engineering II, AA 2015-2016.

1.5 Document Structure

- **Architectural Design**: Here are shown the main architecture and design choices in order to create myTaxiService application.
- **Algorithm Design**: In this section the main algorithms used to develop some of myTaxiService functions are described in a quality way.
- **User Interface Design**: In this section two User Experience Diagrams show the interfaces through which users interacts with the application.
- **Requirements Traceability**: Here is shown how the functional requirements map in the components previously presented.

Chapter 2

Architectural Design

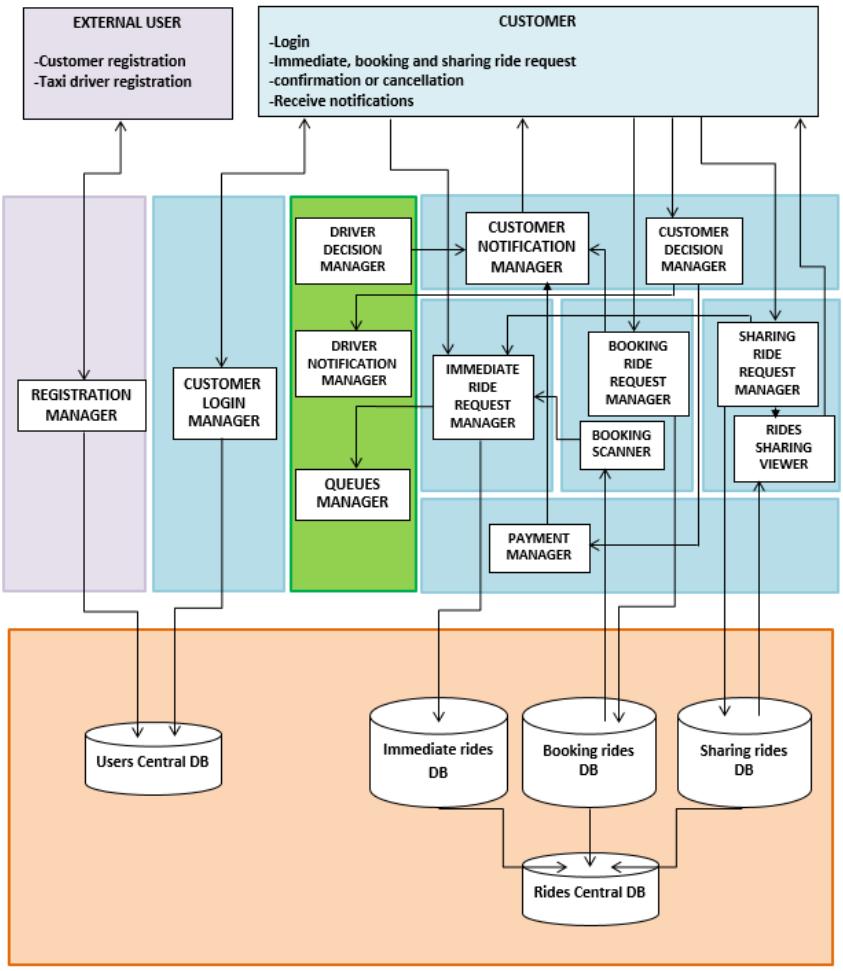
2.1 Overview

In this chapter we are presenting:

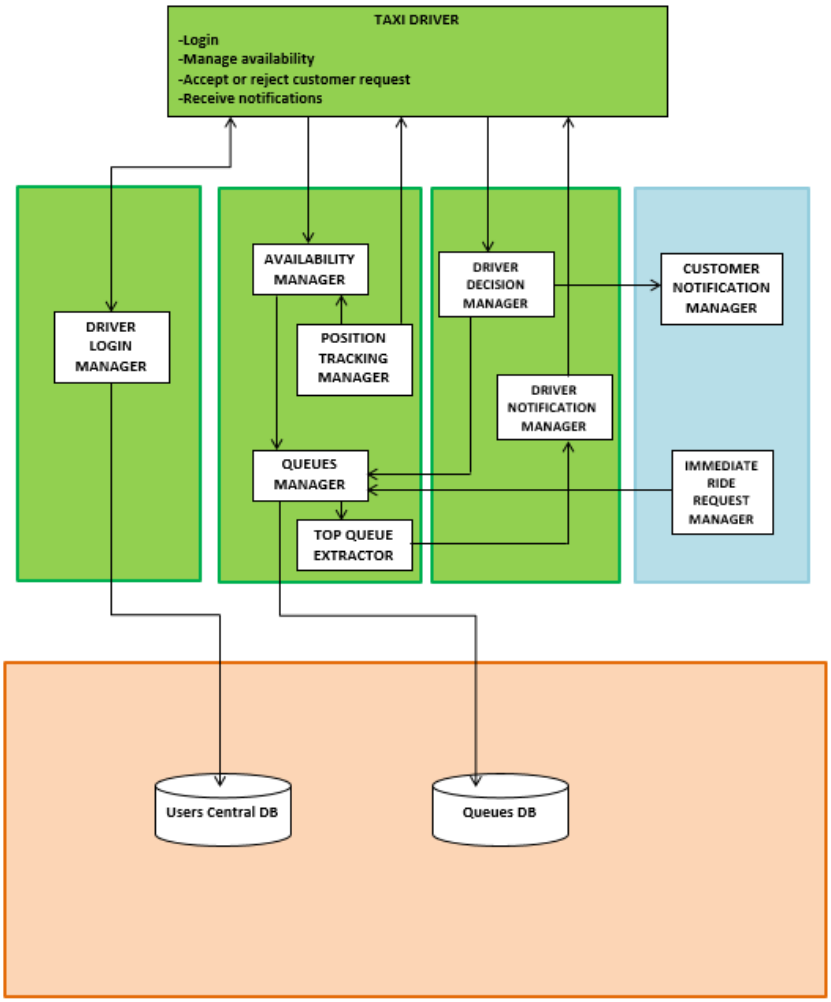
- All the components of the system, and how they interacts. A description of each component is also provided, specifying for each one the needed input, the function(s) that it is supposed to do and the desired output.
- Our choices regarding the needed hardware in order to deploy myTaxiService.
- Some sequence diagrams that let visualize some interactions between the components.
- How component interfaces may be implemented.
- The chosen architectural styles and patterns.

2.2 High Level Components And Their Interaction

2.2.1 Customer-related Components



2.2.2 Driver-related Components



2.2.3 Component Description

1. Registration Manager:

   - Customer Input:
     First name, surname, born date, address, telephone number, username, password, credit card ID.
   - Driver Input:
     Name, Surname, E-mail address, Telephone number, License number, Username, Password, Vehicle number plate, Vehicle brand, Vehicle model, Maximum number of passengers, IBAN,
   - Functions:

(a) Validity check of the input
(b) Storage of users' data into the Users Central DB
(c) Forwarding the registration summary to the user

- Output:
  (a) String with user data
  (b) Success or failure notification

2. Customer Login Manager:

- Input:
  Username, Password
- Functions:
  (a) Validity check of the input searching in the User Central DB a compatibility
  (b) Forwarding login summary to the customer
- Output:
  Success or failure notification

3. Driver Login Manager:

- Input:
  Username, Password, Taxi Code
- Functions:
  (a) Validity check of the input looking in the User Central DB for a compatibility
  (b) Forwarding login form to the driver
- Output:
  Success or failure notification

4. Immediate Ride Request Manager:

- Input:
  Starting point, Destination point
- Functions:
  (a) Validity check of the request
  (b) Storage of request code into Immediate Rides DB
  (c) Reception and management of booking request
  (d) Reception and management of a new shared ride
- Output:
  String with request code

5. Booking Ride Request Manager:

- Input:
  Starting point, Destination point, Date and time
- Functions:
  (a) Validity check of the request
  (b) Storage of request code into Booking Rides DB
- Output:
  String with request code

6. Sharing Ride Request Manager:

   - Input:
     starting point, Destination point
   - Functions:
     (a) Validity check of the request
     (b) Storage of request code into Sharing Rides DB
     (c) Forwarding of unmatched requests
   - Output:
     (a) New request for Immediate Ride Request Manager
     (b) String with request code

7. Booking Scanner:

   - Input:
     Booking Rides DB strings
   - Functions:
     Scanning strings in Booking Rides DB for each interval
   - Output:
     New request for Immediate Ride Request Manager

8. Rides Sharing Viewer:

   - Input:
     Request data, Sharing Rides DB strings
   - Functions:
     Creation and forwarding of a rides set that match with the customer request
   - Output:
     Rides set

9. Payment Manager:

   - Input:
     Customer confirmation

- Functions:
  Automatic management of transaction related to request

10. Customer Notification Manager:

    - Input:
      Summary request, Valuation of booking request
    - Functions:
      Elaboration of the input and forwarding to the customer the related notification
    - Output:
      Notification

11. Customer Decision Manager:

    - Input:
      Customer decision
    - Functions:
      (a) Request confirmation and redirecting of the payment
      (b) Forwarding customer decision to the driver
    - Output:
      (a) User data and payment amount
      (b) Customer decision

12. Availability Manager:

    - Input:
      Driver availability, Queue code
    - Functions:
      (a) Receiving driver position
      (b) Receiving driver availability
      (c) Forwarding availability data to the Queues Manager
    - Output:
      String with availability data

13. Position Tracking Manager:

    - Input:
      Driver position
    - Functions:
      (a) Matching driver position and related queue code
      (b) Forwarding code to the Availability Manager

- Output:
  Queue code

14. Top Queue Extractor:
    - Input:
      Code request, Queues list
    - Functions:
      (a) Extraction of the first driver from the selected queue
      (b) Forwarding the request to the extracted driver
    - Output:
      Data request

15. Queues Manager:
    - Input:
      Queue code, Driver code, Driver decision, Request code
    - Functions:
      (a) Queue and driver position updating
      (b) Forwarding the request to the selected driver
    - Output:
      (a) Request code
      (b) Queue code

16. Driver Decision Manager:
    - Input:
      Driver decision
    - Functions:
      (a) Forwarding the input to the Queues Manager
      (b) Forwarding the input to the Customer Notification Manager
    - Output:
      Driver decision

17. Driver Notification Manager:
    - Input:
      Request code
    - Functions:
      (a) Extracting the request description from the request code
      (b) Forwarding the request description to the driver
    - Output:
      Request description

2.2.4 Database Structure

- **Users Central DB**: There will be a DB for storing the data of all users. specifically, it keeps track of personal information of customers (first name, surname, born date, address, telephone number, username, password and credit card ID) while it keeps also track of the extremes of the license and the vehicle that taxi drivers declare at the moment of registration (name, surname, email address, telephone number, license number, username, password, vehicle number plate, vehicle brand, vehicle model, IBAN, and the maximum number of passengers that he can carry simultaneously).

- **Immediate Rides DB**: Any cab request for immediate ride that is confirmed is recorded in the DB, which holds all codes of active races.

- **Booking Rides DB**: Each reservation request conforms to the specifications and positively evaluated is stored in the DB with his specific code until the submission of the request to the queues manager.

- **Sharing Rides DB**: Each shared ride already created and accepted near to the starting time is stored in the DB with his specific code until time of its actual beginning.

- **Rides Central DB**: The codes of rides completed are stored in the DB, which keeps a history of all codes rides associated to the respective customers and taxi driver.

- **Queues DB**: The DB keeps track of associations between drivers available, their area code and their position in the relative queue.

2.2.5 Specific Interactions Between Components

- It will require a management component for the external users registrations (**Registration Manager**) functioning for the specific recording of drivers and customers. This component interacts directly with the central database containing all the data for the existing users, and saves in it the data of the new user. The interaction between the user and the component is two-way, because it is necessary that it can communicate with user in case of errors during the registration process.

- Each customer will have a login function, which will be managed by a component (**Customer Login Manager**) that interacts with the central database to verify the credentials and provide access to the personal page of the client. In this case the interaction is again two-way for the signals of errors.

- Each driver will have a login function, which will be managed by a component (**Driver Login Manager**) that interacts with the central database to verify the credentials and provide access to the personal page of the driver. In this case the interaction is again two-way for the signals of errors

- Each client can make a request for an immediate ride, which is managed by a component (**Immediate Ride Request Manager**) which interacts with the queue handler (**Queues Manager**) in order to forward the request to the first driver in queue for the area of interest. This component interacts with the Immediate Rides DB to store in it the code of the request. It is equipped with a one-way interaction with the user, to avoid encumbering its tasks and entrust the response to a specific component.

- The request for a booking is handled by a specific component (**Booking Ride Request Manager**) that evaluates requests and communicates with the database of reservations. The DB stores the reservation code requests and, every certain period of time, is scanned by a control component (**Booking Scanner**) which checks times of booking and manages the requests coming at the appointed time, communicating just with the Immediate Ride Request Manager. Also this component is equipped with a one-way interaction with the user, setting it free from response tasks.

- For sharing requests a special component (**Manage Sharing Ride Request**) communicates with a database that stored all shared races codes coming at the start. If there is no ride shared in the DB concordant with the parameters set by the customer it will manage the request as a new request for an immediate ride, communicating with the appropriate component. Upon confirmation the component interacts with the Sharing DB to store the new shared race code. If there is one or more races that meet the parameters entered by the user, the rides list is returned to the client through the **Sharing Rides Viewer**.

- All notifications that the system needs to submit to the customer are managed by **Customer Notification Manager**. This component interacts with the **Driver Decision Manager** to inform the customer of the outcome of his request for immediate or sharing ride, and interacts with the **Booking Ride Request Manager** to inform the customer of the outcome of its booking inquiry. This component compensates for the absence of dual communication in the interaction between the user and the components for requests.

- All the client decisions are managed by the **Customer Decision Manager**, which interacts with the **Driver Notification Manager** for the exchange of messages between the driver and the customer during the confirmation of the ride. It interacts also with the **Payment Manager**, to manage transactions automatically once the ride is confirmed.

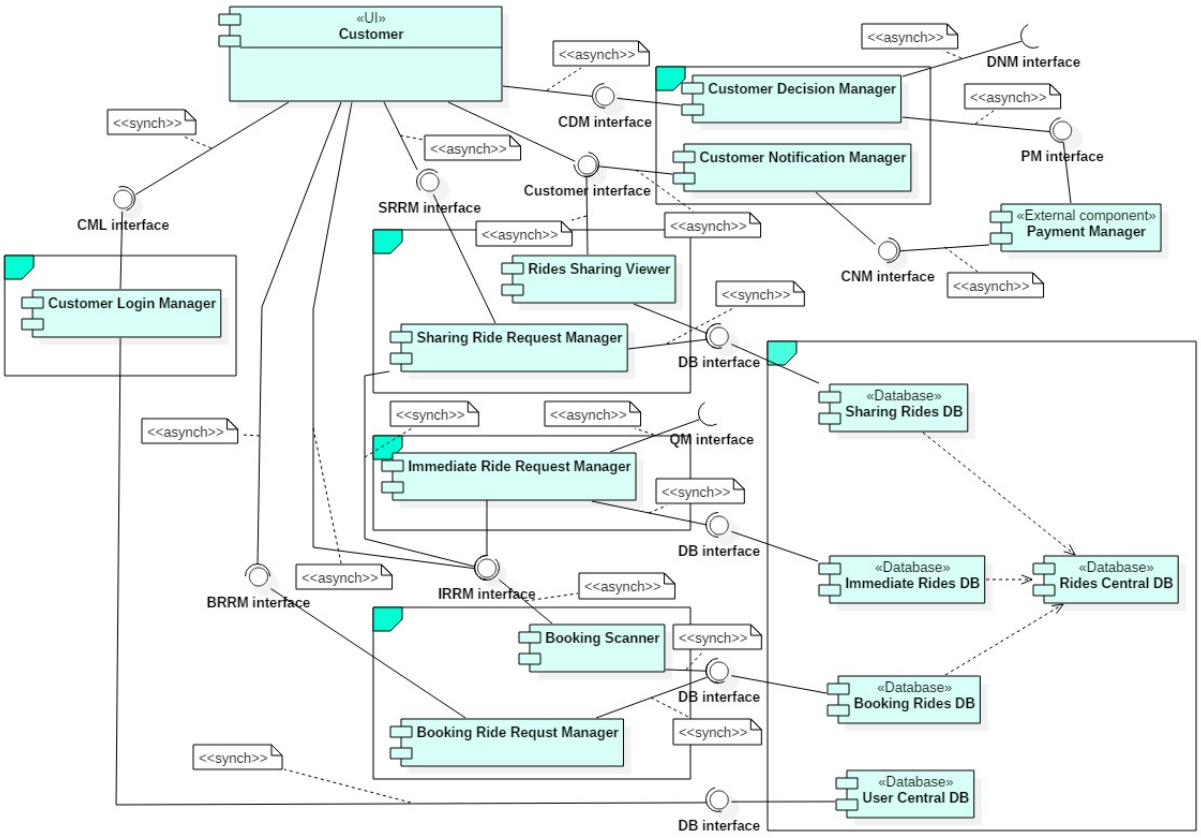
- Each taxi driver can manage his availability. This functionality is managed by the **Availability Manager**, which interacts with the **Position tracking Manager**. This second component retrieves the position of the driver when he changes his availability. This two components together interact with the **Queues Manager**, that receives couples driver-code area and it updates the queues by changing the data in the Queues DB. All interactions are one-way delegating the response tasks to specific components.

- All driver's decisions are managed by the **Driver Decision Manager**. This component has a one-way interaction with the user. It interacts with the Queues Manager, to update the queues based on the driver decision (when a driver refuses a request is placed in the back of the queue), and it has also a one-way interaction with the **Customer Notification Manager**, to communicate to the customer the request recap.

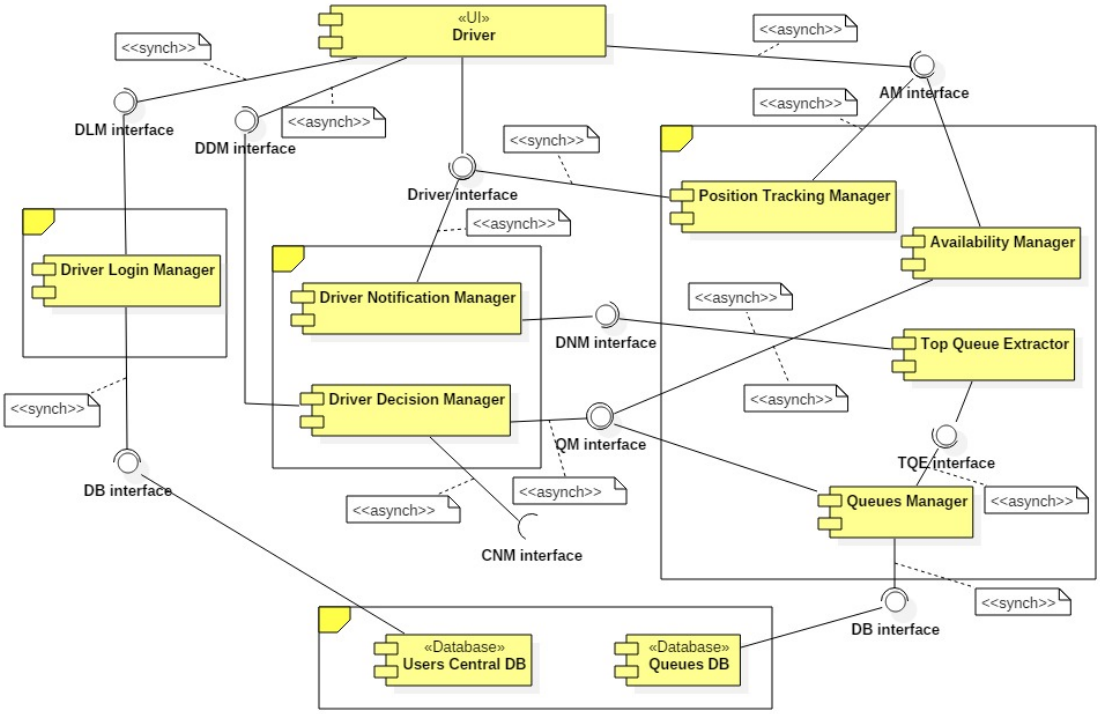
2.3 Component View

For the interactions between the components in the most of cases we have thought to adopt one-way interactions, in order to simplify the management of the information flow. We hypothesized two-way interactions only in accessing the database, where response is required for the formulation of queries, and also in the functions of registration and login, to report the results of operations to the user. The division between synchronous and asynchronous interactions follows the same logic as before.

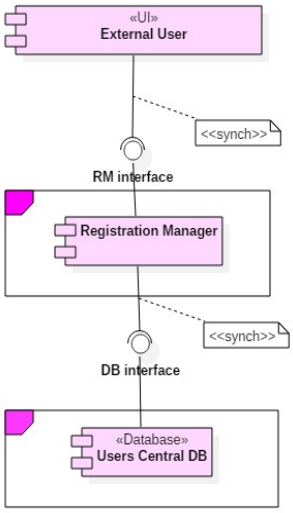
2.3.1 Customer Component View



2.3.2 Driver Component View



2.3.3 External User Component View



2.4 Deployment view

The Application myTaxiService aims to interface with an open set of users through Internet. The inability to predict a priori the processing capacity of the different clients makes less than optimal development of architecture in key fat client, a choice that would introduce the added disadvantage of a complex administration in the update to the application layer. These considerations promote the development of architecture in anticipation of thin clients, typical choice of modern distributed systems.

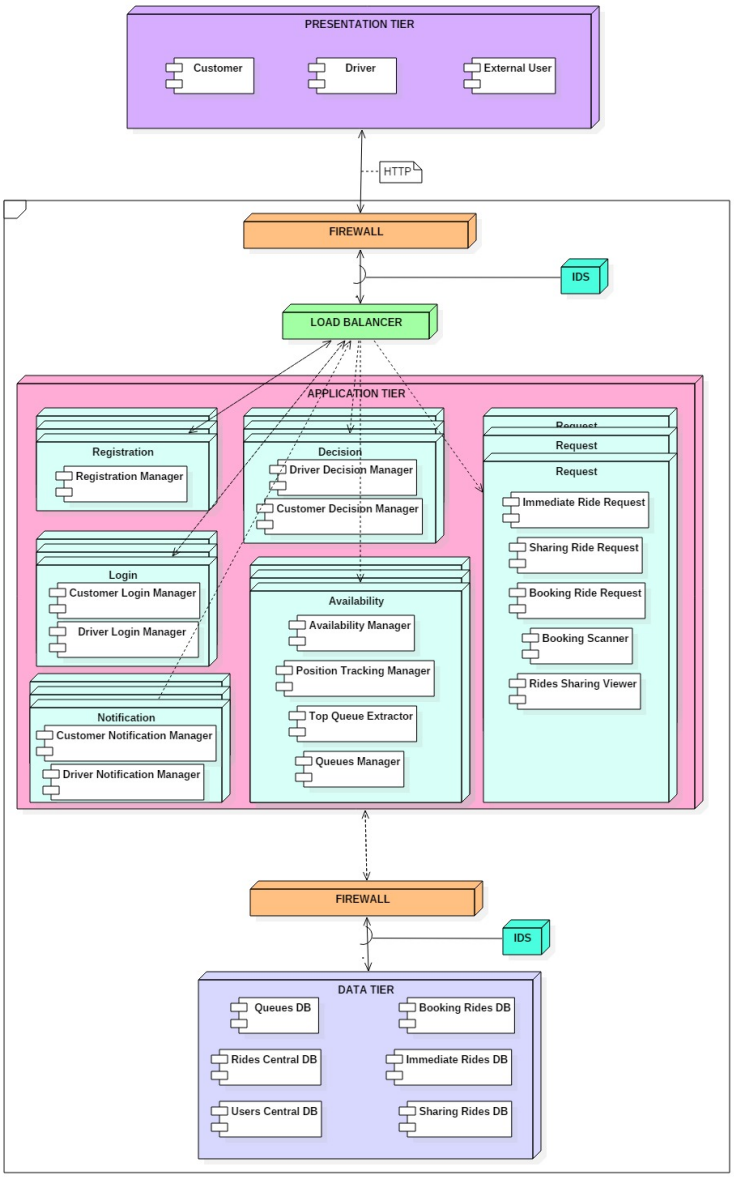
In order to optimize the properties of scalability and reliability of the system, it would be an optimal choice to make use of a 3-tier configuration with server farms. The limitation to only three tier allows to keep a good reliability, because increasing the number of tier increases also the probability of single faults that can make the system unavailable. The scalability, thanks to the presence of the server farm for the replication of the critical components, is optimized by the introduction of additional machines.

For the Web server and the Script engine the best results are obtained with a RACS shared-nothing configuration, because data replication on each clone would allow, with the introduction of an appropriate load balancer, the optimal partition of incoming requests from network between servers clones. The load balancer allows requests from the same client to be forwarded to the same server that will keep the session active. In response to possible failure of the server it requires that the session state should be recovered from another server (recovery from DBMS), in order not to be interrupted.

To ensure scalability and service availability, the DBMS Server using a RAPS configuration server farm. This configuration provides that the data are divided between the different servers, which are thus able to carry out each one specific functionality. To avoid the risk of losing functionality due to a failure in the single server (graceful degradation), this configuration provides that each server to have clones with the same data, to guarantee always the availability of the entire system.

As regards safety, the architecture envisages the presence of two firewalls and the installation of IDS. To establish the connection between the client and the target service is used an application proxy firewall that having security policies at a higher level than a packet filtering firewall. It is ideal to allow user authentication, or not allowing access to certain features, allowing or not the access to certain features. For the access to data is used instead a firewall packet filtering, can provide excellent performance, though its configuration could be complex.

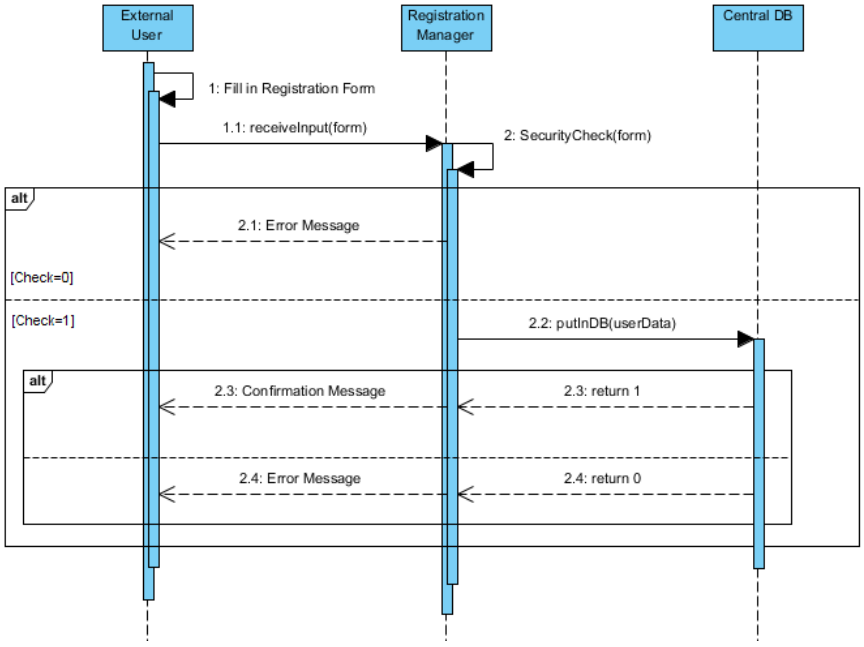
As for the Intrusion Detection System (IDS) Host based IDS is installed on servers that contain sensitive data (database) because the most effective in determining whether an attack was successful and it can also analyse encrypted data, while at the point input-output from the network is installed a network-based IDS for the need to monitor the network using a limited number of sensors. The system will interact with other systems using the technologies based on web services for the exchange of sensitive information about the payment transactions.



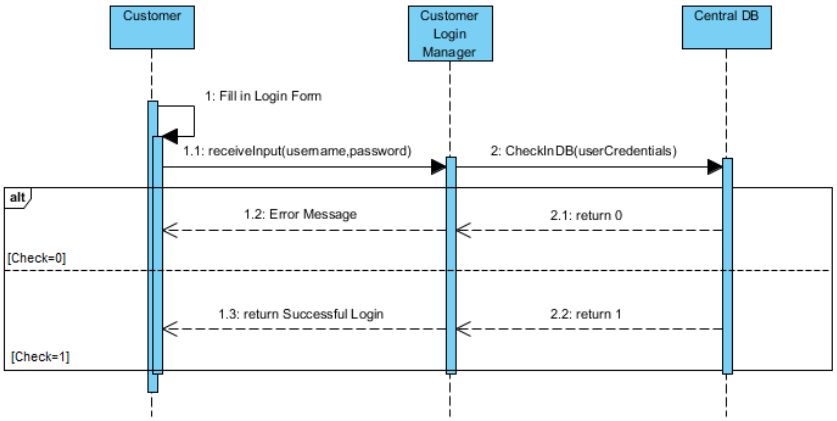
2.5 Runtime View

These sequence diagrams explain better how the component interacts in order to accomplish myTaxiService main functions.

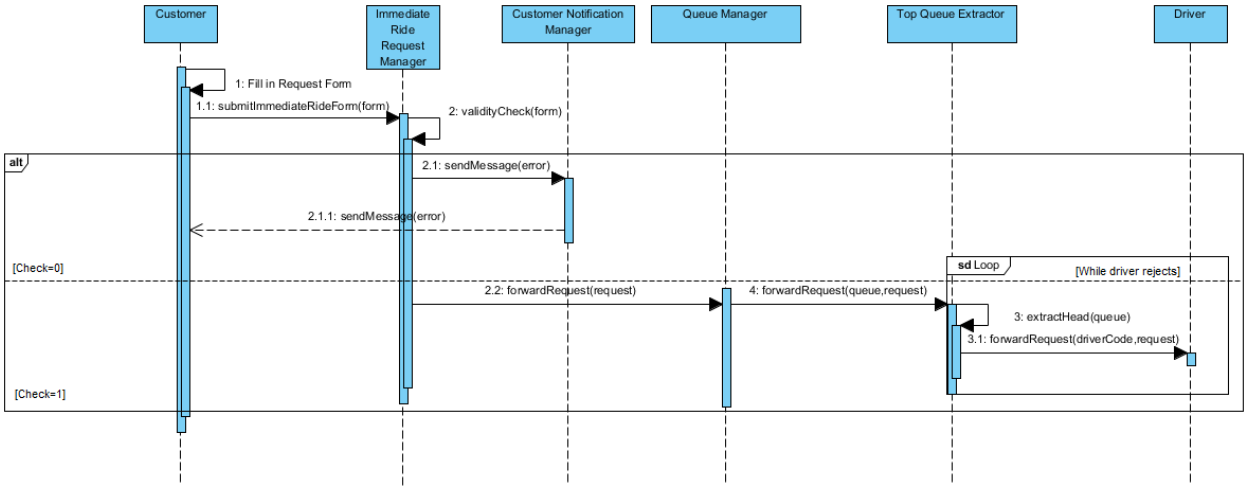
2.5.1 Registration

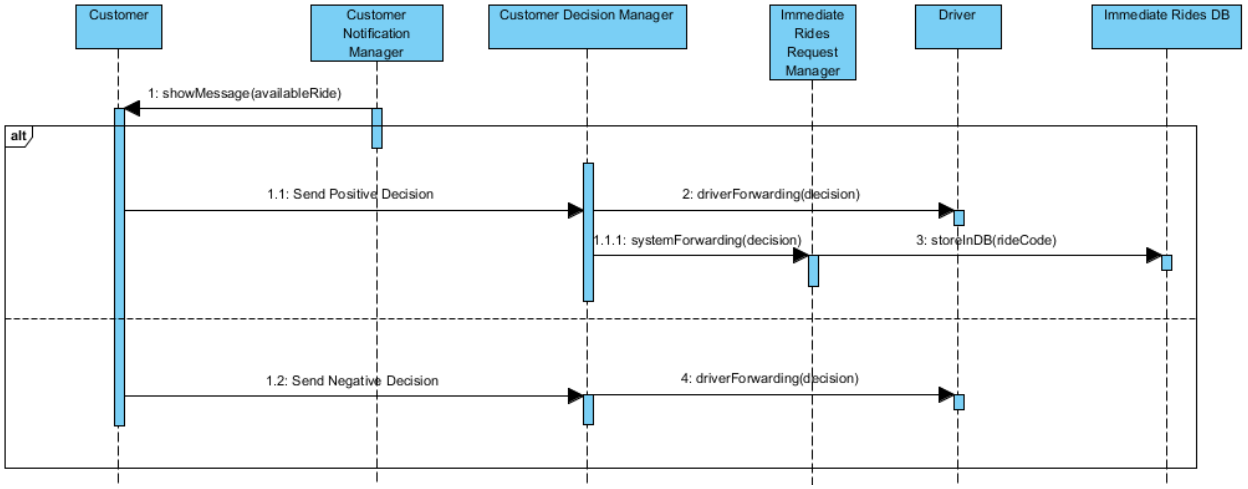


2.5.2 Login

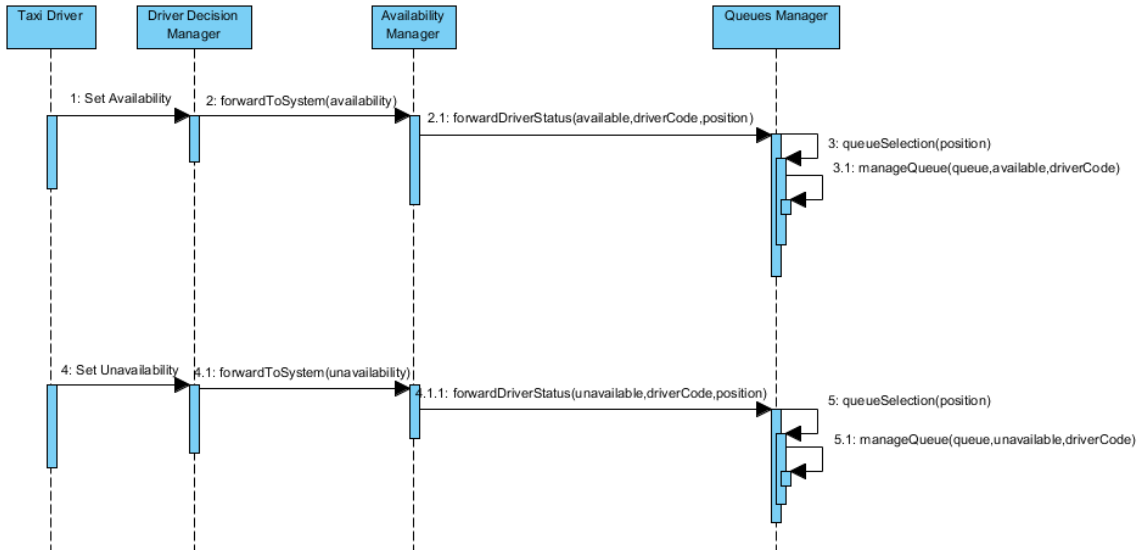


2.5.3 Ride Request

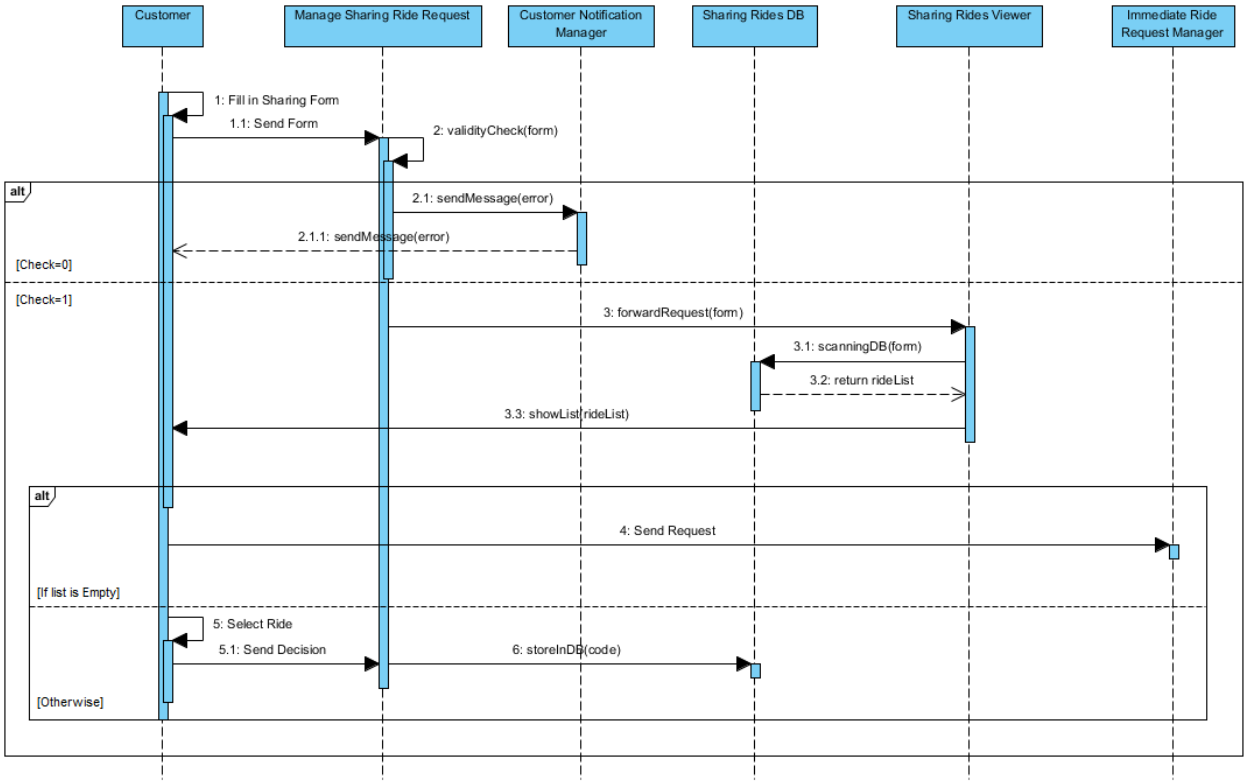




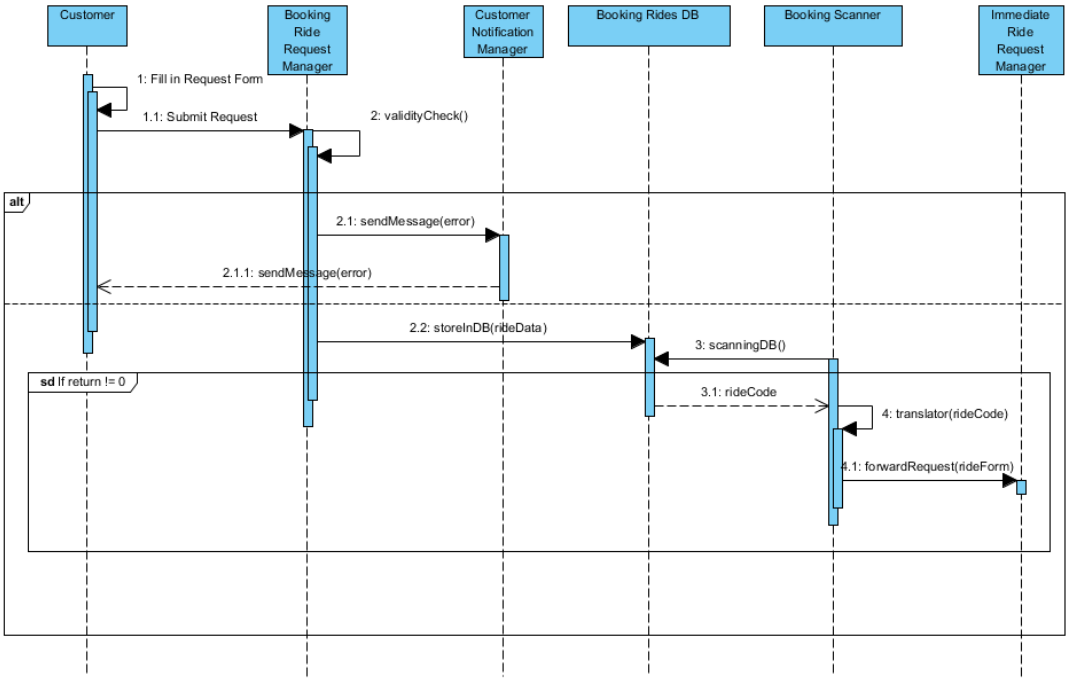
2.5.4 Status Management



2.5.5 Sharing



2.5.6 Booking


Customer
Booking Ride Request Manager
Customer Notification Manager
Booking Rides DB
Booking Scanner
Immediate Ride Request Manager
1: Fill in Request Form
1.1: Submit Request
2: validityCheck()
alt
2.1: sendMessage(error)
2.1.1: sendMessage(error)
2.2: storeInDB(rideData)
3: scanningDB()
sd If return != 0
3.1: rideCode
4: translator(rideCode)
4.1: forwardRequest(rideForm)

2.6 Component Interfaces

2.6.1 Registration Manager

```
public interface RegistrationManager{
    public RegistrationManager();
    public Message receiveInput(Form form);  //it returns a
    message with the operation details to the user
    public int SecurityCheck(Form form);
    private int putInDB(String userData); //it returns a control
     value to check the correctness of the operation
    }
```

2.6.2 Driver Login Manager

```
public interface DriverLoginManager{
    public DriverLoginManager();
    public Message receiveInput(String username, String password
    , String taxiCode); //it returns a Message with the details
    of the login operation
    private int checkInDB(String userCredentials); //it returns
    0 if it doesn't find corrispondances in the DB, 1 otherwise
    }
```

2.6.3 Customer Login Manager

```
public interface CustomerLoginManager{
    public CustomerLoginManager();
    public Message receiveInput(String username, String password
    ); //it returns a Message with the details of the login
    operation
    private int checkInDB(String userCredentials); //it returns
    0 if it doesn't find corrispondances in the DB, 1 otherwise
    }
```

2.6.4 Immediate Ride Request Manager

```
public interface ImmediateRideRequestManager{
    public ImmediateRideRequestManager();
    public int validityCheck(ImmediateRideForm request); //it
    checks the validity of the input
    private int storeInDB(String immediateRequestCode); //it
    puts the generated ride code in the DB and also returns a
    value to check the correctness of the operation
    public void forwardRequest(RideForm request); //it forwards
    details of the request to the Queues Manager
    }
```

2.6.5 Booking Ride Request Manager

```
public interface BookingRideRequest{
    public BookingRideRequest();
    public int validityCheck(BookingRideForm request);
    private int storeInDB(String bookingRequestCode);
    pubblic Message sendMessage(); //it sends messages to the
    Notification Manager
    }
```

2.6.6 Booking Scanner

```
public interface BookingScanner{
    public BookingScanner();
    public String scanningDB(); //it returns the code of a
    request that is approaching to the appointed time
    public RideForm translator(String requestCode); //it
    translates the request code into request form
    public int forwardRequest(RideForm rideForm); //it forwards
    the request to the Immediate Ride Request Manager
    }
```

2.6.7 Sharing Ride Request Manager

```
public interface SharingRideReuestManager{
    public SharingRideRequestManager();
    public int validityCheck(SharingRideForm request);
    public int storeInDB(String sharingRequestCode);
    public int forwardRequest(RideForm rideForm); //it forwards
    the request to the Immediate Ride Rquest Manager or to the
    Sharing Ride Viewer
    }
```

2.6.8 Rides Sharing Viewer

```
public interface RidesSharingViewer{
    public RidesSharingViewer();
    private List[String] scanningDB(SharingRideForm request); //
    it receives the request from the SharingRideRequest Manager
    and returns a list of the request codes that match with the
    user request
    public int showList(List[String] SharingRideSet); //it shows
     the found list to the user and returns also a value used to
    check the correctness of the operation
    }
```

2.6.9 Customer Notification Manager

```
public interface CustomerNotificationManager{
    public CustomerNotificationManager();
    public void ShowMessage(Message message); //it receives and
    forwards all the messages that the system needs to show to
    the user
    }
```

2.6.10 Customer Decision Manager

```
1 public interface CustomerDecisionManger{
2     public CustomerDecisionManager();
3     public int driverForwarding(Decision decision); //it
    forwards the user decision to the associated driver
4     public void forwardToSystem(Decision decision); //it
    forwards the user decision about a request to the system
5     public int paymentForwarding(Decision decision); //it
    forwards the user decision to the Payment Manager to confirm
    the transaction, cancel it or cancel a reservation
6     }
```

2.6.11 Payment Manager

```
1 public interface PaymentManager{
2     public PaymentManager();
3     public void manageTransaction(Decision decision); //it
    receives the user decision for the payment
4     public int sendMessage(Message message); //it sends a
    message to the Notification Manager to inform the user about
    his payment status and also returns a value to check the
    correctness of the operation
5     }
```

2.6.12 Availability Manager

```
1 public interface AvailabilityManager{
2     public Availability Manager();
3     public void receiveStatus(Status status, DriverCode
    driverCode); //it receives the driver code and the new driver
     status
4     public Position receivePosition(DriverCode driverCode); //it
     receives and saves the position of driver
5     public int forwardDriverStatus(Status status, DriverCode
    driverCode, Position position) //it forwards the data to the
    Queues Manager
6     }
```

2.6.13 Position Tracking Manager

```
1
2 public interface PositionTrackingManager{
3     public PositionTrackingManager();
4     public Position readingPosition(); //it reads and saves the
    driver position
5     public int sendPosition(Position position); //it forwards
    the driver position to the Availability Manager and it
    returns a value to check the correctness of the operation
6     }
```

2.6.14 Top Queue Extractor

```
public interface TopQueueExtractor{
    public TopQueueExtractor();
    public DriverCode extractHead(Queue queue);
    public int forwardRequest(DriverCode driverCode, RideForm
    request);
    }
```

2.6.15 Queues Manager

```
public interface QueuesManager{
    public QueuesManager();
    public Queue queueSelection(Position position); //it returns
     the queue code associated to the position received in input
    public void manageQueue(Queue queue, Status status,
    DriverCode driver); //it updates the driver position in the
    queue
    public int forwardRequest(Queue queue, RideForm request); //
    it forwards the request to the Top Extractor
    public void handleDecision(Queue queue, DriverCode driver,
    Decision decision); //it updates the driver position in case
    of rejection of request
    }
```

2.6.16 Driver Notification Manager

```
public interface DriverNotificationManager{
    public DriverNotificationManager();
    public void forwardCustomerDecision(Decision decision,
    DriverCode driver);
    public void forwardRequest(RideForm request, DriverCode
    driver);
    }
```

2.6.17 Driver Decision Manager

```
public interface DriverDecisionManager{
    public DriverDecisionManager();
    public void forwardToCustomer(Decision decision); //it
    forwards the driver decision to the associated customer
    through the CustomerNotificationManager
    public void forwardToSystem(Decision decision, DriverCode
    driverCode); //it forwards the driver decision about a
    request to the system
    }
```

2.7 Selected Architectural Styles And Patterns

2.7.1 Architectural Styles

In this subsection are described the most important rules that identify the kinds of components and connectors that are used to compose the system.

2.7.1.1 Message Bus Architectural Style (Communication)

Message bus architecture is based on a software system able to receive and send messages using one or more communication channels. The implementation consists of common schemas of communication for the applications. This communicational architecture style provides the ability to compute complex processing logic by combining a set of smaller operations. Moreover it's possible to modify the processing logic since the interaction with the bus is based on a common schemas and commands, so we can decide to change the used logic which the system process messages with. By using a massage-based communication model, it's guaranteed the interaction with applications developed for different environments. The main advantages of this kind of communication architectural style are:

- Extensibility: The system can add or remove applications from the bus without alternating the existing ones.
- Low complexity: Each application has to communicate only with the bus.
- Scalability: Multiple instances of the same application can be attached to the bus in order to handle multiple requests.

2.7.1.2 Object-Oriented (Structure)

Object-oriented architecture is a design paradigm based on the division of responsibilities for the system into individual reusable and self-sufficient objects, each containing the data and behaviour relevant to the object. This structure sees the system as a series of cooperating objects, and not ad a set of procedural instructions and routines. The objects communicate through interfaces, by calling methods and by sending and receiving messages. Object-oriented architectural style provides a good level of:

- Abstraction: It reduces complex operations into a generalization.
- Composition: Objects can be assembled from other objects.
- Inheritance: Objects can inherit from other objects or override some functionality.

- Encapsulation: Objects expose functionality only through method, properties and events, and hide the internal details (variables).
- Polymorphism: It allows an object to be overridden and change his behaviour by implementing new types that are interchangeable with the existing object.

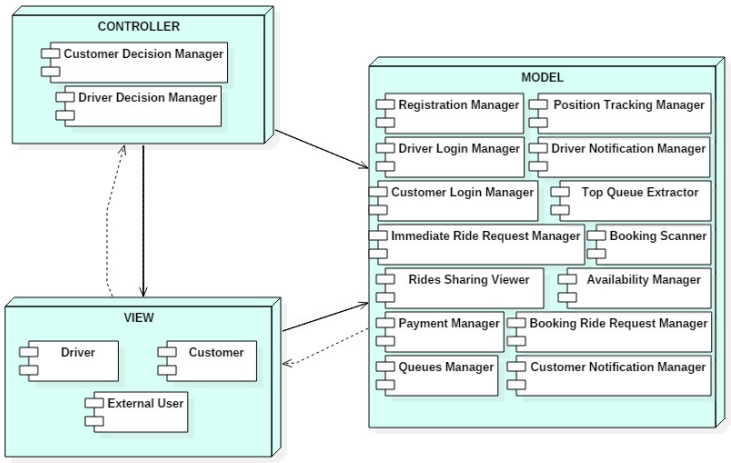
2.7.2 Architectural Pattern

This subsection expresses a fundamental organization schema for the software system. It provides a set of rules and guidelines for organizing the relationships between all subsystem.

2.7.2.1 Model - View - Controller

Model - View - Controller (MVC) is the design pattern chosen for the architecture of the system. It is a largely used pattern by many languages and implementation frameworks. The component of the application are clean separated between three main components:

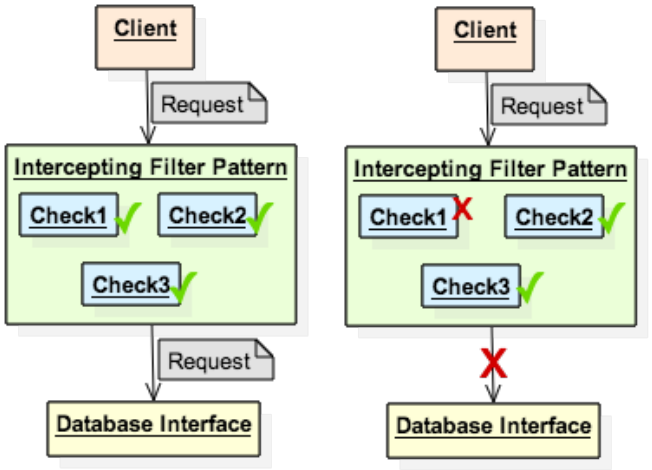
- Model: This component provides functional core of the application, responds to state queries, notifies views of changes. It contains the business logic of the application.
- View: The responsibilities of this component are the creation and initialization of its associated controllers, displaying the information to the user and retrieving data from the model. It represents the user interface (UI)
- Controller: This component defines application behaviour, maps user actions to model updates, accepts user input as events and translates them.



2.7.3 Design Pattern

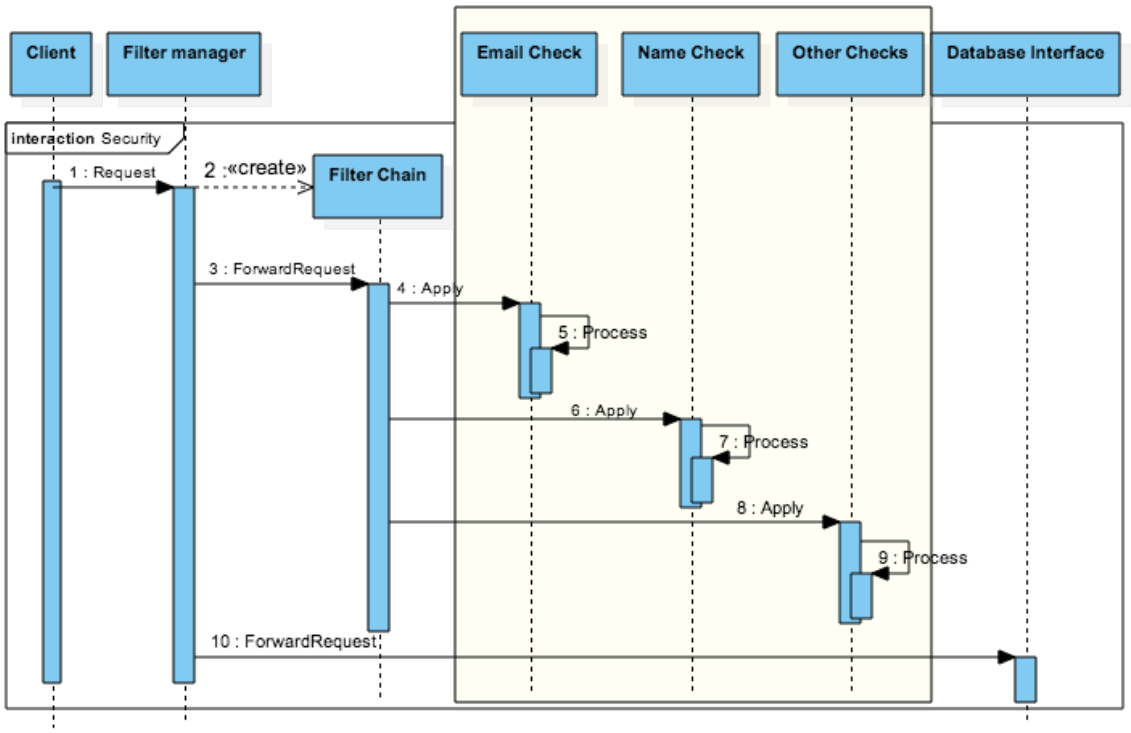
2.7.3.1 Intercepting Filter Pattern

The Intercepting Filter Pattern is used to pre-processing the client requests in order to check their correctness and compatibility with some test imposed by the system (i.e. The request must contain a feasible date). Furthermore this pattern is used when pre-processing of a client Web request and response is required, for instance the client must pass several tests before signing in (i.e. input data are correct, all the mandatory inputs must not to be empty). The classical solution of a if/else series is replaced by the intercepting filter pattern in a flexible manner and it allows the developer to add or remove processing component in the future.



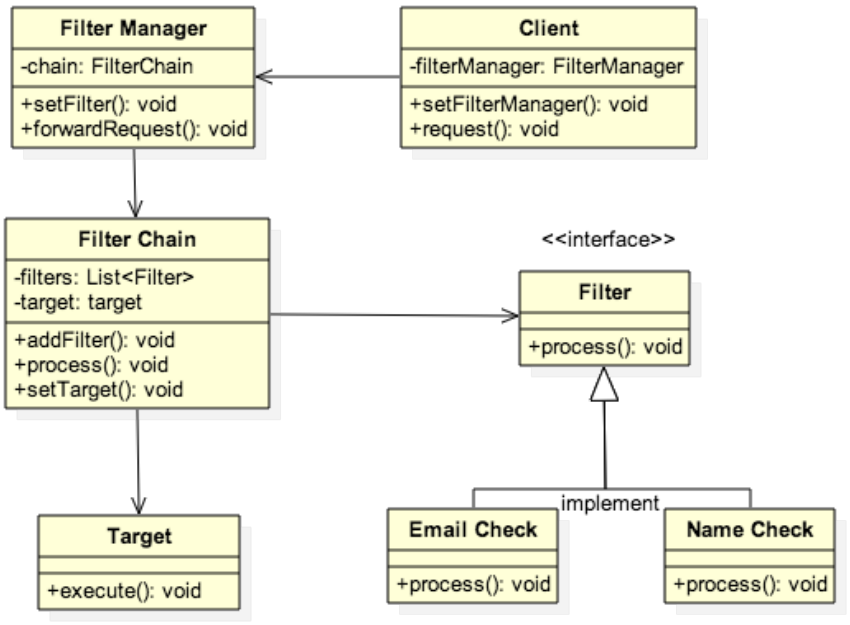
Intercepting Filter Pattern Sequence Diagram

This is the sequence diagram of the Intercepting Filter Pattern. The example shows the pattern used by the system to check the filled form by the client before signing in.



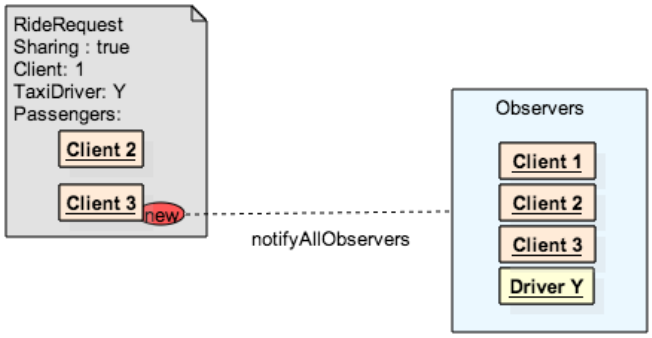
Intercepting Filter Pattern Class Diagram

Here it is the class diagram of the Intercepting Filter Pattern. The entities composing this design pattern are the Filter which performs certain task before the execution of the request, Filter Chain which carries multiple filters and execute them in a defined order, Target which is the request handler, Filter manager which manages Filter and Filter Chain and the Client who forwards the request to the target object.



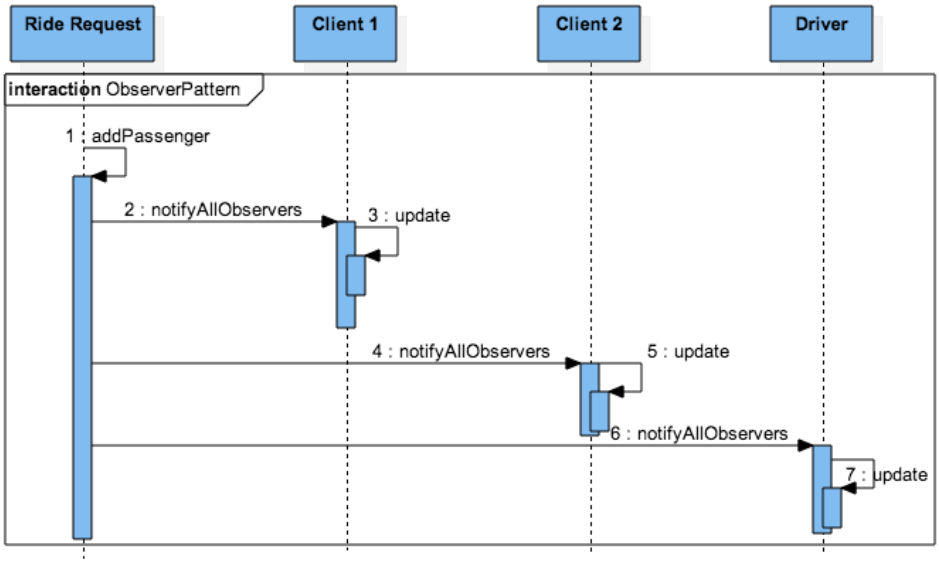
2.7.3.2 Observer Pattern

The Observer Pattern is used to implement the event handing system. When an event is triggered, the system acts in a different way according to the event. This pattern is also a main component in the Model-View-Controller (MCV) architectural pattern. An useful example where Observer Pattern is used is for handling the one-to-many relationship between a shared ride request and the clients. When a new customer joins the ride, the system has to notify all the clients who share the ride, and the related taxi driver. This mechanism can be used for different situation.



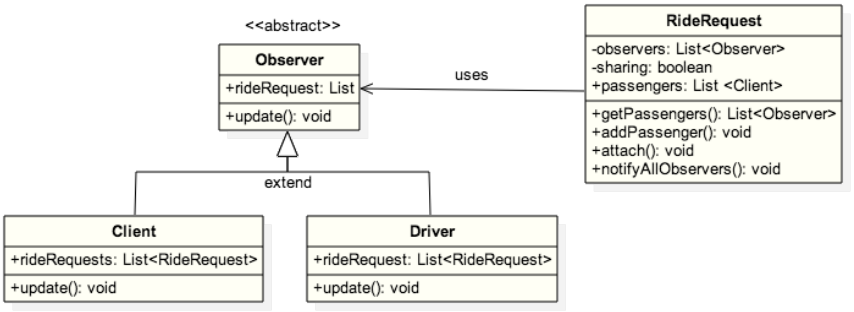
Observer Pattern Sequence Diagram

In this subsection there is the sequence diagram of the Observer Pattern. The example shows the pattern used by the system to notify all the clients who share the same ride when a new client join the request. It's also notified the taxi driver.



Observer Pattern Class Diagram

In this subsection there is the class diagram of the Observer Pattern. It uses four actor classes: RideRequest which contains a list of observers to notify when a passengers is added or removed by the shared ride; Observer which is an abstract class extended by Client and Driver which perform a task when they are notified by the RideRequest.



Chapter 3

Algorithm Design

3.1 Security Check

Algorithm: Security check of the signing in form.
Require: This procedure requires the filled form by the user during the registration.
Ensure: This procedure returns -1 if the form contains mistakes or the form right.

```
1 procedure SecurityCheck
2   if form.getName() is empty then return -1
3   [...]
4   if form.getEmail not contains (@ and .) then return -1
5   else return form
```

3.2 Driver Availability

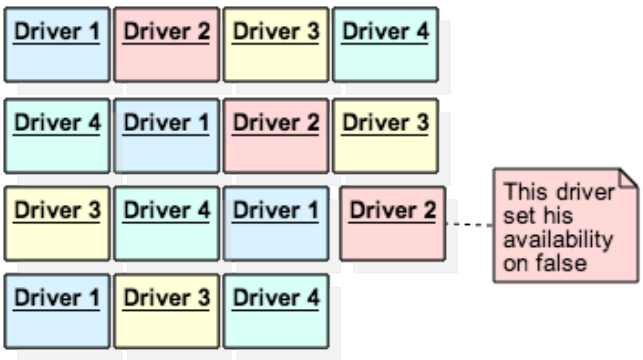
Algorithm: Manage the availability.
Require: This procedure doesn't require any particular element.
Ensure: This procedure ensures to manage driver availability and his priority into the queue.

```
1 procedure DriverAvailability
2   if taxiDriver.switchAvailability() = true then driverQueue.
      enqueue(taxiDriver)
3   if taxiDriver.switchAvailability() = false then
4     deleteFromQueue(taxiDriver, driverQueue)
```

3.3 Remove Driver

Algorithm: Delete Element From Queue.
Require: This procedure requires a specified queue and an element of it.
Ensure: This procedure ensure to delete the specified element from the queue.

```
procedure DeleteFromQueue
  wait(mutex)
  for  1 to queue.length
    x = queue.dequeue()
    if x == element then queue.dequeue()
    else queue.enqueue(x)
  signal(mutex)
```



3.4 Ride Request

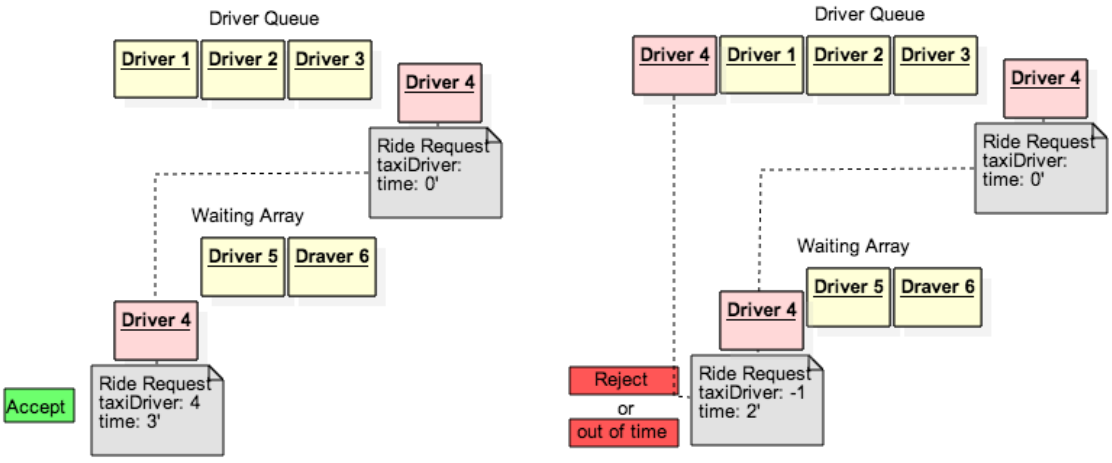
Algorithm: Ride Request.
Require: This procedure requires a customer ride request in order to hail a taxi.
Ensure: This procedure ensures to notify the right taxi driver in order to satisfy the request of the client and the priority of the drivers. If no taxi driver is available the procedure returns a negative value.

```
procedure RideRequest
    if driverQueue.length < 0 then return -1
    else NotifyDriver(rideRequest)
  loop:
    if rideRequest.taxiDriver is empty then goto loop
    if rideRequest.taxiDriver = -1 then NotifyDriver(rideRequest
    )
    else payment(rideRequest)
```

3.5 Driver Notification

Algorithm: NotifyDriver.
Require: The procedure requires a queue containing all available taxi drivers and an array able to contain taxi drivers.
Ensure: The procedure ensures to manage the request by managing the taxi driver queue. If a taxi driver rejects the request, the system set his new position inside the queue. But if the taxi driver accepts the request the system removes him from the queue because of his unavailability.

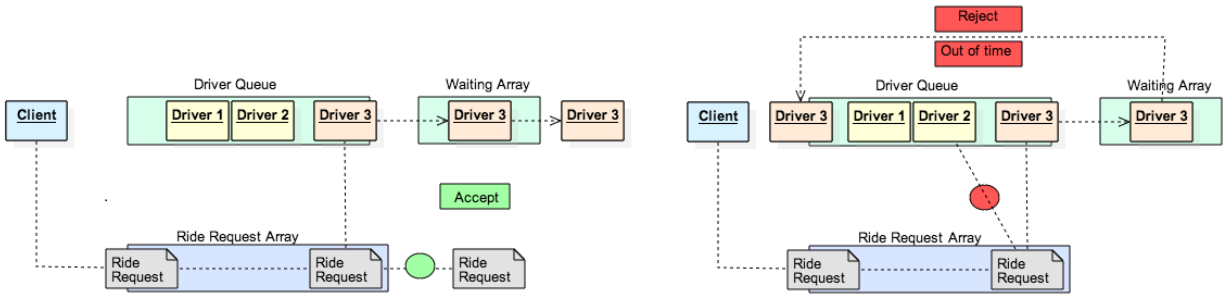
```
1  procedure NotifyDriver
2      x = driverQueue.dequeue()
3      waitingArray.add(x)
4      displayRequest(x, rideRequest)
5    loop:
6      if rideRequest is Accepted
7      then
8        rideRequest.taxiDriver = x
9        return
10     else if rideRequest.time > 2 OR rideRequest is Rejected
11       then
12         waitingArray.removeElement(x)
13         driverQueue.enqueue(x)
14         rideRequest.taxiDriver = -1
15         return
16       if rideRequest.taxiDriver is empty then goto loop
```



3.6 Request Manager

Algorithm: Management of the sent requests by the clients.
Require: This procedure requires two queues, the first one containing the available taxi driver, the other one for the requests coming from the clients and an array to keep track of all the forwarded requests to the relative drivers.
Ensure: This procedure ensures to manage all the requests coming from the clients and to forward them to the drivers according to their priority into the driver queue. If a taxi driver rejects the request or runs out of time to accept it, the system will send it to the next available driver. If the request is accepted, it will be removed from the queue. Moreover it's guaranteed that if a request is not satisfied before the departure, the procedure will return a negative value.

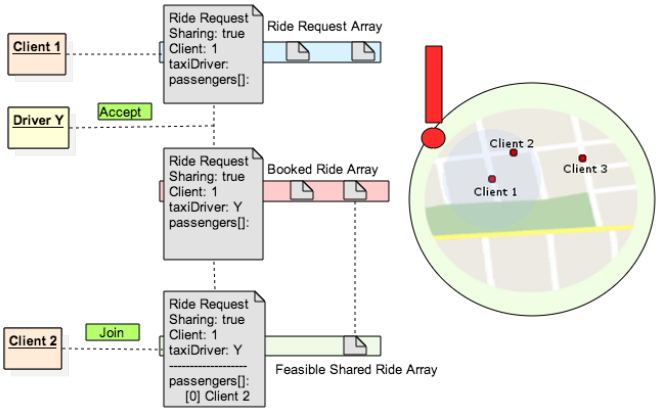
```
procedure RequestManagement
  rideRequestArray.add(requestN)
loop:
  requestN.time = 0
  notifyDriver(requestN)
  if requestN is Rejected and requestN.Data < now then goto loop
  else if requestN.data > now
  then
    rideRequestArray.removeElement(requestN)
    return -1
  rideRequestArray.removeElement(requestN)
  return
```



3.7 Ride Booking

Algorithm: Booked Ride Requests Management.
Require: This procedure requires three different arrays all able to contain ride requests. Ride Request array to contain the request which the taxi drivers have to respond to, Booked Ride Array to keep track of all the booked ride by the clients, and Feasible Shared Ride Array with all the possible ride requests in sharing to join.
Ensure: This procedure ensures to manage all booked accepted ride requests and it gives the clients the possibility to display all shared rides if their requests has similar starting point, destination and the same data. If there is no feasible ride to join, the request will be saved in sharing mode. It's also guaranteed that if no taxi driver is available for that ride before the departure, the procedure will return a negative value.

```
1  procedure BookRide
2    if requestManagement(rideRequest) < 0 then return -1
3    if rideRequest.Sharing is true
4    then
5      for i from 0 to bookedRideArray.length
6        if bookedRideArray[i].sharing is true
7        then
8          distance_between_destinations = google.maps.distance(
     rideRequest.destination, bookedRideArray[i].destination)
9          distance_between_starting_points = google.maps.distance(
     rideRequest.startingPoint, bookedRideArray[i].startingPoint)
10         if distance_between_starting_points < 500m AND
     distance_between_destinations < 500m AND bookedRideArray[i].
     data == rideRequest.data
11         then
12           feasibleSharedRideArray.add(bookedRideArray[i])
13         if feasibleSharedRideArray is empty then bookedRideArray
     .add(rideRequest)
14       else bookedRideArray.add(rideRequest)
15       return
```

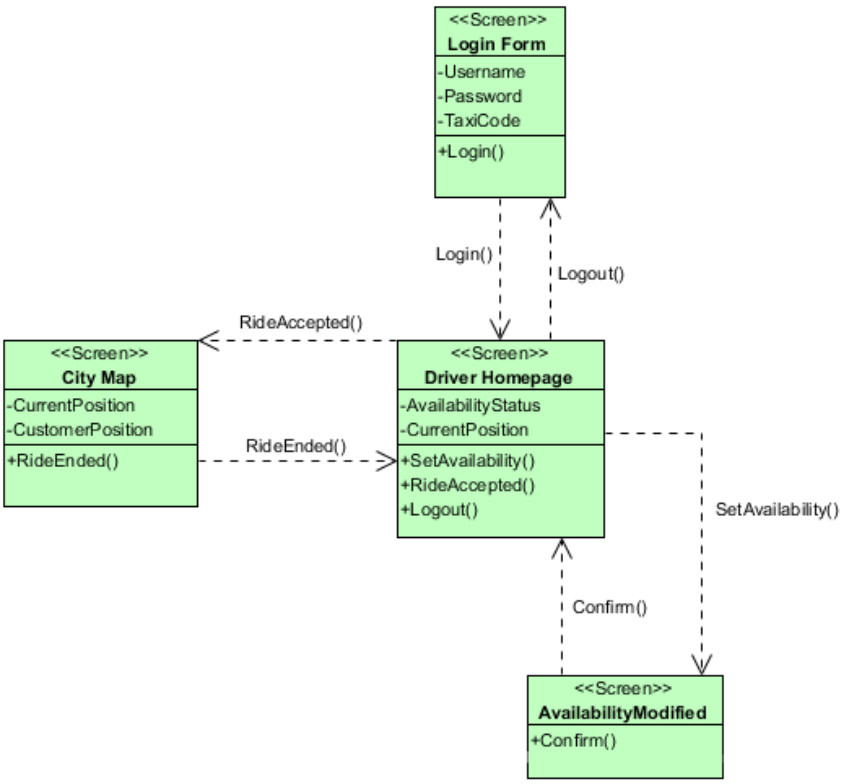


Chapter 4

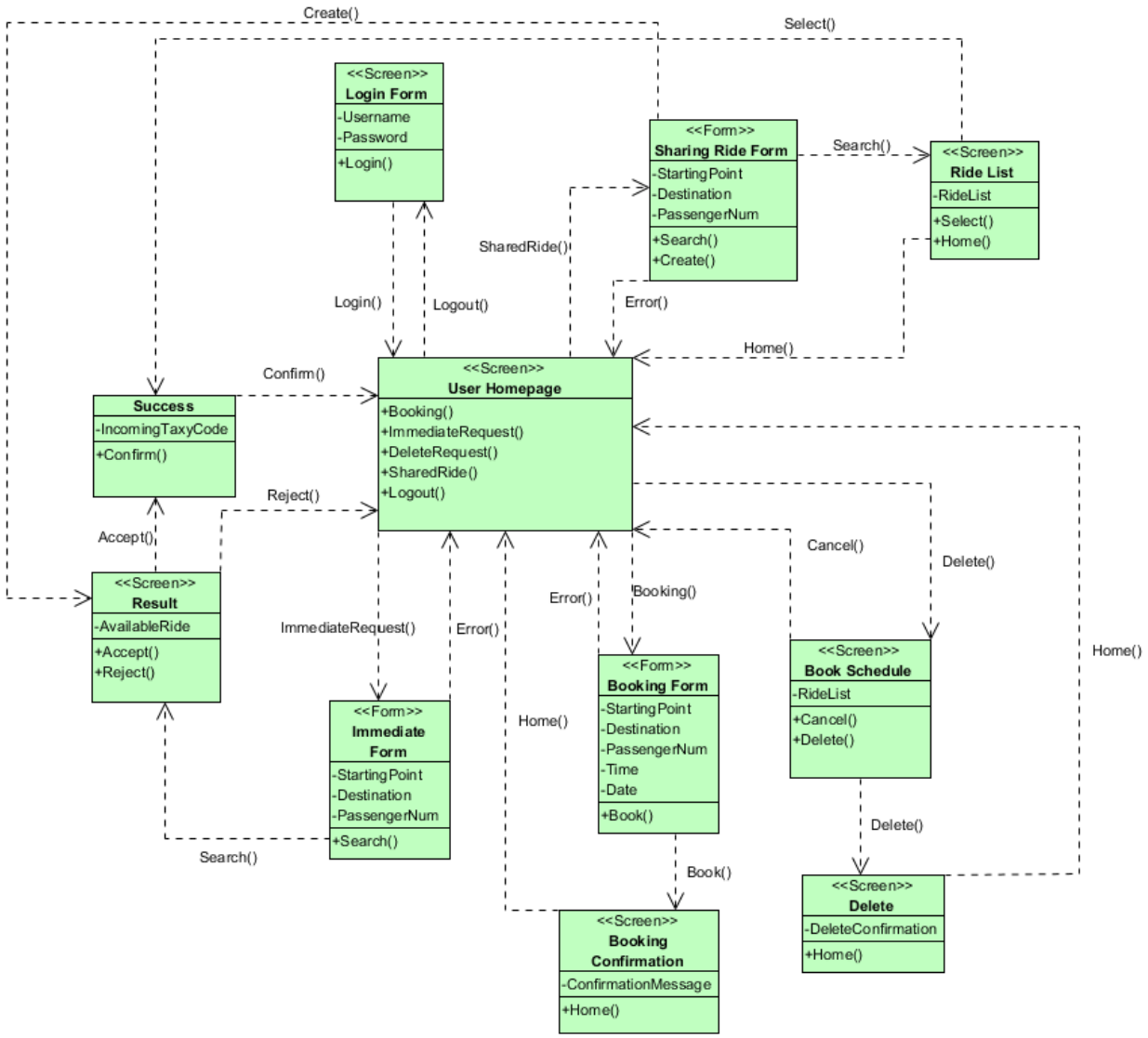
User Interface Design

The most important mockups of the application have already been presented in the RASD document. Here two UX diagrams provide an high-level explanation of how a driver and a customer can move among the main functions of myTaxiService.

4.1 Driver Interface



4.2 Customer Interface



Chapter 5

Requirements Traceability

Requirement	Component
The system shall provide a sign up interface asking for first name, surname, born date, address, telephone number, username, password and credit card ID for each customer that want to use the service.	Registration Manager
The system shall be able to recover all personal data of users that register through their social network accounts.	Registration Manager
The system shall save all registration data of users in his database.	Registration Manager
The system shall provide a sign in interface through which a customer can login with his username or e-mail, and his password.	Customer Login Manager
The system shall provide a function that lets users login with their social network accounts.	Customer Login Manager
The system shall provide a sign up interface through which a taxi driver can register giving his name, surname, e-mail address, telephone number, licence number, username, password, vehicle number plate, vehicle brand, vehicle model, IBAN, and the maximum number of passengers that he can carry simultaneously.	Registration Manager
The system shall assign to each driver a unique taxi code.	Queues Manager
The system shall save all registration data of taxi drivers in his database.	Registration Manager
The system shall provide a sign in interface that lets a taxi driver login with his username, password and taxi code.	Driver Login Manager

Requirement	Component
The system shall modify the saved data of an user that wants to change his password removing his old one.	Registration Manager
The system shall send to the user an e-mail with the link through which he can set a new password.	Registration Manager
The system shall save in the database the new password of the user.	Registration Manager
The system shall provide an interface through which an user can specify the starting point and the destination of a ride.	Immediate Ride Request Manager
The system shall forward users' requests to the first taxi of the queue of the zone that the selected starting point belongs to.	Top Queue Extractor
The system shall forward a user's request to the next taxi in the queue if the first one rejects it or after a timer expiration (2 minutes).	Top Queue Extractor
The system shall answer a user request with the awaiting time for his taxi and the ride cost, and it must let him accept or reject that ride.	Customer Notification Manager
The system shall entrust the transactions management to an external system, in the event that the customer confirms the ride.	Payment Manager
The system shall forward user's decision to the taxi driver.	Driver Notification Manager
The system shall provide an interface through which an user could reserve a ride by specifying: starting point, destination, number of passengers, date and time of the ride.	Booking Ride Request Manager
The system shall check the time chosen by the user; if it is at least two hours after the reservation, it must save all the data of the ride. Otherwise it must reject the reservation.	Booking Ride Request Manager
The system shall provide an interface that lets a taxi driver change his current status between available and not available.	Availability Manager
The system shall add a taxi driver at the end of the queue of his zone when his status becomes available.	Queues Manager
The system shall provide a function that lets each taxi driver accept or reject a ride asked by an user.	Driver Decision Manager

Requirement	Component
The system shall update the queue moving in last position an available taxi driver that rejects a ride.	Queues Manager
The system shall provide an interface that lets an user join an existing shared ride or create a new one, by specifying his position, the desired destination and the number of passengers.	Sharing Ride Request Manager
The system shall provide to each user that search an existing shared ride a list of all available rides, specifying starting point, leaving time, number of confirmed passengers and cost of the ride (depending on the number of passengers).	Rides Sharing Viewer
The system shall provide an interface that lets a user select his favourite shared ride.	Rides Sharing Viewer
The system shall keep the client up to every price modification.	Customer Notification Manager
The system shall forward all the new information to the selected taxi driver.	Driver Notification Manager
The system shall manage the possibility to cancel client's reservation	Customer Decision Manager

Chapter 6

References

The documents used in order to redact this work have already been listed in the first chapter.
We have also used these software:

- TeXstudio: to write this Design Document
- Visual Paradigm Community Edition 12.2: to create Sequence and UX Diagrams.
- StarUML: to create most of other diagrams ad images.
- Gimp: to manipulate some images.