myTaxiService

Design **D**ocument

A.Y. 2015/2016

Politecnico di Milano

Version 1.0

Cattaneo Michela Gaia, matr. 791685
Barlocco Mattia, matr. 792735

December 4, 2015

Contents

1 Introduction

1.1 Purpose

This document represents the Design Document, whose main goal is to describe the overall system architecture and to show the technical design decisions made, with the support of schema and diagrams.
It delineates how the software system will be structured in order to satisfy the requirements defined in the Requirement Analysis and Specification Document (RASD), each one translated into a representation of components, interfaces, and data necessary in the implementation phase.
This document is addressed to the project development teams, technical architects, database designers and testers, as long as it has useful guidelines and a specific description of the design implementation of the system.

1.2 Scope

The system aims at simplifying the access of the passengers and guaranteeing a fair management of taxi queues.
The server listens for the requests of the clients, which can be taxi drivers or users and it access to their GPS positions.
The user clients are able to request a taxi and, if logged into the system, they can also make reservations and choose the payment method they prefer. On the other hand, taxi driver clients can change their status from available to unavailable, accept or decline the requests of the users and see their position in the queue of the area they are in.
The server manages the notifications to forward to the clients, for example the arrival of a taxi for a user or the requests of the users for the taxi drivers.

1.3 Definitions, Acronyms, Abbreviations

- **Definitions**

- **User**: a person who requests a service from the system. It can be a visitor or a passenger.

- **Visitor**: a person who is not registered in the application.

- **Passenger**: a person who is registered in the application.

- **Taxi driver**: a taxi driver who access the application with a specific ID.

- **Request**: the request of a taxi in a certain area and position in the city made by a user.

- **Reservation**: the reservation of a taxi in a certain area, place and time that can be made only by passengers.

- **Acronyms and abbreviations**

- **RASD**: Requirement Analysis and Specification Document

- **DBMS**: Database Management System

- **JEE**: Java Enterprise Edition

- **API**: Application Programming Interface

- **UML**: Unified Modeling Language

- **RMI**: Remote Method Invocation

- **HTTP**: HyperText Transfer Protocol

- **UX**: User eXperience

1.4 Reference documents

- myTaxiService Requirement Analysis and Specification Document (RASD)

1.5 Document structure

This document is divided in six main parts:

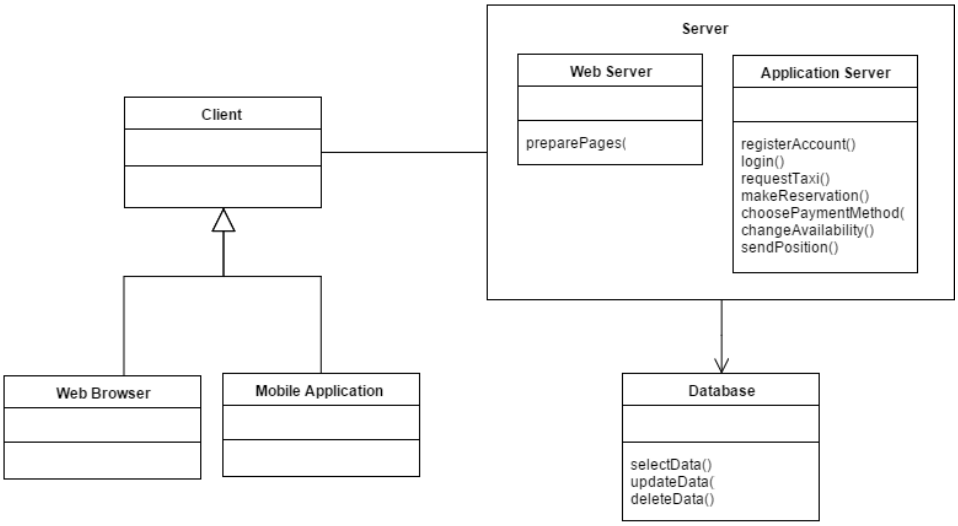
- **Introduction**: this section describes the document in general and its purpose.
- **Architectural Design**: this section specifies the architectural design part, giving information about the components involved and the architectural styles adopted.
- **Algorithm Design**: this section provides a general description of the main algorithms used during implementation.
- **User Interface Design**: this section outlines an overview on how the user interfaces of the system will look like.
- **Requirements Traceability**: this section explains the connection between the requirements already defined in the RASD and the design elements introduced in this document.
- **References**: this section contains the references to external documents used to redact this document.
- **Appendix**: this section provides information about the tools used to redact this document and how many hours of work each author has spent.

2 Architectural Design

2.1 Overview

This section deals with the specific description of the architectural choices made for the myTaxiService system. These decisions are supported by different diagrams, whose aim is to highlight and simplify the view of the components of the system, and explanations about the architectural styles adopted.
In the first subsections there is a representation of the components of the system with diagrams. Here are presented the high level components and their interaction, with a brief definition of the functionalities of the three components individuated, then these are unfolded and there is a more specific description of the components, the interfaces and their relationships and features.
The last subsection explains and justifies the architectural styles selected for the myTaxiService system, which consists in a client/server, event-based system, with a service-oriented, three-tier architecture (client, server and database).

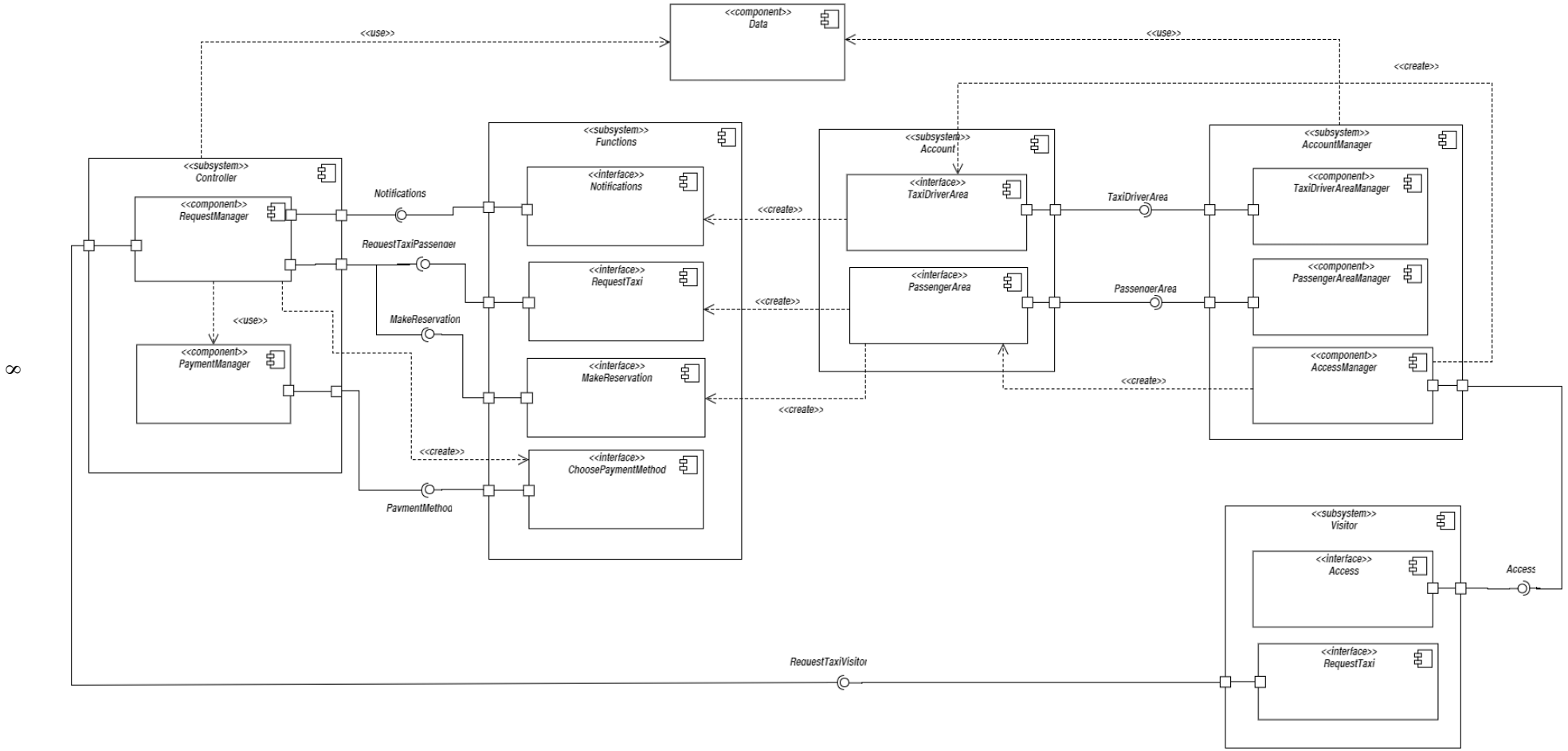
2.2 High level components and their interaction



- **Client**: this part runs on the client devices via a Web browser or the mobile application. It allows the users to insert and submit the data in the input forms, that are sent to the application server. On the other hand, the taxi drivers can send information about their availability to the server and the application client monitors their GPS position in order to move the taxi drivers to another queue if they change their area.
- **Application Server**: this part runs on the JEE server. It is composed of a web server part, which always listens to all the clients requests and is responsible for the creation of the faces and pages of the client interfaces. The application server, instead, contains the logical part of the application, collecting and managing the information from the clients and the database. In fact, it analyses the data coming from the clients and, according to the requests, it modifies or asks for the required information stored in the database, then it is able to answer the client request, sending it the result.
- **Database**: this part contains the database where all the application data are stored. It is not only accessed by the application server, but also by the administrators, who can, for example, directly add a taxi driver account to the database.

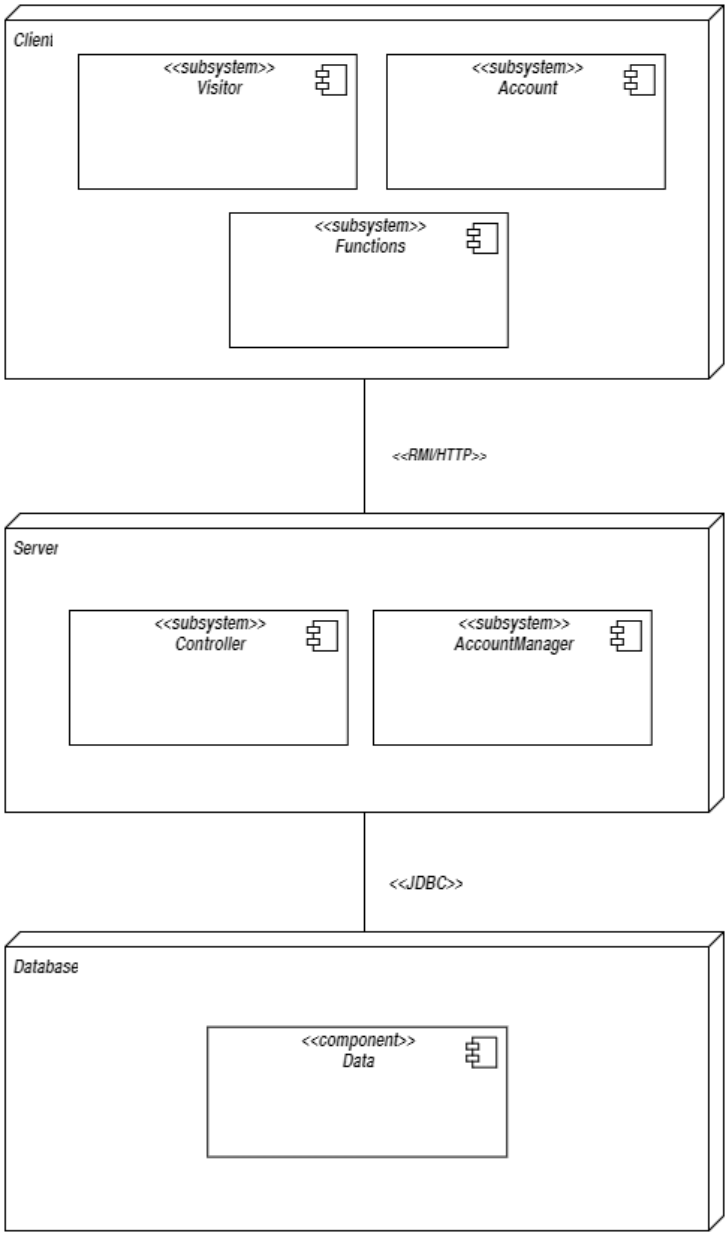
2.3 Component view

In the diagram below are showed the components that define the myTaxiService system architecture, which is composed by many subsystems and an external component which belongs to the database.
Starting from the left, in the first subsystem, the **Controller**, there are two components that manage the requests of the users and the payments.
The *RequestManager* component manages all the logic of the requests, getting the information from the users and forwarding them to the designated taxi driver. As it deals with several tasks, he needs more than one interface to be implemented: *RequestTaxiVisitor*, *Notifications*, *RequestTaxiPassenger* and *MakeReservation* are the required interfaces for this component. It also relates with other components: it uses the *PaymentManager* and the *Data*, in order to complete its operations, and creates the *ChoosePaymentMethod* interface, which is required by the *PaymentManager*.
The second subsystem is called **Functions**, as it provides the possible actions that the user can do while using the system. It is composed by four interfaces provided to the **Controller**: *PaymentManager*, *RequestTaxiPassenger*, *MakeReservation* and *Notifications*.
This last interface can be accessed from the taxi driver mobile application to see the notifications of the requests, in fact it is produced by the *TaxiDriverArea* interface. The *RequestTaxiPassenger* and *MakeReservation* interfaces, instead, are created by the *PassengerArea* interface, which is part of the subsystem **Account** with *TaxiDriverArea*. They are used by the passenger when he needs to request a taxi or make a reservation from his personal area.
Another subsystem is the **AccountManager** whose main goal is to manage the access of the users, forwarding them to the proper designated area. The *TaxiDriverAreaManager* requires the interface *TaxiDriverArea*, which is provided by the *TaxiDriverArea* interface in **Account**. The same relation is defined between the *PassengerAreaManager* component and the *PassengerArea*. The third component of this subsystem, the *AccessManager*, which is responsible for the log in and the registration of the users, creates an instance of *PassengerArea*, while requires the interface *Access*.
The last subsystem is the **Visitor**, which provides the interface *Access* to the *AccessManager* and another interface, *RequestTaxi* to the *RequestManager*. In fact, this subsystem represent the home page where the visitor can register or log in and request a taxi.


<<component>>
Data
<<use>>
<<use>>
<<create>>
<<subsystem>>
Controller
<<component>>
RequestManager
<<use>>
<<component>>
PaymentManager
Notifications
RequestTaxiPassenger
MakeReservation
<<create>>
PaymentMethod
<<subsystem>>
Functions
<<interface>>
Notifications
<<interface>>
RequestTaxi
<<interface>>
MakeReservation
<<interface>>
ChoosePaymentMethod
<<subsystem>>
Account
<<interface>>
TaxiDriverArea
<<interface>>
PassengerArea
<<create>>
<<create>>
<<create>>
<<subsystem>>
AccountManager
<<component>>
TaxiDriverAreaManager
<<component>>
PassengerAreaManager
<<component>>
AccessManager
TaxiDriverArea
PassengerArea
<<create>>
<<subsystem>>
Visitor
<<interface>>
Access
<<interface>>
RequestTaxi
Access
RequestTaxiVisitor

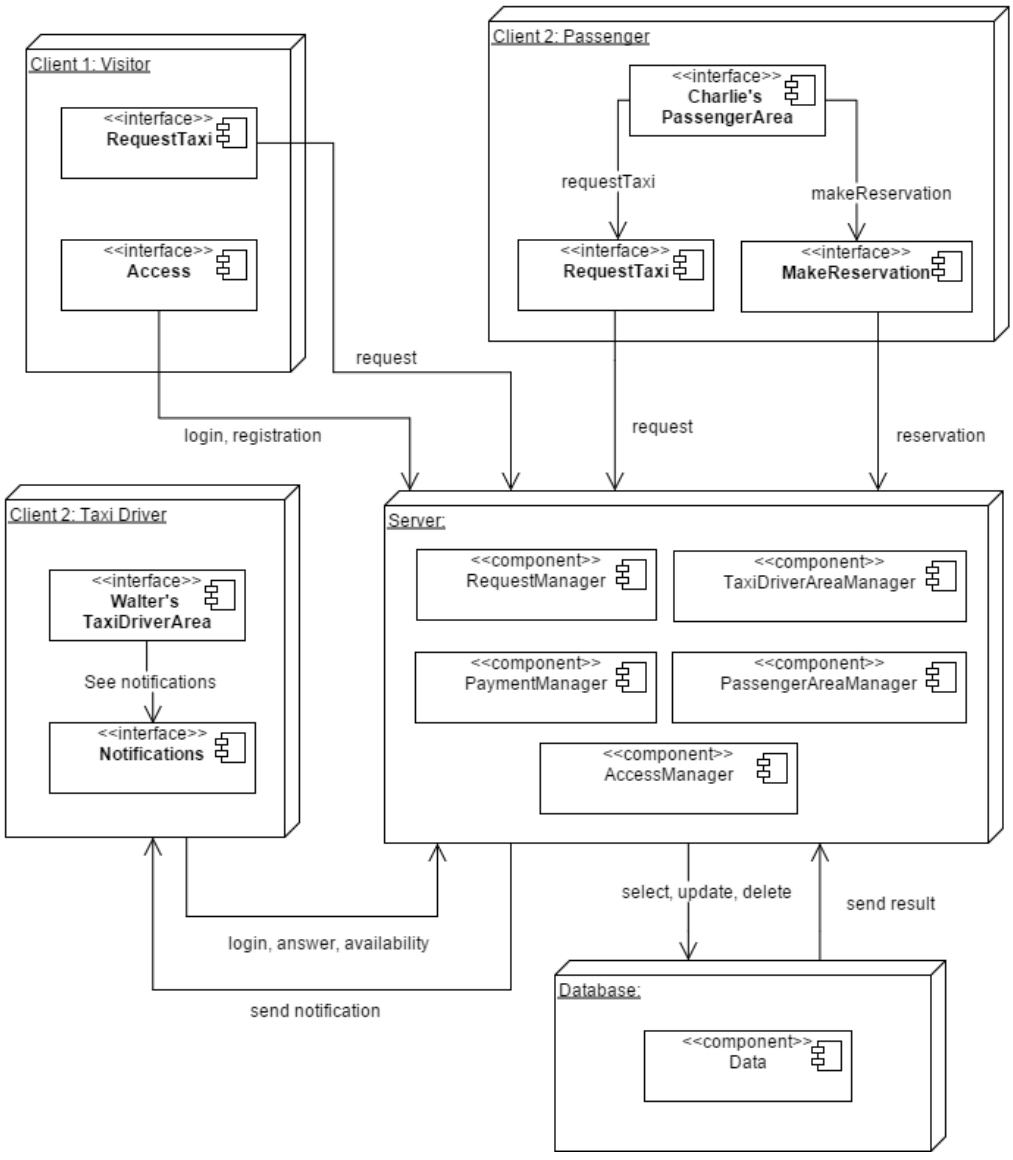
2.4 Deployment view

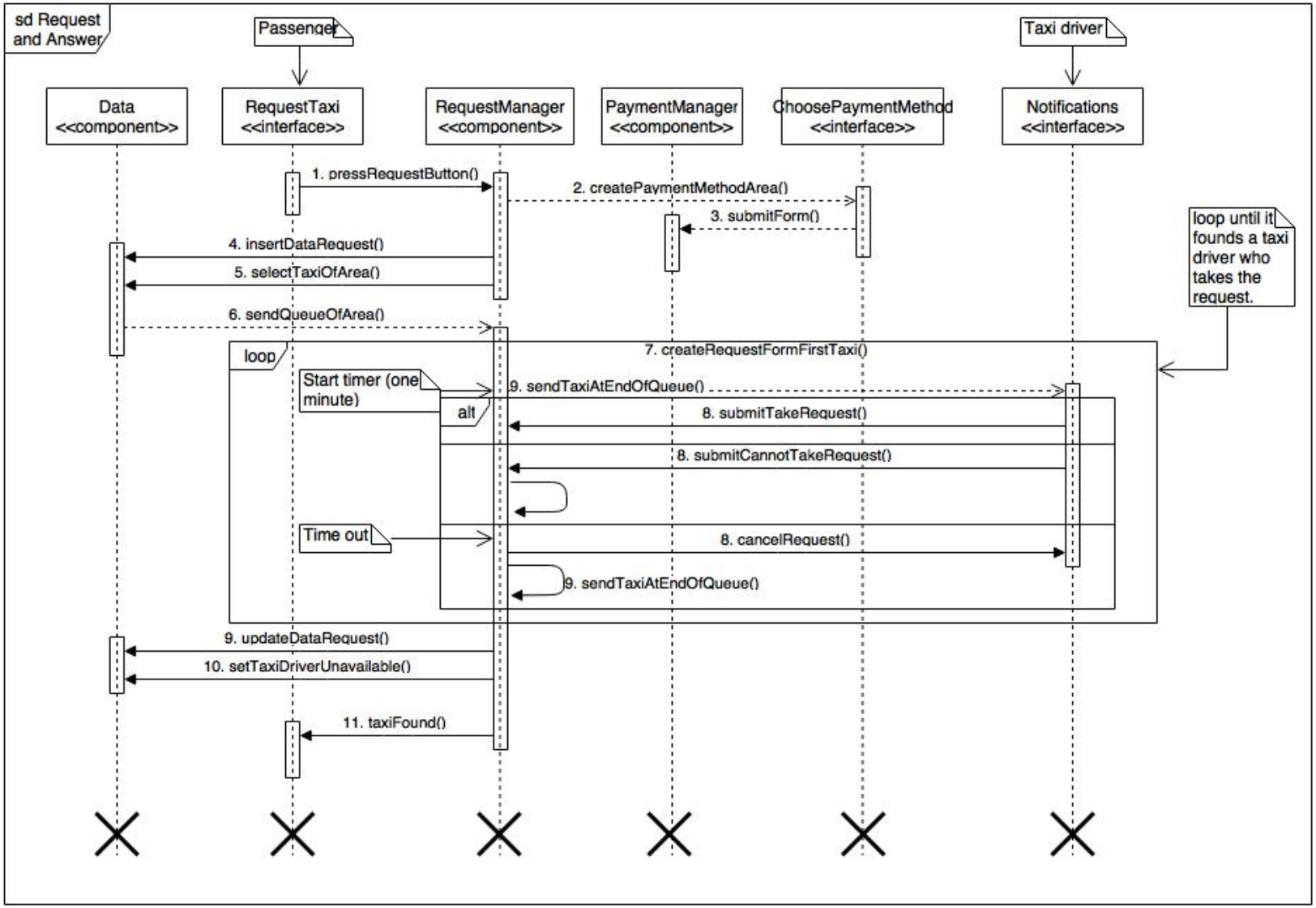
This diagram shows where the components presented in section 2.3 are deployed in the system.

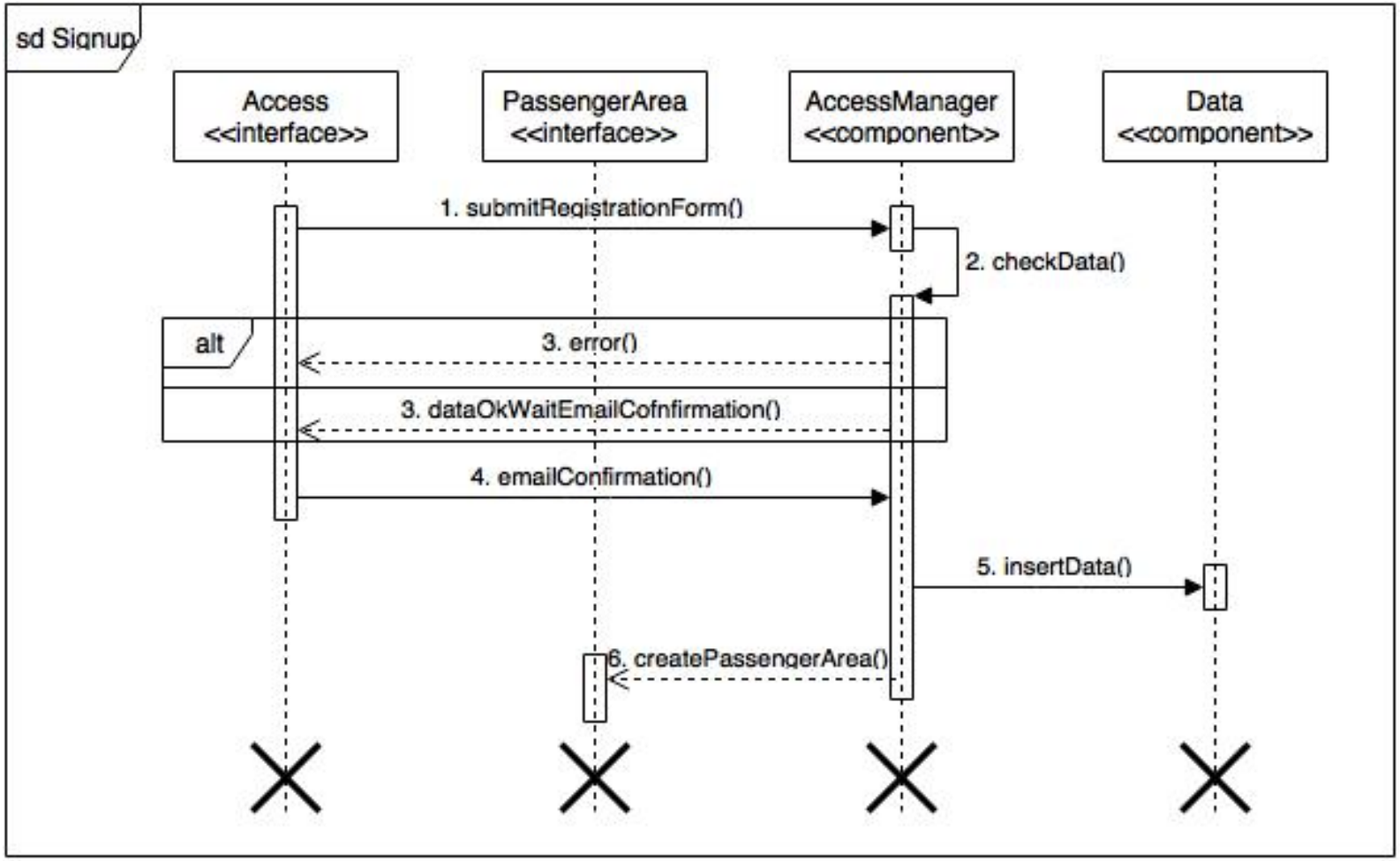


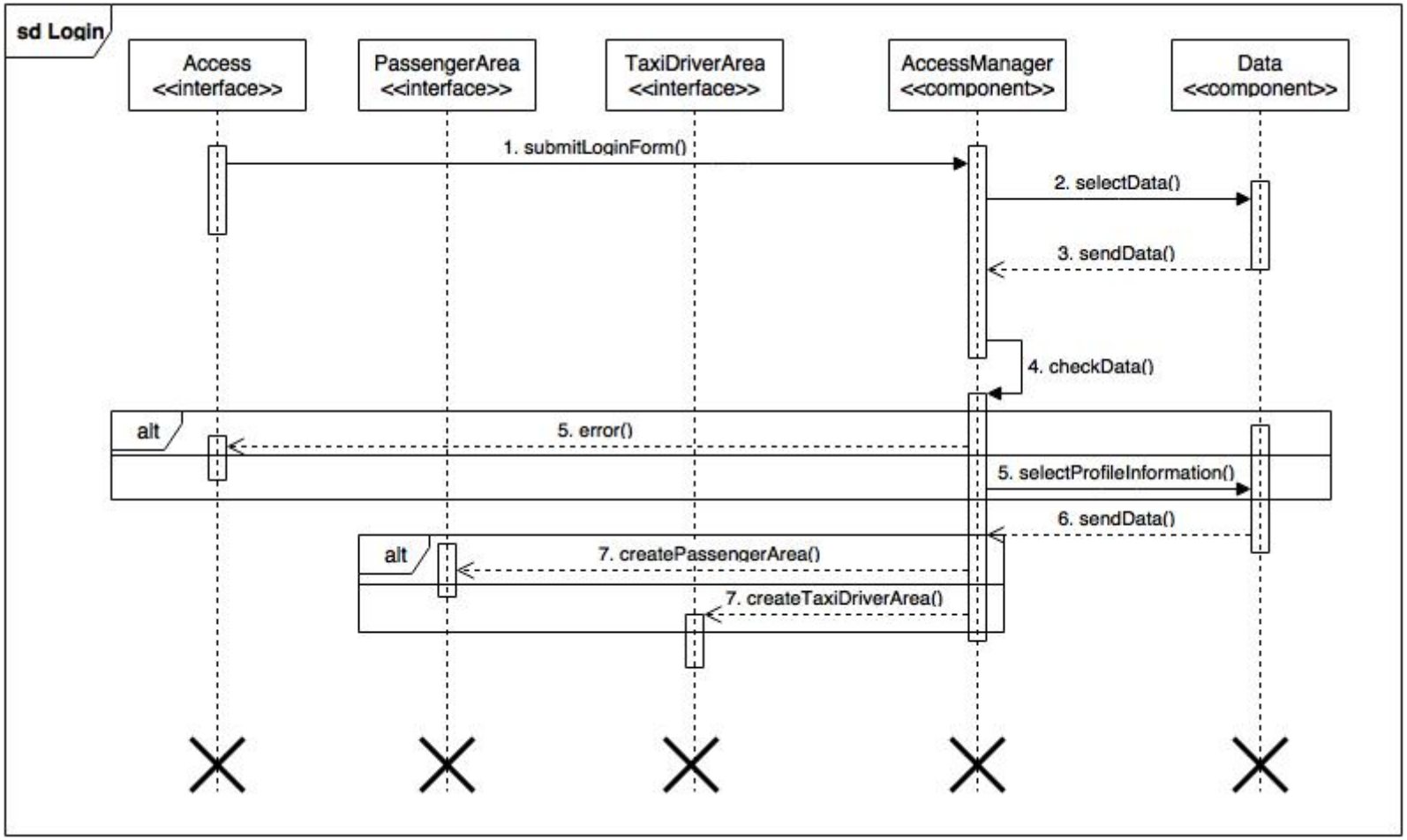
2.5 Runtime view

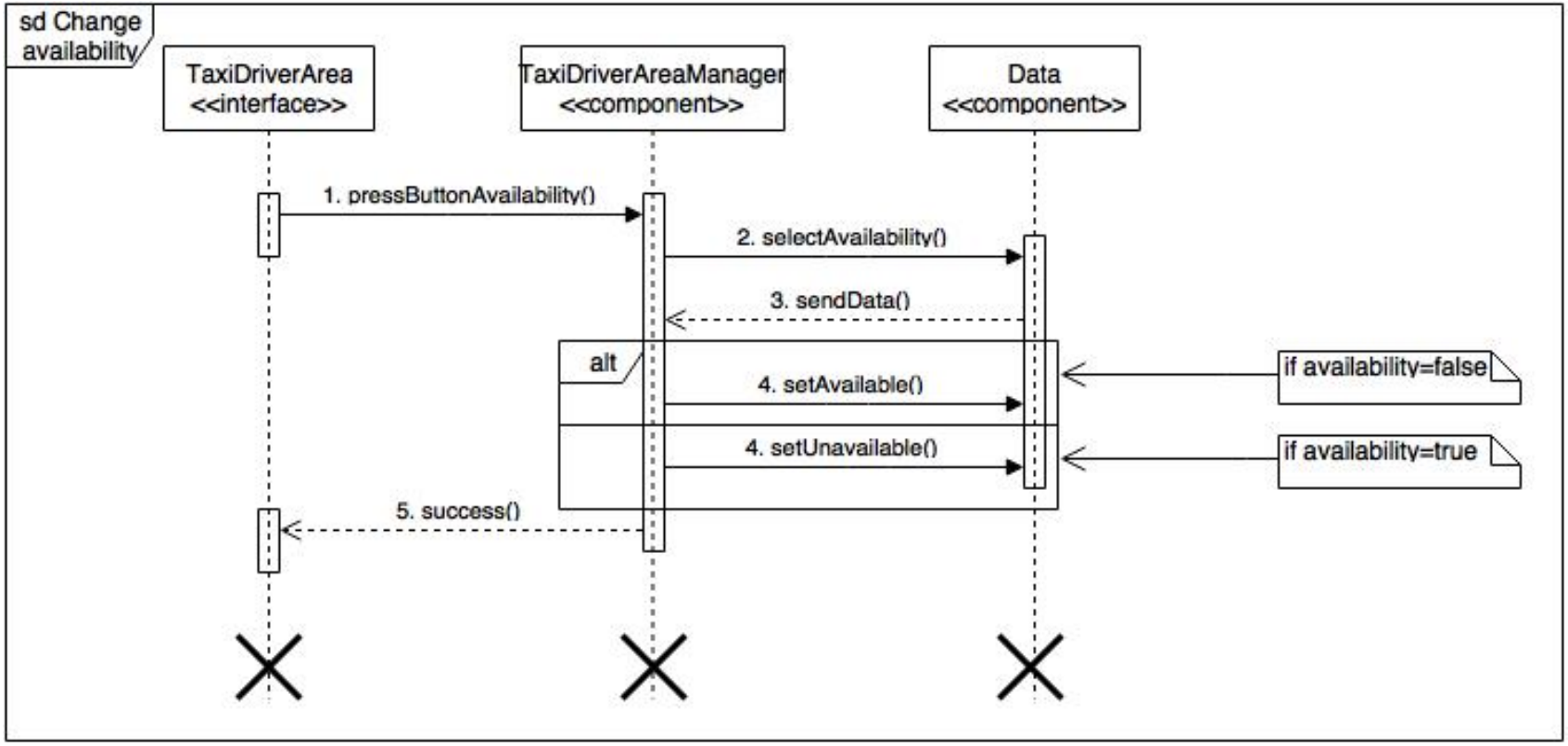
This diagram depicts how components presented in section 2.3 works in order to execute the operation of a request from a user and its forwarding to the taxi driver.
These kind of operations are shown also in the following sequence diagrams, here reported in order to show more in details how the components behave accomplishing these activities.




sd Request and Answer
Passenger
Taxi driver
Data <<component>>
RequestTaxi <<interface>>
RequestManager <<component>>
PaymentManager <<component>>
ChoosePaymentMethod <<interface>>
Notifications <<interface>>
1. pressRequestButton()
2. createPaymentMethodArea()
3. submitForm()
4. insertDataRequest()
5. selectTaxiOfArea()
6. sendQueueOfArea()
loop until it founds a taxi driver who takes the request.
loop
7. createRequestFormFirstTaxi()
Start timer (one minute)
9. sendTaxiAtEndOfQueue()
alt
8. submitTakeRequest()
8. submitCannotTakeRequest()
Time out
8. cancelRequest()
9. sendTaxiAtEndOfQueue()
9. updateDataRequest()
10. setTaxiDriverUnavailable()
11. taxiFound()


sd Signup
Access
<<interface>>
PassengerArea
<<interface>>
AccessManager
<<component>>
Data
<<component>>
1. submitRegistrationForm()
2. checkData()
alt
3. error()
3. dataOkWaitEmailCofnfirmation()
4. emailConfirmation()
5. insertData()
6. createPassengerArea()


sd Login
Access
<<interface>>
PassengerArea
<<interface>>
TaxiDriverArea
<<interface>>
AccessManager
<<component>>
Data
<<component>>
1. submitLoginForm()
2. selectData()
3. sendData()
4. checkData()
alt
5. error()
5. selectProfileInformation()
6. sendData()
alt
7. createPassengerArea()
7. createTaxiDriverArea()


sd Change availability
TaxiDriverArea <<interface>>
TaxiDriverAreaManager <<component>>
Data <<component>>
1. pressButtonAvailability()
2. selectAvailability()
3. sendData()
alt
4. setAvailable()
4. setUnavailable()
if availability=false
if availability=true
5. success()

2.6 Component interfaces

Here is presented a list of the interfaces defined in the component diagram in section 2.3 and their functionalities.

- **Access.** This interface allows a generic user to log into the system as a passenger or as a taxi driver and allows the visitor to sign up. It requires the visitor to insert his personal information, e-mail address and password if he wants to register, otherwise only e-mail address and password to log in.
  The inputs of the login as a passenger are:

  - Email (String)
  - Password (String)

  The inputs of the login as a taxi driver are:

  - Email (String)
  - Password (String)
  - Code (String)

  The inputs of the registration are:

  - Email (String)
  - Name (String)
  - Surname (String)
  - Password (String)
  - Phone number (String)

  If the data entered are correct, the output is the redirection to the user personal area, otherwise an error message is displayed.

- **TaxiDriverArea.** This interface allows the taxi driver to see and modify his profile information, to change his availability and to see his position in the queue. The taxi driver has to insert new personal information if he wants to modify his profile or he can press the availability button if he only needs to change his state.
  The inputs of the edit profile form are:

  - Email (String)
  - Name (String)
  - Surname (String)
  - Password (String)
  - Phone number (String)

  The input of the availability button is true or false whether he wants to declare itself available or not.
  The output of this interface is the profile of the taxi driver with the position of the queue, the availability button and the notifications.

- **PassengerArea.** This interface allows the passenger to see and modify his profile information, to make a request and to make a reservation. The passenger has to provide new personal information if he wants to edit his profile, otherwise he can press the designated button, in order to make the request or the reservation.
  The inputs of the edit profile form are:

  - Email (String)
  - Name (String)
  - Surname (String)
  - Password (String)
  - Phone number (String)

  The outputs are the profile of the passenger or the redirection to the screen of the request or of the reservation.

- **Notifications.** This interface consists of a panel that allows the taxi driver to see and to accept or decline the requests of the users. The input is *accept* if he wants to take the request, *decline* otherwise. If the taxi driver does not answer within one minute the input is automatically set to *timeOut*.
  The output is the history of the requests previously forwarded.

- **RequestTaxiVisitor.** This interface allows the visitor to make a request directly from the homepage. The input is his position in that moment, which is sent automatically from the user device when he presses the button.
  The output is a waiting screen.

- **RequestTaxiPassenger.** This interface allows the passenger to make a request. The input is his position in that moment, which is sent automatically from the user device when he presses the button.
  The output is the redirection to the payment method panel.

- **MakeReservation.** This interface allows the passenger to make a reservation by filling in a form.
  Its inputs are:

  - DepartureTime (Time)
  - Origin (String)
  - Destination (String)

  The output is the redirection to the payment method panel.

- **PaymentMethod.** This interface allows the passenger to choose the payment method he prefers. The input is *cash* if he wants to pay the taxi driver with cash, *card* otherwise. If he chooses to pay with the credit card he needs to provide his credit card data if he has not done it yet.
  The inputs of the credit card data form are:

  - CardType (String)
  - FirstName (String)
  - LastName (String)
  - CardNumber (int)
  - ExpirationDate (Date)
  - CVV (int)

  The output is the redirection to the waiting screen.

2.7 Selected architectural styles and patterns

Here are explained the architectural styles and patterns adopted.

- **Client/Server architecture.** The client/server architecture is the optimal solution for the myTaxiService system, as it is necessary to have a central system that listens, manages and forwards the requests of the different clients. There is a central server that contains the logic of the application and the clients are the users of the system, such as visitors, passengers and taxi drivers.

- **Three-tier architecture.** It has been adopted a three-tier architectural model, composed by thin client, application server and database.
  This architecture is the best choice for our system, even if it has some cons, such as the complexity of the structure and the difficulty of set up and maintenance, it still has several pros. For example, it guarantees increased performance and great flexibility, useful if there will be any future change concerning the architecture. Moreover it is granted a great security level, thanks to the decoupling of logic, data and presentation, which is essential as the system deals with several personal data.

- **Event-based system.** The myTaxiService application is based on the event firing.
  It is necessary, in fact, that the system is reactive and is able to do different quick operations according to the action of the clients. For the myTaxiService system events can be requests and reservations, as long as change of availability state or the answer to a request.
  The users and the taxi drivers are registered to different events and expect to receive notifications about what they need, whether it is the arrival of a taxi or the requests of the users.
  The events are asynchronous and based on a "send and forget" paradigm, where the system only cares for sending the notifications to the designated clients or doing the actions needed in response of the event triggered.

- **Service-Oriented Architecture (SOA).** A service-oriented architecture is necessary if the system wants to be more flexible and expandable. In fact, the myTaxiService application needs to provide programmatic interfaces in order to be open to future implementations of additional services. This can be guaranteed with this architectural choice, which is based on the loose coupling of the services, allowing to easily add more functionalities, without starting from scratch when a change is needed, and to simplify the maintenance of the system.

3 Algorithm Design

The most relevant algorithm of the myTaxiService application is the one implementing the queue management. It is important to optimize its implementation as long as it is the most complex and the one that distinguishes the system.
Another significant algorithm is the one that manages the forwarding of the requests of the users to the taxi drivers.

3.1 Initialization

The class which manages the areas distribution and the queues of the taxi drivers also deals with the initialization of the queues of the taxis. It starts assigning to every area as many taxi as the average number of the requests expected in that area, based on the statistics. Once all the areas have been filled with their optimal number of taxis, the remaining taxi drivers are sent to the areas that are most needy.
The "needyArea" function, in fact, returns the index of the most needy area at the moment, calculating the area that has the maximum difference between the maximum number of requests recorded and the average of requests.

Algorithm 1 Initialization

```
1:  procedure INITQUEUES(TaxiDriver[] taxiDrivers, Area[] areas)
2:      i ← 0
3:      j ← 0
4:      for i < areas.size() do
5:          k ← 0
6:          for k < areas[i].getAverage() do
7:              areas[i].addToQueue(taxiDrivers[j])
8:              j ← j + 1.
9:          end for
10:     end for
11:     if j < taxiDrivers.size() then
12:         for j < taxiDrivers.size() do
13:             areas[needyArea(areas)].addInQueue(taxiDrivers[j])
14:         end for
15:     end if
16: end procedure
```

3.2 Manage requests

After the initialization, the taxi drivers are able to take requests. When the taxi driver answers to the request or the timer is over, the system calls the function manageAnswer(). If they accept the request, the function sets their availability to false and stops monitoring their position. It is also possible that a taxi driver declines the request or is not able to give an answer to the server. In this case, the first taxi driver is moved to the last position of the queue and the system forwards the request to the second of the queue, who is now moved to the first position. This action is iterated in the queue of that area until a taxi accept the request. After accepting the call, the taxi driver is set unavailable and he is deleted from the queue, as it is likely that he will end up in another area.

Algorithm 2 Manage requests

```
1:  procedure FORWARDREQUEST(Position pos, Area area)
2:      timer ← new Timer(60)
3:      sendRequest(area.getFirstOfQueue(), pos)
4:      timer.startTimer()
5:  end procedure
6:  procedure MANAGEANSWER(String answer,Area area)
7:      if answer = "no" || answer = "timeOut" then
8:          area.moveFirstToEndOfQueue()
9:          forwardRequest(pos,area)
10:     end if
11:     if answer = "yes" then
12:         area.getFirstOfQueue().setUnavailable()
13:         area.deleteFirstFromQueue()
14:     end if
15:     timer.stopTimer()
16: end procedure
```

3.3 Manage queues

The areas of the city are represented by a graph with an array of adjacencies, which is a useful representation in order to decide how to distribute and move the taxi drivers within the areas.
In the first place the function search for the area where the taxi is in and sets the "startingArea" variable to the index of that area, then adds the taxi driver to the queue of the area if it has not reached the maximum number of taxi. If the area is completely full it is necessary to iterate on the areas, exploiting a breadth-first search algorithm.
The function searches for adjacent areas with the number of taxi lower than the average of the requests and, if it does not find any, it looks for adjacent areas with the number of taxi lower than the maximum allowed for that area.
If these two conditions are not supplied, the function assign this taxi driver to the first adjacent area visited and takes from that area the last taxi of the queue. This "rejectedTaxi" is moved to an adjacent needy area, repeating the iterations done before, searching needy areas and moving taxi drivers, until it is possible to reach an equilibrium state.
It is always feasible to exit the while cycle because the number of taxi driver can not be the maximum for all the areas, as it is not cost efficient. The system only ensures that the minimum number of taxi drivers per area is guaranteed.
The complexity of this algorithm is O(A) with A = number of areas.

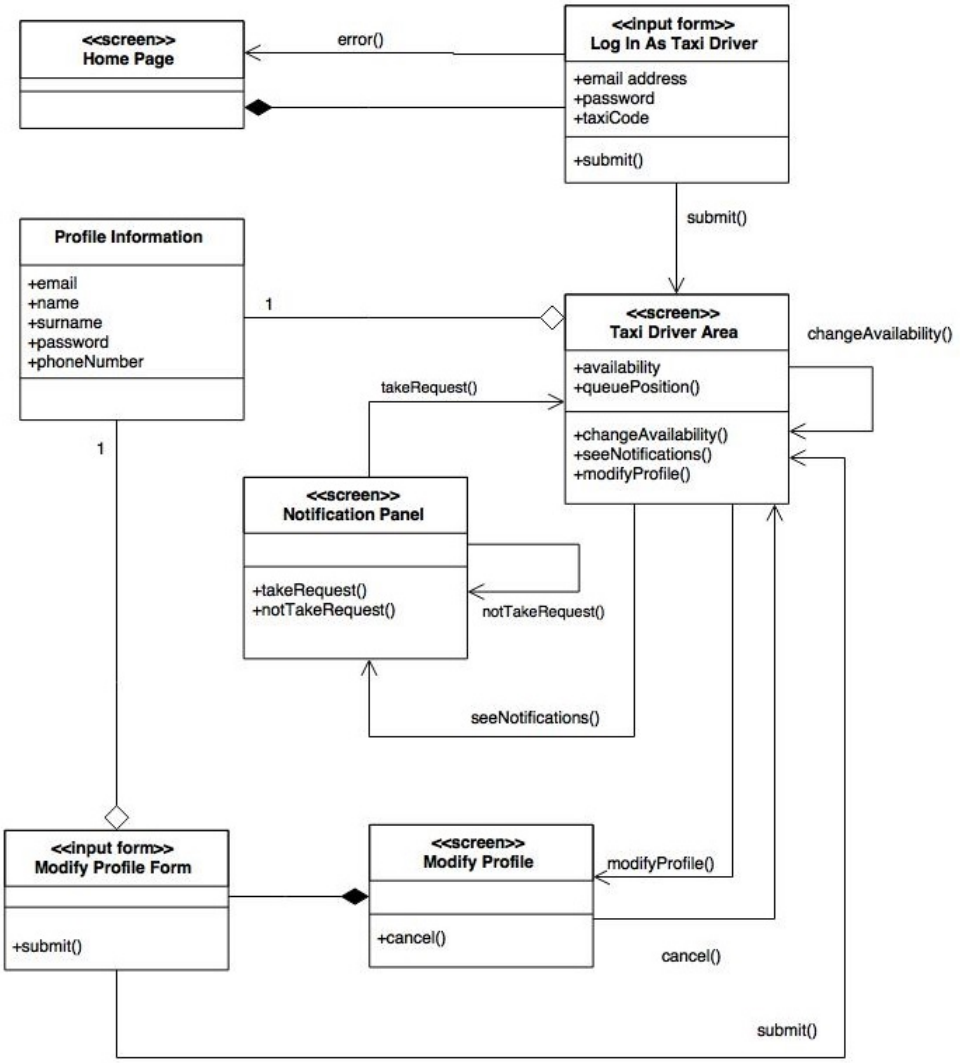
Algorithm 3 Manage queues

```
1: procedure MANAGEQUEUE(Area[] area, TaxiDriver taxiDriver)
2:     for i < area.size() do
3:         if taxiDriver.position is in area[i] then
4:             startingArea ← i
5:         end if
6:     end for
7:     if area[startingArea].getNumOfTaxi() < max[i] then
8:         area[startingArea].addToQueue(taxiDriver)
9:     else
10:        for i < area.size() do
11:            area[i].distance ← INFINITY
12:        end for
13:        area[startingArea].distance ← 0
14:        unvisitedQueue.enqueue(area[startingArea])
15:        rejectedTaxi ← taxiDriver
16:        while !unvisitedQueue.isEmpty() do
17:            tmpArea ← unvisitedQueue.dequeue()
18:            for i < tmpArea.adjacencies.size() do
19:                if tmpArea.adjacencies[i].getNumOfTaxi() < avg[i] then
20:                    tmpArea.adjacencies[i].addToQueue(rejectedTaxi)
21:                    return
22:                end if
23:            end for
24:            for i < tmpArea.adjacencies.size() do
25:                if tmpArea.adjacencies[i].getNumOfTaxi() < max[i] then
26:                    tmpArea.adjacencies[i].addToQueue(rejectedTaxi)
27:                    return
28:                end if
29:                if tmpArea.adjacencies[i].distance = INFINITY then
30:                    tmpArea.adjacencies[i].distance ← tmpArea.distance+1
31:                    unvisitedQueue.enqueue(tmpArea.adjacencies[i])
32:                end if
33:            end for
34:            firstUnvisited ← unvisitedQueue.getFirstOfQueue()
35:            rejectedTaxi ← firstUnvisited.addToFullQueue(rejectedTaxi)
36:        end while
37:    end if
38: end procedure
```

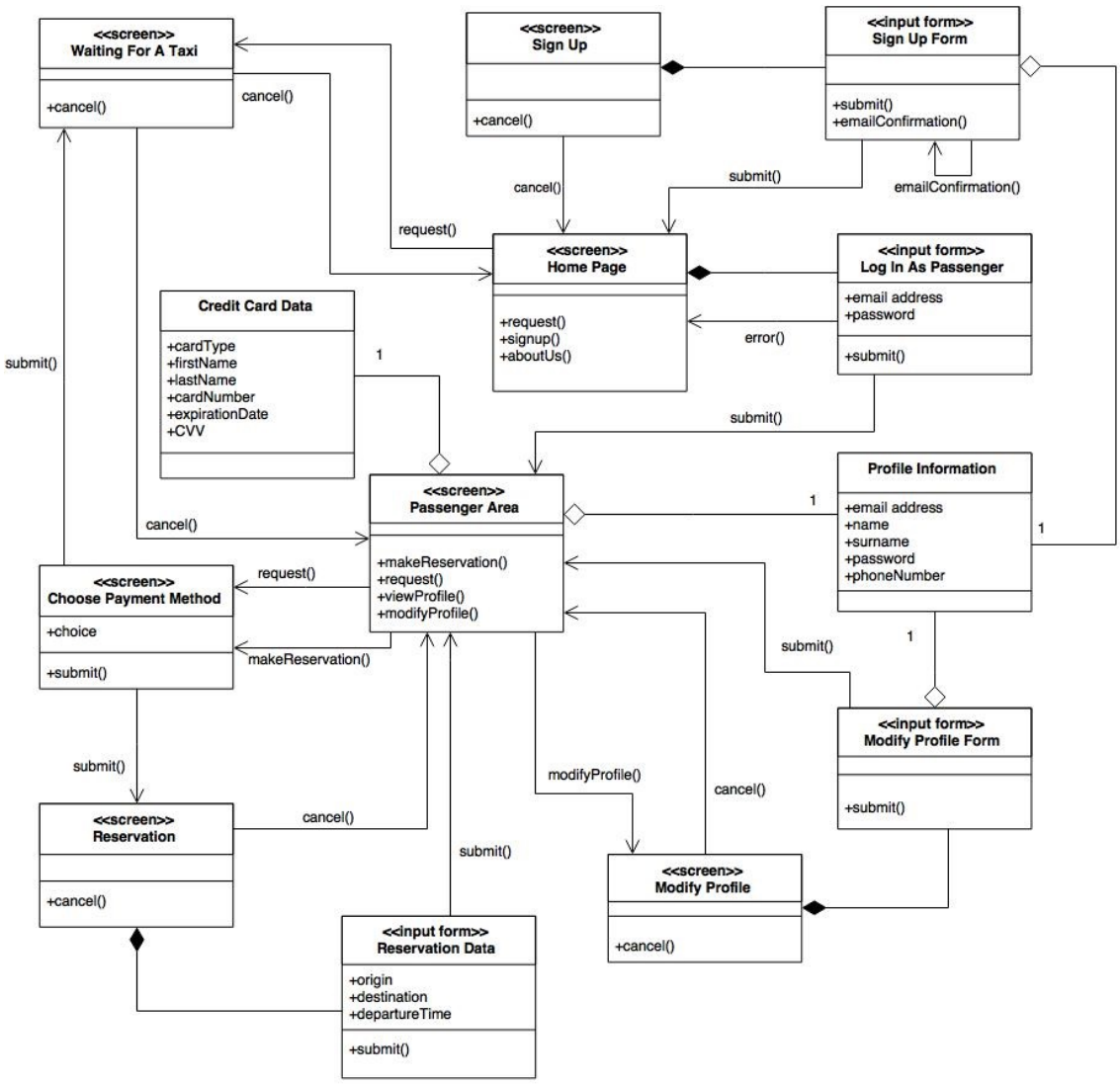
4 User Interface Design

The user interface design has been already specified in the section 3.1.1 of the RASD, where all the user interfaces are described with the support of mockups. In order to show in a simplified way the steps through the different screens, it has been used the UX diagram, one to illustrate the user interfaces and the other for the taxi driver ones.

4.1 Taxi driver UX diagram



4.2 User UX diagram



5 Requirements Traceability

- **Registration of the visitor.** This requirement is specified in section 2.3 in the "Access" interface and in the "AccessManager" component.
  It is also defined in section 4 in the "Sign Up" screen and in the "Sign Up Form" input form.

- **E-mail confirmation.** This requirement is specified in section 4 in the "Sign Up Form" input form.

- **Look up information about the system.** This requirement is specified in section 4 in the "Home Page" screen.

- **Log in as a passenger.** This requirement is specified in section 2.3 in the "Access" interface and in the "AccessManager" component.
  It is also defined in section 4 in the "Log In As Passenger" input form.

- **Log in as a taxi driver.** This requirement is specified in section 2.3 in the "AccessManager" component.
  It is also defined in section 4 in the "Log In As Taxi Driver" input form.

- **Log in with social networks.** This requirement is specified in section 2.3 in the "Access" interface and in the "AccessManager" component.
  It is also defined in section 4 in the "Log In As Passenger" input form.

- **Retrieve password.** This requirement is specified in section 2.3 in the "AccessManager" component.

- **Request a taxi.** This requirement is specified in section 2.3 in the "RequestTaxi" interface and in the "RequestManager" component.
  It is also defined in section 4 in the "Passenger Area" screen.

- **Reserve a taxi.** This requirement is specified in section 2.3 in the "MakeReservation" interface and in the "RequestManager" component.
  It is also defined in section 4 in the "Reservation" screen and "Reservation Data" input form.

- **Choose payment method.** This requirement is specified in section 2.3 in the "ChoosePaymentMethod" interface and in the "PaymentManager" component.
  It is also defined in section 4 in the "Choose Payment Method" screen.

- **View profile.** This requirement is specified in section 2.3 in the "PassengerArea" and "TaxiDriverArea" interfaces.
  It is also defined in section 4 in the "Passenger Area" screen.

- **Edit profile.** This requirement is specified in section 2.3 in the "PassengerArea" and "TaxiDriverArea" interfaces and in the "PassengerAreaManager" and "TaxiDriverAreaManager" components.
  It is also defined in section 4 in the "Modify Profile" screen and "Modify Profile Form" input form.

- **See waiting time.** This requirement is specified in section 4 in the "Waiting For A Taxi" screen.

- **Change availability.** This requirement is specified in section 2.3 in the "TaxiDriverArea" interface and in the "TaxiDriverAreaManager" component.
  It is also defined in section 4 in the "Taxi Driver Area" screen.
- **Notification of the requests.** This requirement is specified in section 2.3 in the "TaxiDriverArea" interface and in the "TaxiDriverAreaManager" component.
  It is also defined in section 4 in the "Notification Panel" screen.
- **See position in the queue.** This requirement is specified in section 2.3 in the "TaxiDriverArea" interface and in the "TaxiDriverAreaManager" component.
  It is also defined in section 4 in the "Taxi Driver Area" screen.

6 References

- ISO/IEC/IEEE 42010:2011, Systems and software engineering - Architecture description
- UML Component Diagrams
  (`http://www.uml-diagrams.org/component-diagrams.html`)
- Introduction to the Diagrams of UML 2.X
  (`http://www.agilemodeling.com/essays/umlDiagrams.htm`)

7 Appendix

7.1 Software and tools used

- TeXstudio (`http://www.texstudio.org/`): to redact and to format this document.
- draw.io (`https://www.draw.io/`): to create all the diagrams.

7.2 Hours of work

Time spent redacting this document:

- Cattaneo Michela Gaia: ~35 hours of work.
- Barlocco Mattia: ~35 hours of work.