A.Y. 2015/2016

POLITECNICO DI MILANO

COMPUTER SCIENCE AND ENGINEERING

SOFTWARE ENGINEERING 2

MyTaxiService

Design Document

Authors

SARA PISANI – 854223
LEONARDO TURCHI – 853738

04/12/2015

MyTaxiService-DD.pdf · V 1.0

Table of contents

1. Introduction

1.1 Purpose

The purpose of this document is to illustrate the design of the application to transport service " MyTaxiService"

1.2 Scope

This paper aims to describe in detail the architectural and functional characteristics of the system, with particular attention to the description of the choices related to the structure of the database and the interaction between users and MyTaxiService.

This document complies with the specifications described in the RASD and its draft is totally based on it, although the RASD has undergone some changes, as will be seen later in this document.

1.3 Definition, acronyms and abbreviations

- **FIFO**: First In First Out.
  Elements may be inserted at any time, but only the element which has been in the queue for more time may be removed.
  Elements are inserted at the rear (*enqueued*) and removed from the front (*dequeued*).

- **LIFO**: Last In First Out.
  Elements may be inserted at any time, but only the element which has been in the queue for less time may be removed.
  Elements are inserted at the rear (*enqueued*) and removed from the rear (*dequeued*)

- **MVC**: Model View Controller.
  Is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

- **UML**: Unified Modeling Language.
  Is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system.

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

1.4 Sources and reference documents

- IEEE Standard for Information Technologies – Systems Design – Software Design Descriptions
- IEEE Standard Systems and Software Engineering – Architecture Description
- Requirements Analysis and Specification Document (RASD), already available in this folder

1.5 Document structure

The paper is organized as follows:

- **Chapter 1**: This section explains the purpose, the scope and give some general details about the Project.
- **Chapter 2**: This section provides description of the architectural design, paying attention to the pattern used to relate each component involved.
- **Chapter 3**: This section contains all the necessary software information about the most algorithmic part of the project.
- **Chapter 4**: This section outlines an overview on how the user interfaces on the system will look like.
- **Chapter 5**: This section explains the connection between the requirements already defined in the RASD and the design elements introduced in this document.
- **Chapter 6**: This section describes references to other documents.
- **Chapter 7**: This section let the reader know the amount of work to write this document.

2. Architectural design

2.1 Overview

This chapter will be focused on how users can navigate *MyTaxyService*, explaining iterations between different pages and the system behavior after a user's action.

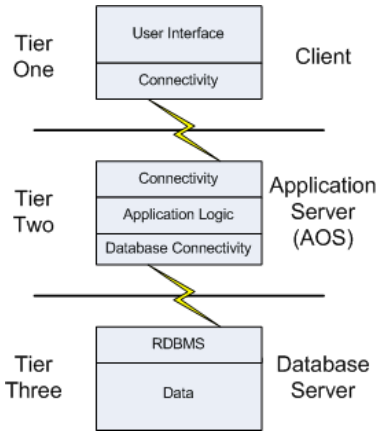
The first page that everyone can see is the **Registration** or **Login** page, because we hypothesized that a not registered user can't use MyTaxiService or a part of this service, so a Registration and a Login are required.

After Login, the system recognizes two kind of users: **Passenger** and **Drivers**, and directs each one to the appropriate home page.

MyTaxyService's Menu (the **passengers'** home page), allows users to reach two different form (to book a taxi immediately or at a certain time) by pressing a button.
In case of multiple booking, this menu allows to consult a page with the list of rides and with the button "*Info*", associated to each reservation, a popup shows all reservation data.

MyTaxyService's Driver Area (the **drivers'** home page), allows users to see the imminent ride (to the left page) and a list of booking (to the right page).
A driver can see details of each ride pressing a button connected to an information popup and, if a passenger requires his taxi, a popup appears on his screen: he can decline or accept the ride pressing a button on it.

2.2 High level components and their interaction



2.2.1 Infrastructure

This is an overview of the infrastructure that makes the website and app of MyTaxiService working.
For the backend is used the three tier architecture, that consist of:

1. **TIER ONE**
   - **Client Tier**
     The part that runs on the client devices through a 'browser' or through an application (for mobile devices).
     In the case of the app for drivers, the client monitors for the GPS coordinates and send this information to the server.
     In the case of the user, this tier makes possible to insert data into web-forms and send these to the tier two.
     This is the topmost level of the application.

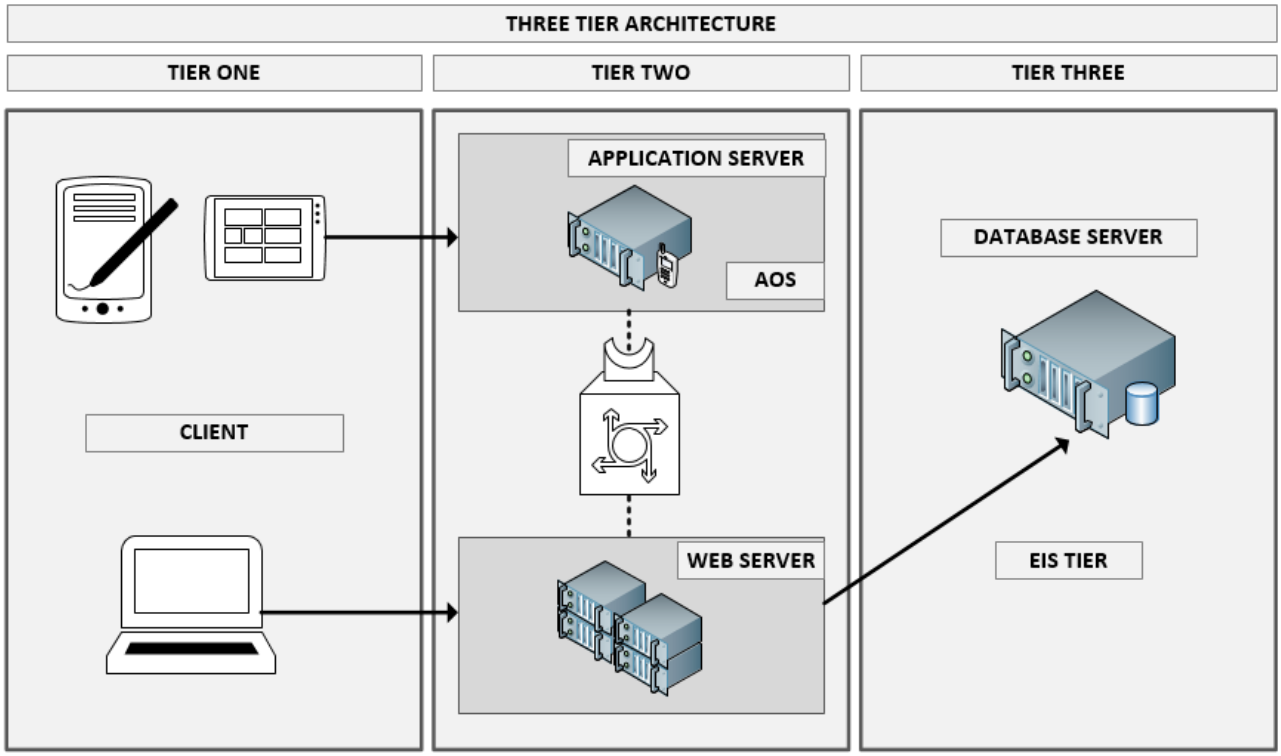
2. **TIER TWO**
   Runs on the JEE server and consist of two subsets:
   - **Web Tier**
     It creates the pages (interfaces) for client tier (using a direct communication with the Business Tier).
     Is always in listening for clients' requests. In case of need a post-processing of these, it forward them to the Business Tier.
   - **Business Tier (AOS)**
     It contains all the logical part of the application.
     It can also communicate with the Database, for the retrieval or the insertion of data.

3. **TIER THREE**
   - **EIS Tier**
     The data tier includes the data persistence mechanisms (database servers, file shares, etc.): where all data is stored.
     It can be fetched only by the Tier Two, but administrators (call center) could access to this tier too.

NOTE

- **AOS**
  Application Object Server
  The AOS is used to execute application objects, such as queries and classes. Application objects in the user interface, such as forms and reports, run on the client computer.
  More sessions are needed as more client programs connect to the AOS tier. The workload on the AOS tier can be shared among multiple instances of the AOS. These AOS instances can be distributed among one or more computers. The system controls which AOS each new client session connects to. The system balances the workload among AOS instances to improve performance.

- **EIS**
  Enterprise Information System Tier
  The enterprise information system (EIS) tier, in J2EE architecture, handles enterprise information system software, which provides an enterprise's critical business information infrastructure.



2.2.2 Components

2.2.2.1 Map API Controller

This component manages an eventual interaction between Google APIs and MyTaxiService: information exchange and callings of methods.

2.2.2.2 Queue Algorithm

This component use the algorithms A1 and A2 to manage the queue in each zone.

2.2.2.3 Application Controller

This component define the application behavior: it's the logical part of the system.
It manages the user requests, communicating with other components.

2.2.2.4 Mobile Application and View Controller

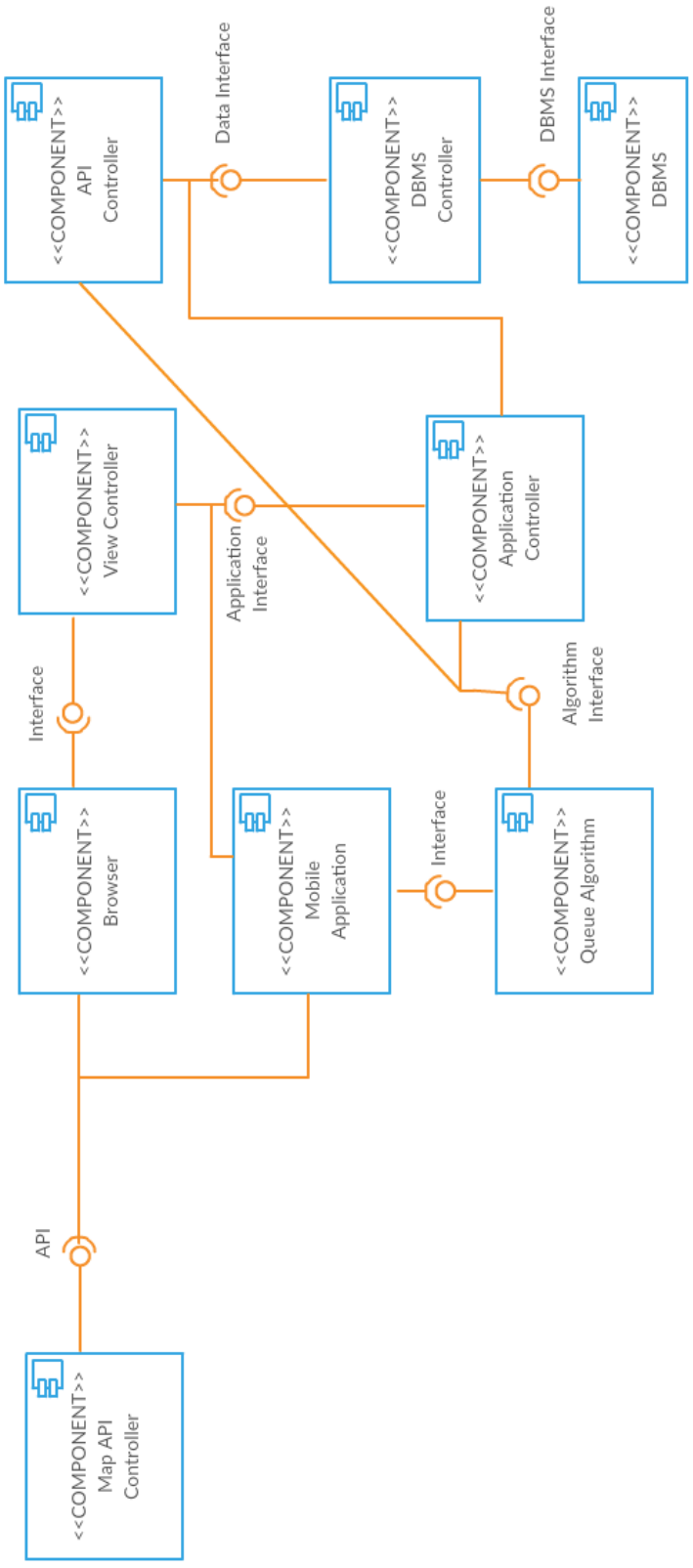
This component takes care to provide the correct interface: via web or via mobile.

2.2.2.5 DBMS Controller

This component gives to the application the APIs necessary to communicate to the DBMS.

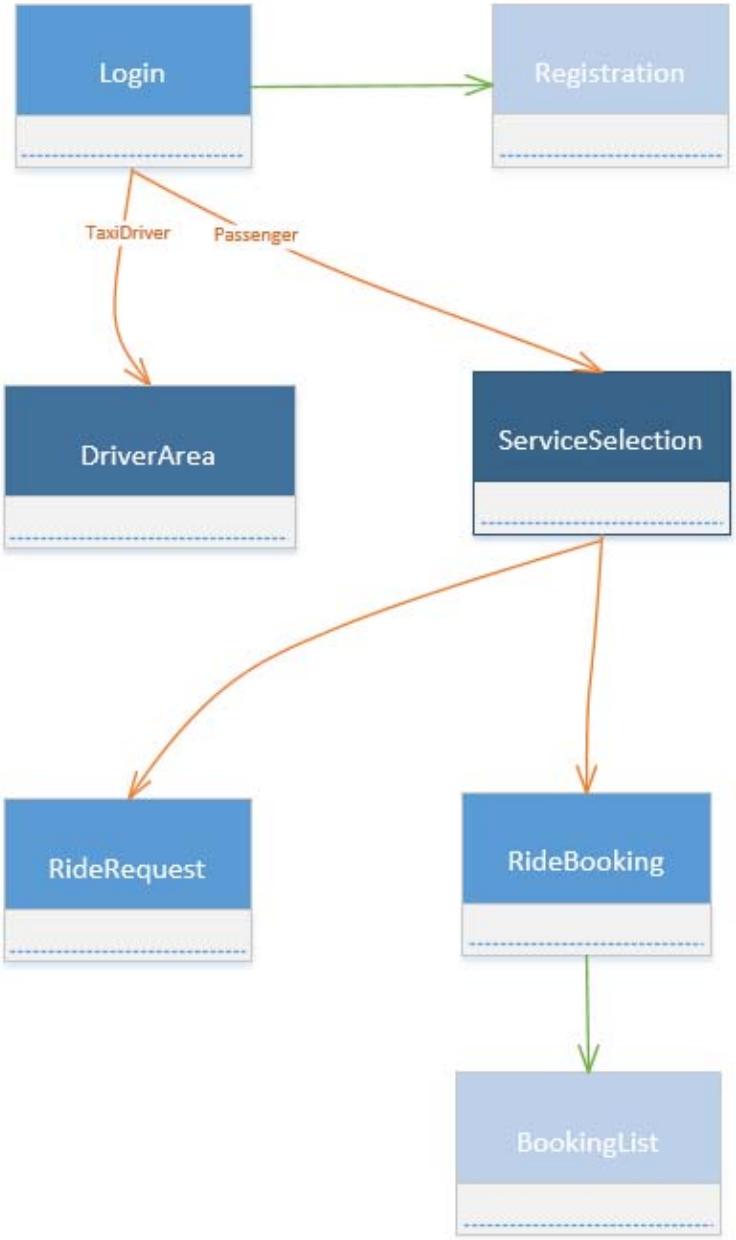
2.2.2.6 DBMS

This component manages the query, has the access to the database and handles the errors and the conflicts.


<<COMPONENT>>
API
Controller
Data Interface
<<COMPONENT>>
DBMS
Controller
DBMS Interface
<<COMPONENT>>
DBMS
<<COMPONENT>>
View Controller
Application
Interface
<<COMPONENT>>
Application
Controller
Interface
Algorithm
Interface
<<COMPONENT>>
Browser
<<COMPONENT>>
Mobile
Application
Interface
<<COMPONENT>>
Queue Algorithm
API
<<COMPONENT>>
Map API
Controller

2.2.3 Navigation

Regarding the navigation flow for the client side, follows a high level diagram that shows the structure of the main pages and choices that could be done by user.



2.3 Component view

In this paragraph are described in details the components previously defined.

2.3.1 Controllers

2.3.1.1 Application Controller

The application controller, manage the communication between clients and internal components.
It provides the ability to exchange data from an internal component to a client (and so from a client to the system).
This interface is unique and does not depend by other components that are going to use it. It dispatches the message coming from the client to the right internal component.
It manages the ride requests and ride bookings, by call to DBMS and Queue Algorithm.

2.3.1.2 Mobile Application Controller

This is the main controller for the mobile application.
It has to show to user the correct view and correct data, in order to make a best fruition of the service in mobile.
Via this controller can be made ride request and all the other ride managements in a mobile environment.
It has also the charge of retrieve the GPS coordinates, and send them to server.

2.3.1.3 Algorithm Controller

The main algorithm controller has to provide a communication between the algorithm pool, and the processes that need one of them.
The main algorithm of the system, is the queue's one, because it is called by the application manager every time there is the necessity to manage a request or a taxi.
The execution flow of an algorithm is provided in the section 3 of this document.

2.3.1.4 API Controller

The API controller replicate many functionalities of the Application Controller, written over.
Even there is a duplication of some functionalities, we decided to maintain this controller because we wanted a clear separation between the API and Application.
The API controller expose the DBMS and other components of the

application to external requests. In case of a ride request (for example) it allow the queue algorithm to communicate with the DBMS. In case of a mobile use (a mobile device via the app), it makes possible the data exchange from/to the DBMS to/from the application controller.

2.3.2 UX Components

2.3.2.1 Registration form

This part of the UX Diagram describes the first page that the user can see.
If a user is not already registered, he has to compile a form with the fields

- *"e-Mail"*
- *"Password"*
- *"Retype Password"*
- *"PayPal Account"*
- *"Date of Birth"*
- *"Name", Surname"*
- *"CF/ID"*
- *"Address"*

and if all data are corrected its submission allow user to create a new account, otherwise a popup with an error message will appear on the screen.

2.3.2.2 Login form

This part of the UX Diagram describes the first page that the user can see.
If a user has already an account, he can navigate to the other pages after the compilation of the login form.

In this area only two field are required:

- *"Username"*
- *"Password"*

2.3.2.3 Driver Area

This area is reachable only if the system recognizes the user as a taxi driver.
This is a sort of Home page, divided into two different pages placed side by side and selectable with a scroll gesture on the screen.

In the left page a taxi driver can examines the current ride requests in a list, while in the right page he can examine his list of booked ride requests.

2.3.2.4 Service Selection (Menu)

This area is reachable only if the system recognizes the user as a passenger.
This is a sort of Home page, where the user can choose a reservation service through a button.

2.3.2.5 Ride request and Ride booking form

These forms are reachable clicking a button between

- *"RideRequest" (immediate booking)*
- *"RideBooking" (delayed booking)*

Both of the form have the fields

- *"Start Address"*
- *"Destination Address"*
- *"Time of Booking"*
- *"Date of Booking"*

but in the case of a ride request the system automatically sets the last two field with the current hour and date.

2.3.2.6 Booked rides List

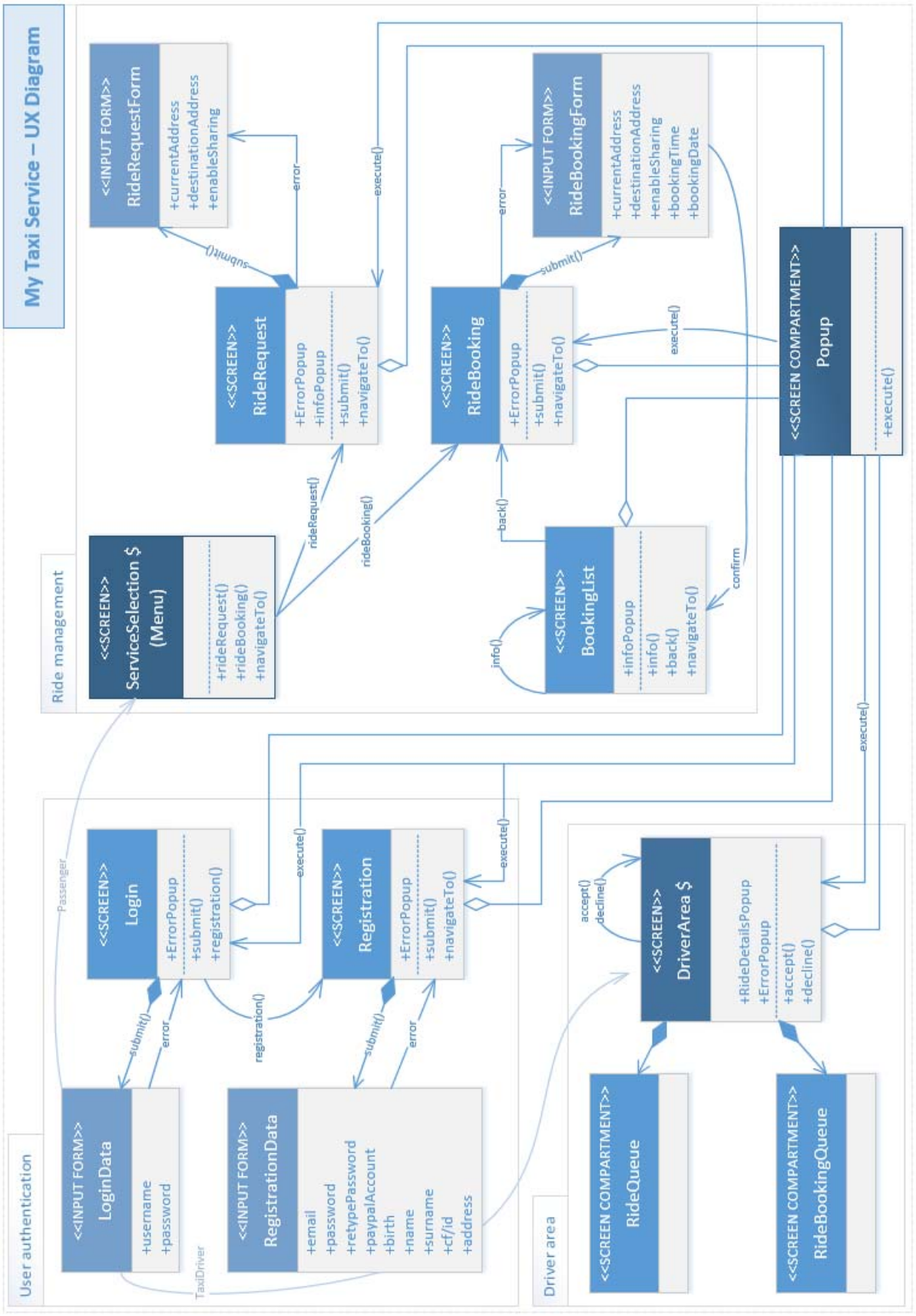
This page has a similar functionality both passenger side and driver side: allows drivers to see the list of all the rides' details that they have confirmed and allows passengers to see the list of all the rides' details that they have booked simply pressing the button "Info".

2.3.2.7 Popups

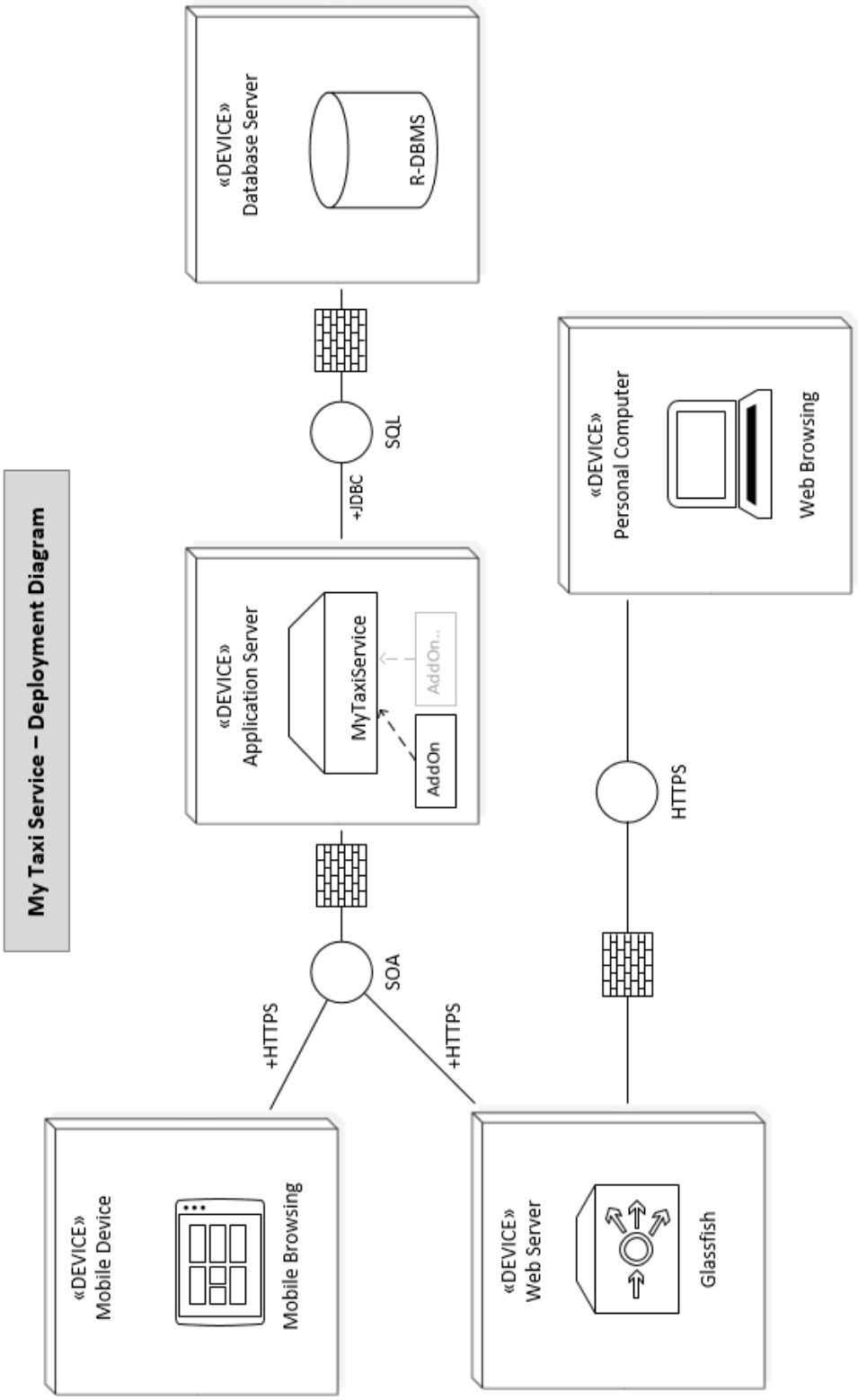
these elements are used in different way, so there are two types depending on use:

- Choice popup: with a text including a question that need that the user makes a choice between two buttons (✓ and ✗).
  Message Popup.

- Info popup: with a text including information about the action that the user is playing and the only closing option (✗) or the confirm option (✓).


My Taxi Service – UX Diagram
Ride management
User authentication
Driver area
Passenger
TaxiDriver
<<INPUT FORM>>
RideRequestForm
+currentAddress
+destinationAddress
+enableSharing
<<INPUT FORM>>
RideBookingForm
+currentAddress
+destinationAddress
+enableSharing
+bookingTime
+bookingDate
<<SCREEN>>
RideRequest
+ErrorPopup
+infoPopup
+submit()
+navigateTo()
<<SCREEN>>
RideBooking
+ErrorPopup
+submit()
+navigateTo()
<<SCREEN COMPARTMENT>>
Popup
+execute()
<<SCREEN>>
ServiceSelection $
(Menu)
+rideRequest()
+rideBooking()
+navigateTo()
<<SCREEN>>
Bookinglist
+infoPopup
+info()
+back()
+navigateTo()
<<SCREEN>>
Login
+ErrorPopup
+submit()
+registration()
<<SCREEN>>
Registration
+ErrorPopup
+submit()
+navigateTo()
<<SCREEN>>
DriverArea $
+RideDetailsPopup
+ErrorPopup
+accept()
+decline()
<<INPUT FORM>>
LoginData
+username
+password
<<INPUT FORM>>
RegistrationData
+email
+password
+retypePassword
+paypalAccount
+birth
+name
+surname
+cf/id
+address
<<SCREEN COMPARTMENT>>
RideQueue
<<SCREEN COMPARTMENT>>
RideBookingQueue
submit()
error
execute()
rideRequest()
rideBooking()
back()
confirm
info()
registration()
accept()
decline()

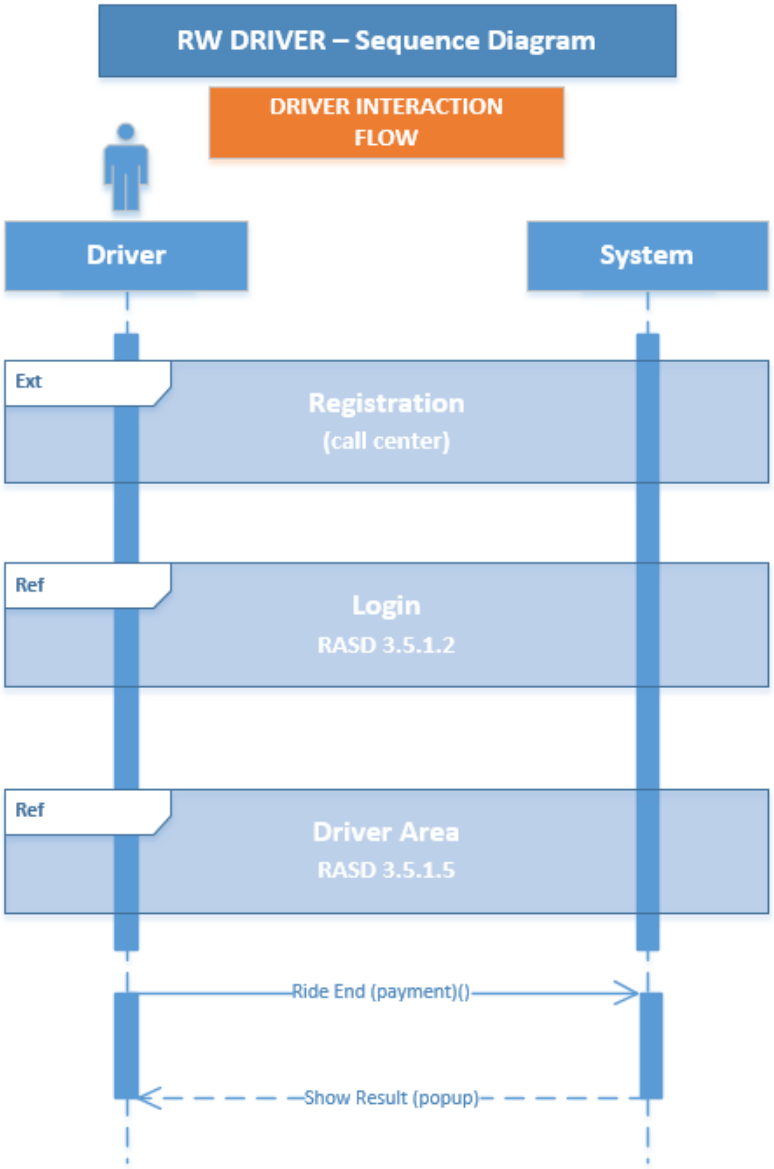
2.4 Deployment view

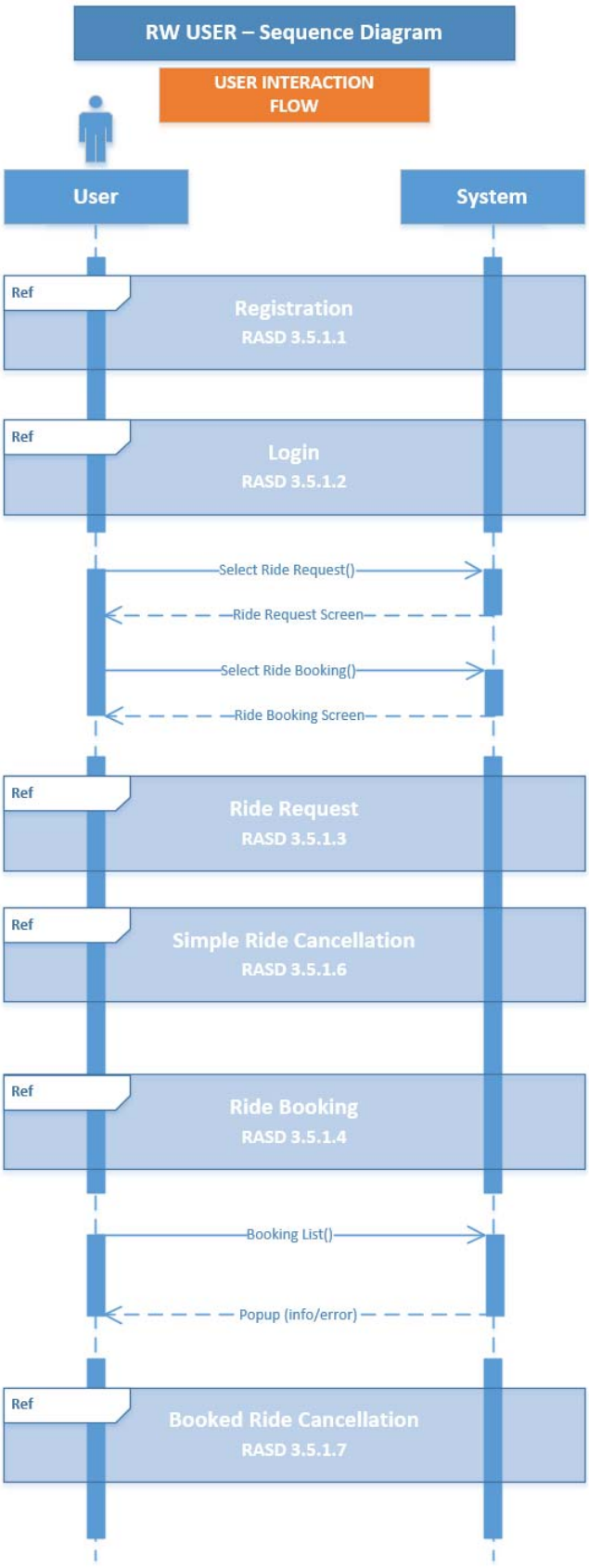


2.5 Runtime view

In this section, sequence diagrams are used to describe the way components interact to accomplish specific tasks.

In the RASD document already exists a section that explains how the system allows some operations, but it is general and tasks are divided and analysed one by one: here we will tie them, specifying some details about the RASD documentation that treat this topic.




RW USER – Sequence Diagram
USER INTERACTION FLOW
User
System
Ref
Registration
RASD 3.5.1.1
Ref
Login
RASD 3.5.1.2
Select Ride Request()
Ride Request Screen
Select Ride Booking()
Ride Booking Screen
Ref
Ride Request
RASD 3.5.1.3
Ref
Simple Ride Cancellation
RASD 3.5.1.6
Ref
Ride Booking
RASD 3.5.1.4
Booking List()
Popup (info/error)
Ref
Booked Ride Cancellation
RASD 3.5.1.7

2.6 Component interfaces

2.6.1 Map API Interface

This interface takes the GPS coordinates of the driver by the Mobile Application and sends the position of the taxi to the queue algorithm, that will manage the taxi's position in the queue.

2.6.2 Queue Algorithm Interface

This interface allows the call to the method that assigns a taxi to a passenger's request. The management of the queue is guaranteed by the algorithms [A1], [A2][C1] and [A2][C2] explained in the chapter 3.

2.6.3 Application Interface

This interface links the component Application Manager and the rest of application: It allows the communication between View Controller and the Mobile Application from a side, and the logic of the application, the actions and requests done by the client by the other side. It allows the Queue Algorithm to communicate an assignation to the Taxi Driver chosen by the assignation algorithm.

2.6.4 DBMS Interface

The DBMS Interface is thinked as a generic one, because it can be of many types (MySQL, MsSQL etc.). The DBMS Controller is not related to the specific DBMS, because it's only an abstraction of this interface and it should work with everyone. This interface is fondamental to do operations on the real database.

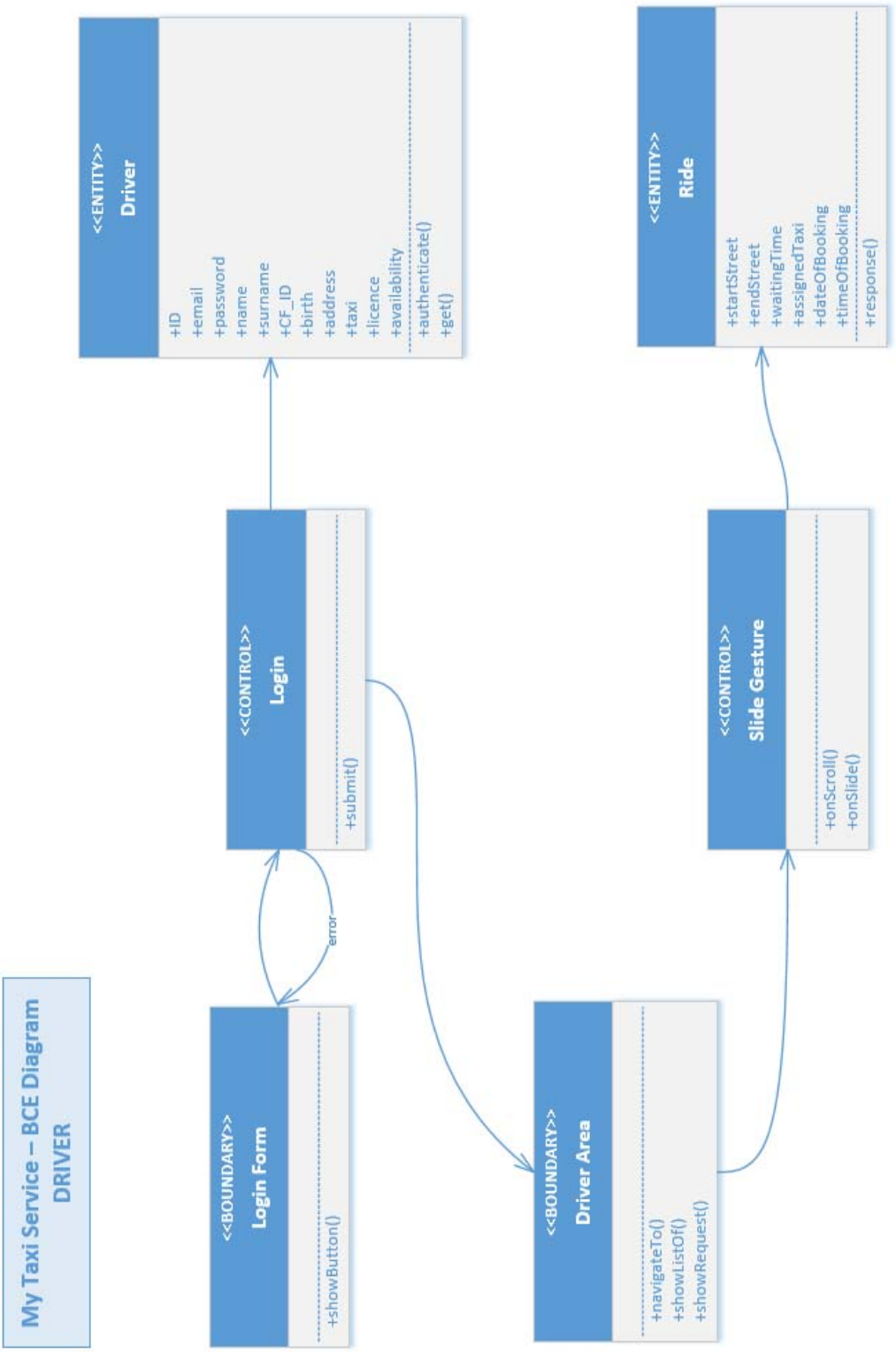
2.7 Selected architectural styles and patterns

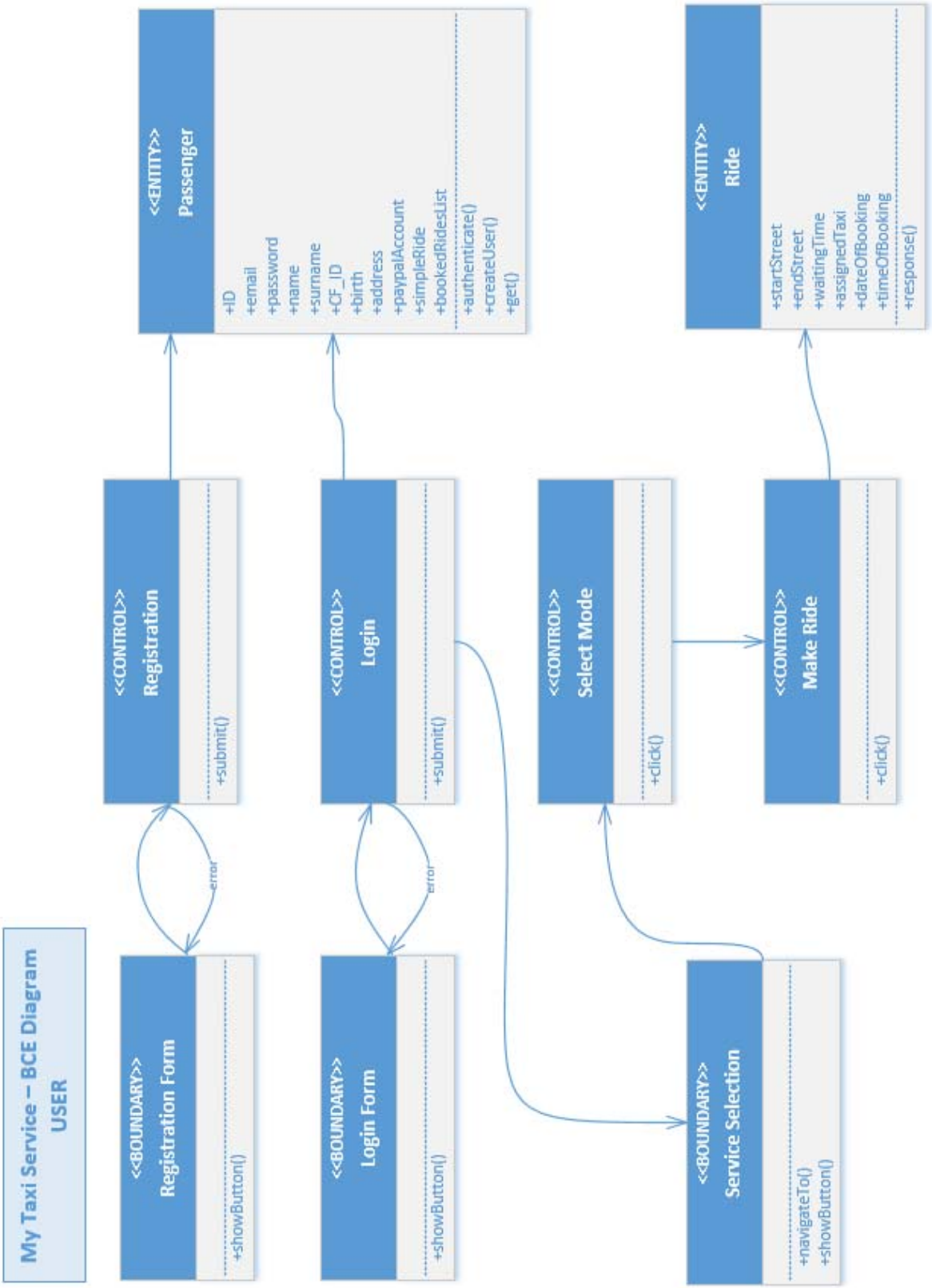
MyTaxiService application is structured by the pattern **BCE** (*Boundary-Control-Entity*).
This diagram is used to give a representation of "*model-view-controller*" pattern that was used to design the application.

- The stereotype Boundary is the user interface that has been described in detail through the **UX** diagram.

- The Control comprehends the classes that manage the logic of the system and that mediate between the user interface and the data.

- The Entity comprehends the classes of data in the database.


My Taxi Service – BCE Diagram
DRIVER
<<ENTITY>>
Driver
+ID
+email
+password
+name
+surname
+CF_ID
+birth
+address
+taxi
+licence
+availability
+authenticate()
+get()
<<ENTITY>>
Ride
+startStreet
+endStreet
+waitingTime
+assignedTaxi
+dateOfBooking
+timeOfBooking
+response()
<<CONTROL>>
Login
+submit()
<<CONTROL>>
Slide Gesture
+onScroll()
+onSlide()
error
<<BOUNDARY>>
Login Form
+showButton()
<<BOUNDARY>>
Driver Area
+navigateTo()
+showListOf()
+showRequest()


My Taxi Service – BCE Diagram
USER
<<BOUNDARY>>
Registration Form
+showButton()
<<BOUNDARY>>
Login Form
+showButton()
<<BOUNDARY>>
Service Selection
+navigateTo()
+showButton()
<<CONTROL>>
Registration
+submit()
<<CONTROL>>
Login
+submit()
<<CONTROL>>
Select Mode
+click()
<<CONTROL>>
Make Ride
+click()
error
error
<<ENTITY>>
Passenger
+ID
+email
+password
+name
+surname
+CF_ID
+birth
+address
+paypalAccount
+simpleRide
+bookedRidesList
+authenticate()
+createUser()
+get()
<<ENTITY>>
Ride
+startStreet
+endStreet
+waitingTime
+assignedTaxi
+dateOfBooking
+timeOfBooking
+response()

2.8 Other design decisions

- **Singleton Pattern**
  This is an useful pattern to create an unique instance of the class able access to the database: in this way conflicts as the reservation of the same taxi made by different passengers at the same time are avoided.

- **Strategy Pattern**
  This is an useful pattern to manage the algorithm [A2], that will be described in the next chapter: it must be used, at runtime, in different ways, depending on the need for which is moved a taxi.

- **Factory Pattern**
  This is an useful pattern to manage the "after login" behavior of the system: it recognize a login made by a driver or a passenger , so it's important, at runtime, to create different types of session that conduct at the Driver Area in the first case and at the Service Select in the second case.

3. Algorithm design

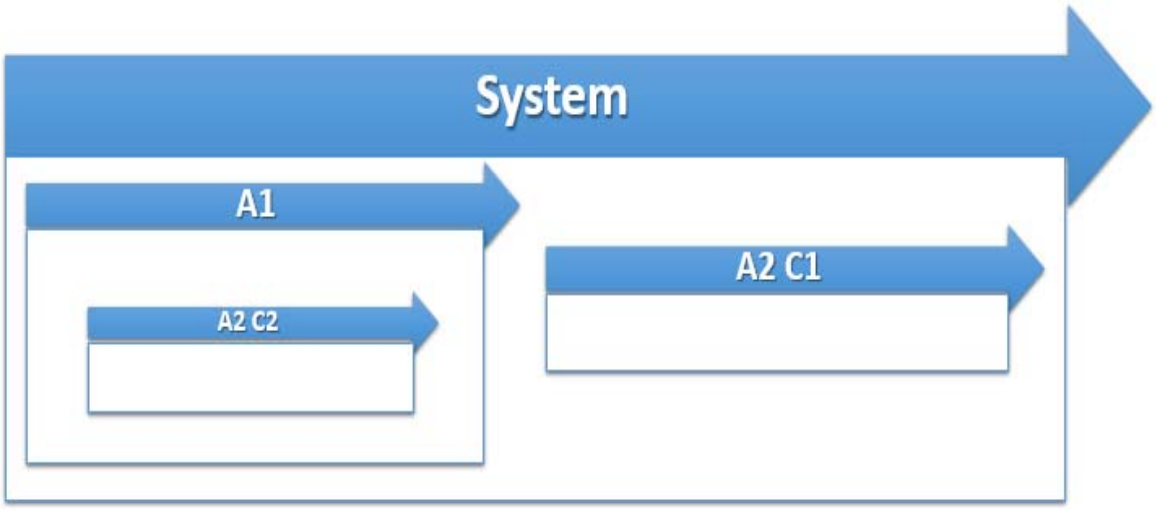
This section will be focused on the definition of the most relevant algorithmic part of MyTaxiService project: the management of the taxi, both among different areas and in a single zone.

The division of the algorithm is based on two macro-areas:

- [**A1**] Taxi balancing around zones
- [**A2**] Management of queues in depth

The second algorithm is split in two different uses [C1] and [C2].

NOTE
The following algorithms are strictly related, in fact on the inside of [A1] is called the algorithm [A2] [C2], as you can see in the following recap:



3.1 [A1] The positioning strategy of taxies in the city (balancing)

1. The basic algorithm, obtained through the application of the linear regression, assumes the following relationship:

$$N = \frac{Population}{5000 \cdot S}(\alpha \cdot \beta \cdot \gamma) + \Delta$$

where

- ***N*** *: theoretical requirement bid for the performance of taxi service in the City*
- ***Population*** *: Resident population in the town (n° inhabitants)*
- $\alpha \cdot \beta \cdot \gamma$ *: corrective factors for public transport/hospitals/public buildings/rail stations*
- Δ ***coefficient*** *: additional factor for summer season or touristic events*
- ***NOTE****: the reduction coefficient 5000 is valid only for applications in cities with an average number of population between 700.000 and 1.000.000; in case of different use, scale this coefficient with a linear value (****S****).*

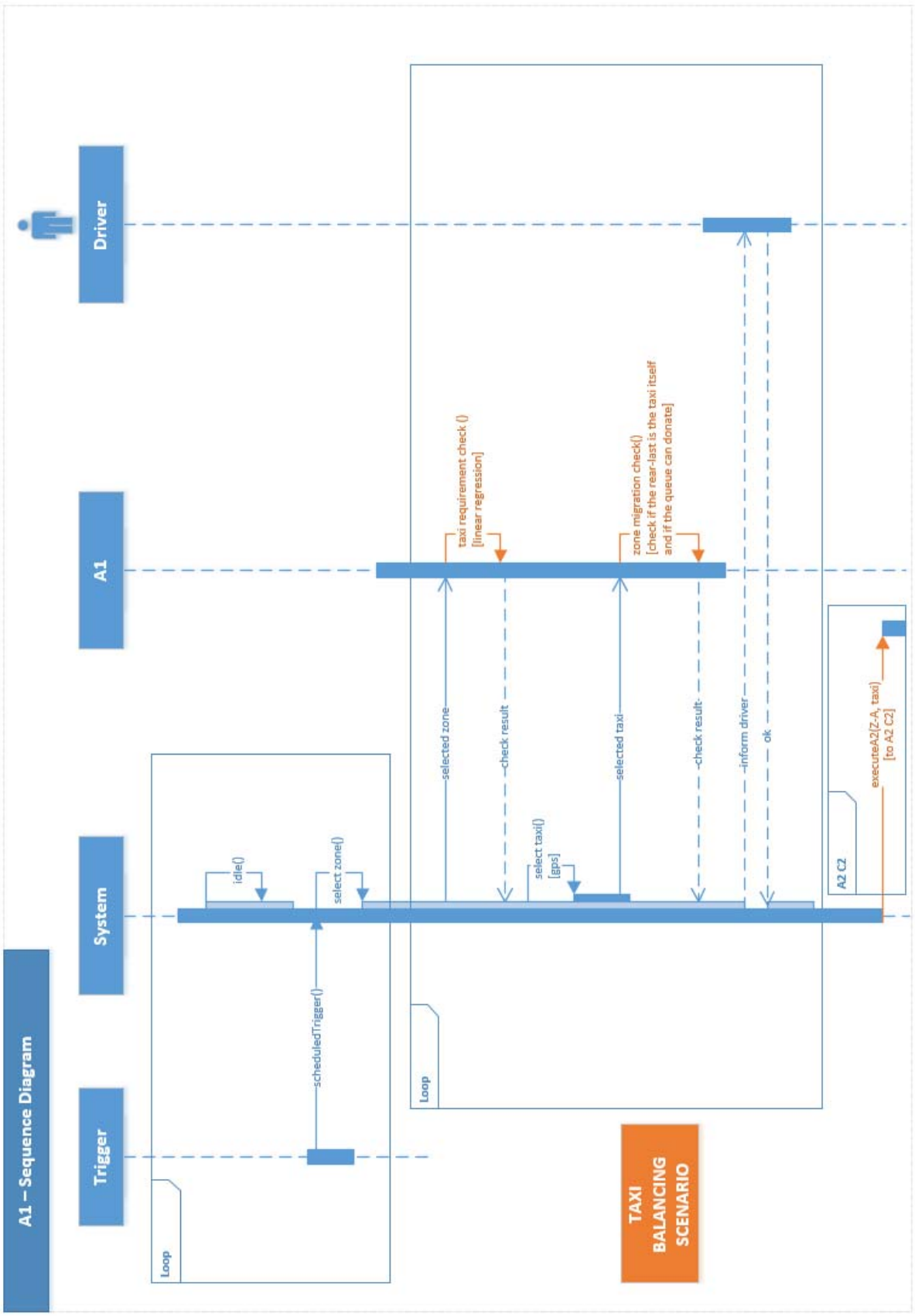
2. Once calculated the need for taxies for each zone, the system tracks the closest taxi using its GPS coordinates.

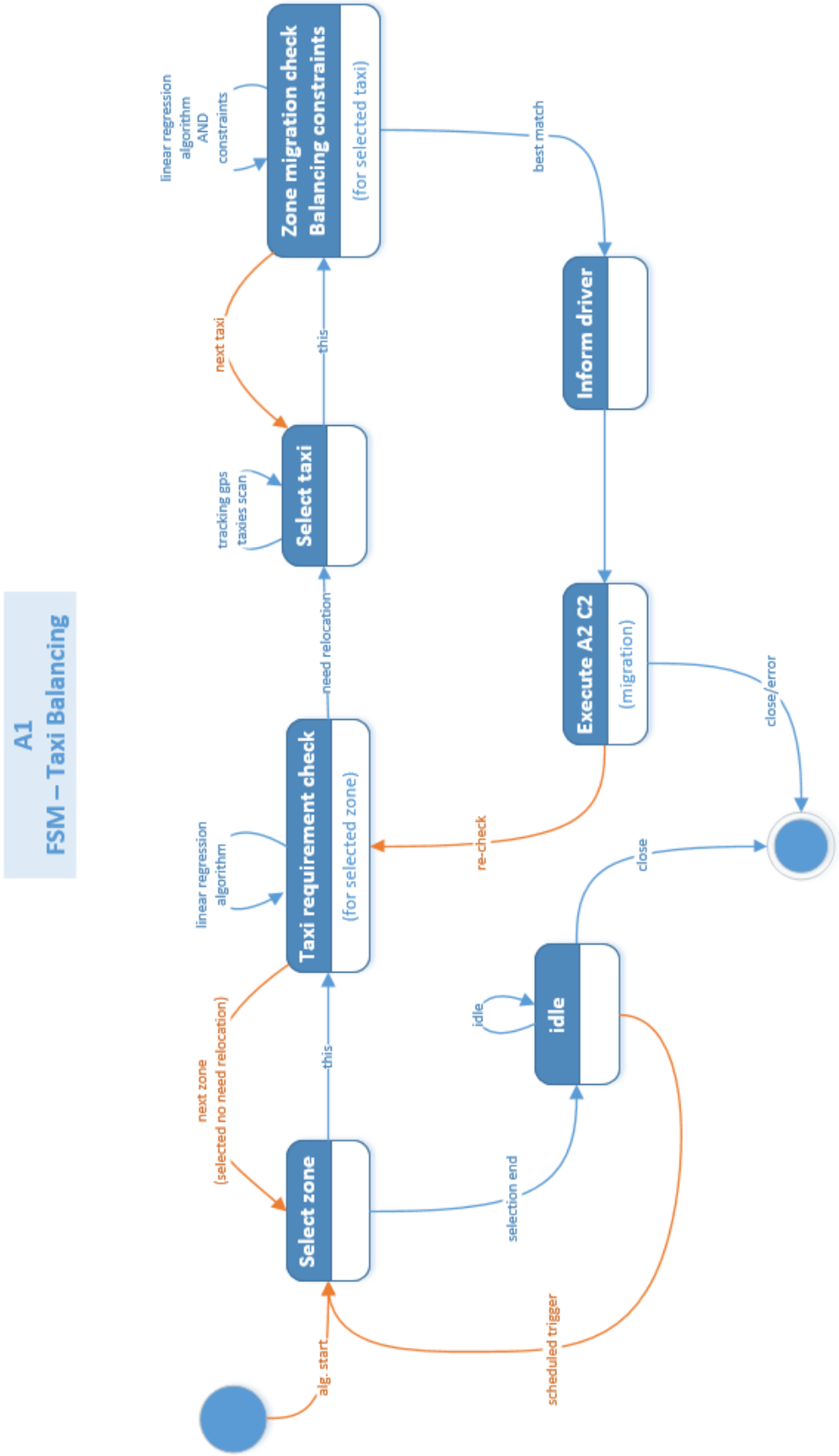
3. The system makes a cascade check: it controls the position of the selected taxi in its zone's queue (eg. *taxi.zone.queue.rear-last == taxi*) and controls if its zone is overcharged

4. If both checks are successfull the system sends a message to the driver to relocate his taxi in the new zone (by a popup message that could only be accepted).

5. If the first click is not successful, the system starts again from step 2 looking for the second nearest taxi and so on

6. At this point [A2][C2] takes care to fix queues with the method *migration*.


A1 – Sequence Diagram
Driver
A1
System
Trigger
taxi requirement check ()
[linear regression]
zone migration check()
[check if the rear-last is the taxi itself
and if the queue can donate]
selected zone
check result
selected taxi
check result
inform driver
ok
executeA2[Z-A, taxi]
[to A2 C2]
A2 C2
idle()
select zone()
select taxi()
[gps]
scheduledTrigger()
Loop
Loop
TAXI
BALANCING
SCENARIO


A1
FSM – Taxi Balancing
Zone migration check
Balancing constraints
(for selected taxi)
linear regression
algorithm
AND
constraints
best match
Inform driver
next taxi
this
Select taxi
tracking gps
taxies scan
need relocation
Execute A2 C2
(migration)
close/error
Taxi requirement check
(for selected zone)
linear regression
algorithm
re-check
close
idle
idle
next zone
(selected no need relocation)
this
Select zone
selection end
alg. start
scheduled trigger

3.2 [A2] The management of queues in zones

The management of zone queues is complex:
we decided to divide the management algorithm into two subcategories, because MyTaxiService needs to extract taxies by the queue, depending on the case, using FIFO or LIFO politics.

3.2.1 [C1] Passenger request case

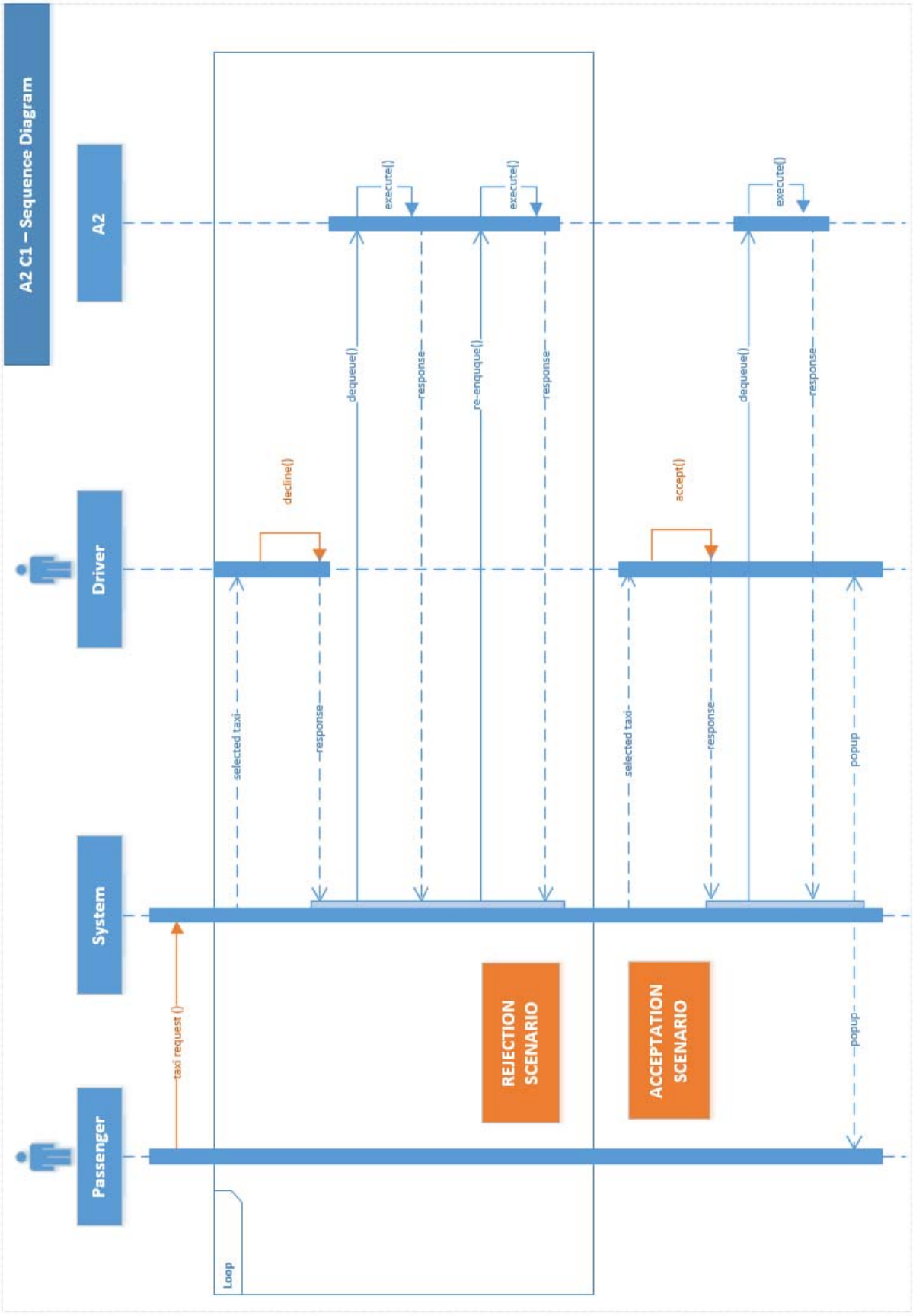
This case the algorithm uses a **FIFO** politics: the taxi in the queue for more time has the priority to answer a call.

1. A passenger calls a taxi.
2. The system look for an available driver in the passenger's departure zone.

   If the driver selected by the system accepts the request, the algorithm *dequeues* the taxi and wait the end of the ride to append this taxi in another queue.

   If the driver selected by the system declines the request, the algorithm *dequeues* the taxi and re-*enqueue* it at the end of the same queue.
3. The system searches another available driver: starts again from step 2 and so on.

Follows the Sequence Diagram for the algorithm A2 C1:


A2 C1 – Sequence Diagram
A2
Driver
System
Passenger
execute()
execute()
execute()
dequeue()
response
re-enqueue()
response
dequeue()
response
decline()
accept()
selected taxi-
response
selected taxi-
response
popup
taxi request ()
popup
REJECTION SCENARIO
ACCEPTATION SCENARIO
Loop

3.2.2 [C2] Taxi migration case

This case the algorithm uses a **LIFO** politics: it would not be correct to oblige a taxi on the top of the queue to migrate in another zone, because it would be appended in the last position of the new queue and it would lose the deserved priority.

At the same time, it would not be correct to appended this taxi on the top of the new queue because the older first taxi would lose the right to answer the next call.

For these reasons, in case of migration to another zone that needs more taxies, will be taken the taxi in the last position of the starting queue and it will be placed in the last position of the destination queue.

1. The algorithm [A1] has obliged a taxi driver to move from ZD (departure zone) to ZA (arrival zone).
   The *read-last* method gives a taxi.

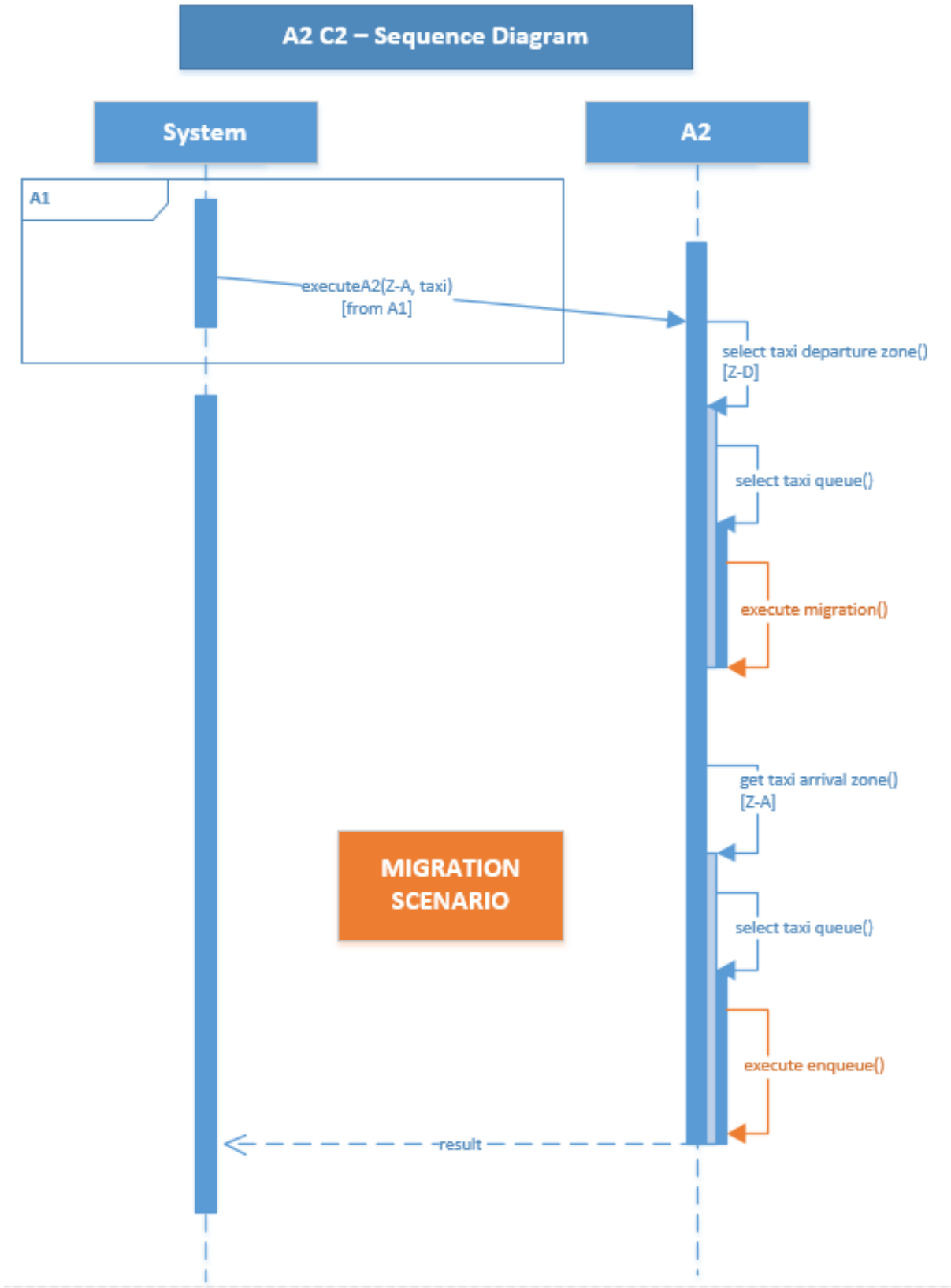
2. The algorithm reads the ZD from the taxi.

3. The algorithm accedes to the queue of the taxi's ZD and take the last taxi with *migration* method.

4. When the taxi arrives in the new zone the algorithm reads the ZA from the taxi.

5. The algorithm accedes to the queue of the taxi's ZA and *enqueue* the taxi in the last position.

Follows the Sequence Diagram for the algorithm A2 C2:


A2 C2 – Sequence Diagram
System
A2
A1
executeA2(Z-A, taxi)
[from A1]
select taxi departure zone()
[Z-D]
select taxi queue()
execute migration()
get taxi arrival zone()
[Z-A]
MIGRATION
SCENARIO
select taxi queue()
execute enqueue()
result

3.3 Algorithm appendix

Queue Management Algorithm

Notation

- *f: front element*
- *r: element after rear one*
- *N: maximum queue size*
- *Q[#]: queue array*

Thrown errors

- *QueueEmptyException*
- *QueueFullException*

Procedures

- *enqueue(x)*
- *dequeue()*
- *migration()*

- *front()*
- *rear-last()*

- *size()*
- *isEmpty()*

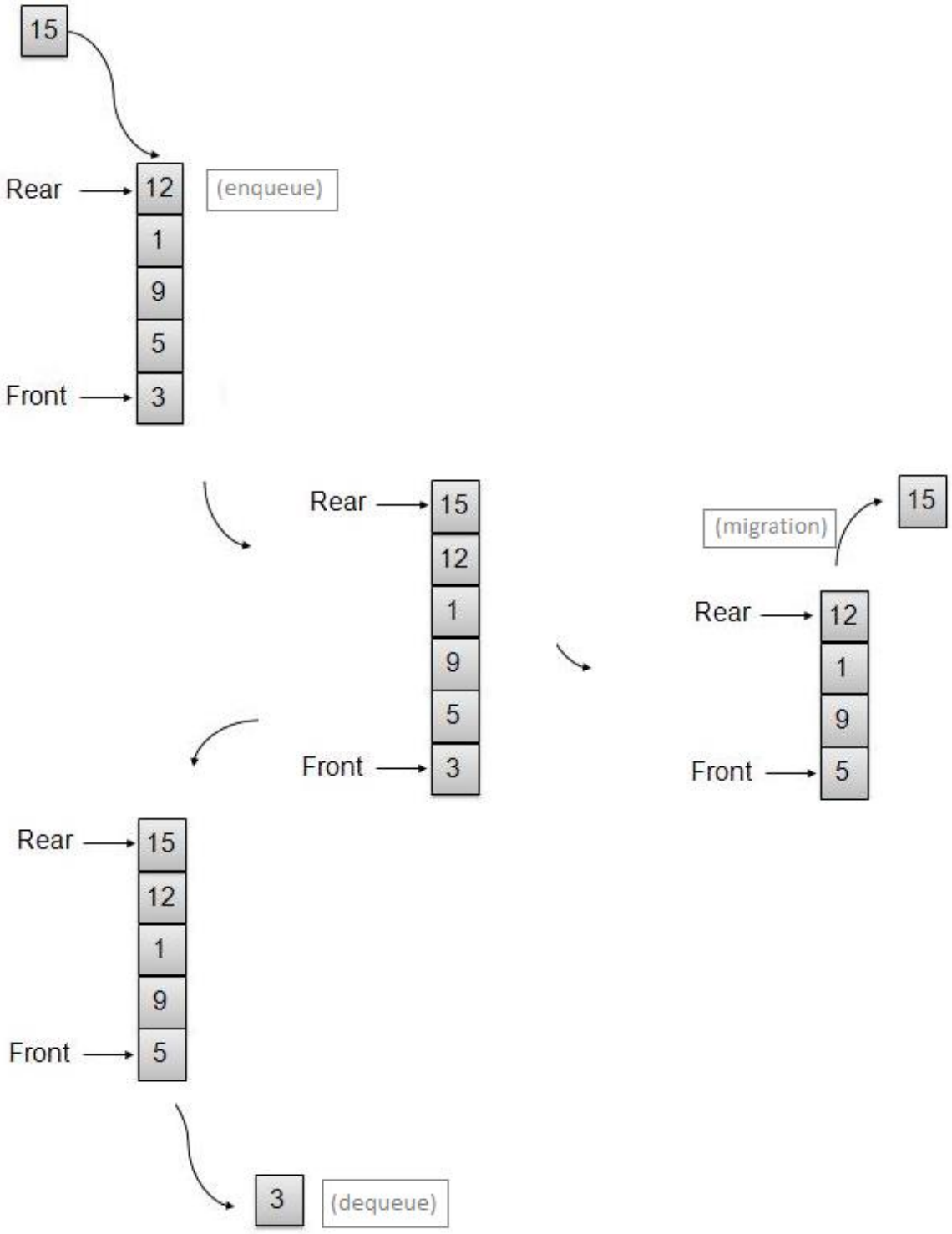
Pseudocode
O(k)

```
////////////////////////////////////////////////////
// ELEMENT INSERTION AND DELETION

procedure enqueue(x):
        if size = N - 1 then
                throw a QueueFullException
        Q[r] ← x
        r ← (r + 1) mod N
end procedure

procedure dequeue():
        if isEmpty() then
                throw a QueueEmptyException
        temp ← Q[f]
        Q[f] ← null
        f ← (f + 1) mod N
        return temp
end procedure

procedure migration():
        if isEmpty() then
                throw a QueueEmptyException
        temp ← Q[r-1]
        Q[r-1] ← null
        return temp
end procedure
```

Complexity of each procedure »

```
////////////////////////////////////////////////
// ELEMENT GETTERS

procedure front():
        if isEmpty() then
                throw a QueueEmptyException
        return Q[f]
end procedure

procedure rear-last():
        if isEmpty() then
                throw a QueueEmptyException
        return Q[r-1]
end procedure

////////////////////////////////////////////////
// QUEUE CHECK

procedure size():
        return (N - f + r) mod N
end procedure

procedure isEmpty():
        return (f = r)
end procedure
```

Example of a double-queue step logic »



4. User interface design

This is an overview of the user interface of the system.

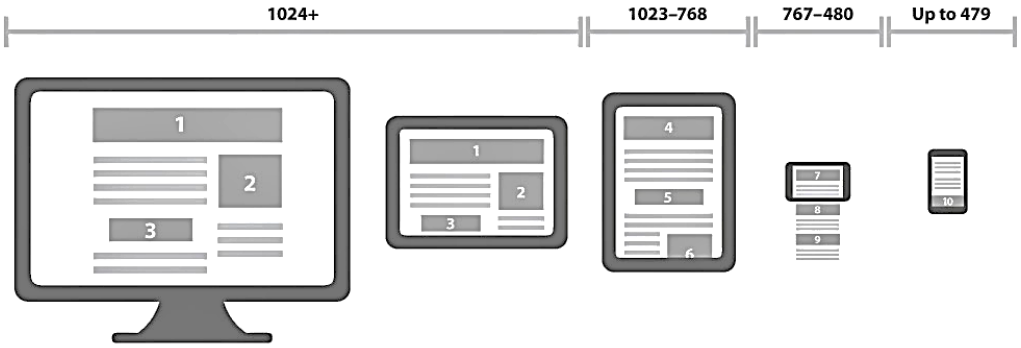
» ***Note 1***
*All the mockups for the UI are included in the section **3.1.1** of the **RASD** document, refer to:*

3.1.1.1 Login
3.1.1.2 Registration form
3.1.1.3 Service selection
3.1.1.4 Ride Request
3.1.1.5 Ride Booking
3.1.1.6 Booked Rides List
3.1.1.7 Driver area
3.1.1.8 Popups

» ***Note 2***
<u>In this document are provided some extensions for the following mockups in the RASD document:</u>

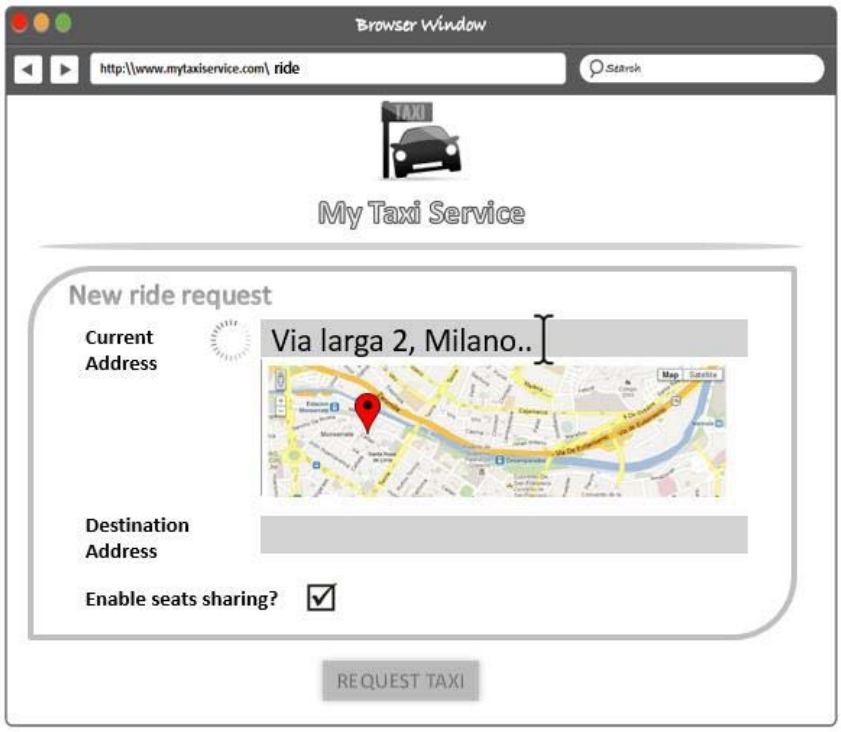
3.1.1.4 Ride Request
3.1.1.5 Ride Booking
3.1.1.8 Popups

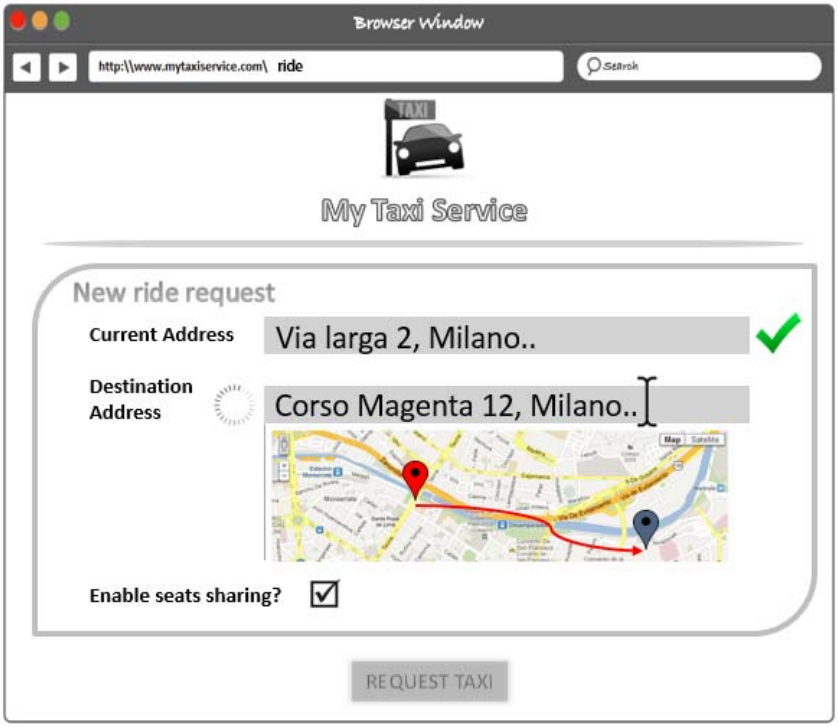
» ***Note 3***
The interface of the website is resizable, and can adapt to the screen resolution of a variety of devices (in pixels).
This provide the support for the registration of a new user, or the use of the website in case of an app-fault in a mobile-device environment.



4.1 [E1] Extension for RASD 3.1.1.4 - Ride request

In depth view of the address map selector





4.2 [E2] Extension for RASD 3.1.1.5 - Ride booking

In depth view of the date/time picker

4.3 [E3] Extension for RASD 3.1.1.8 – Popups

In depth view of some error popup

5. Requirements traceability

This section explains how the requirements defined in the chapter 3.2 of the RASD map into the design elements that are defined in this document.
In the RASD has been defined some functionality and goals that need some guarantee:

- **[F1] Allow a visitor to become a registered user.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Registration form" and "Popup".
  Managed by:
  - *View Controller*
  - *DBMS Controller*

- **[F2] Allow user to log in to application.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Login form" and "Popup".
  Managed by:
  - *View Controller*
  - *Application Controller*
  - *DBMS Controller*

- **[F3] Allow user to request an immediate taxi ride.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Service Selection (Menu)", "Ride Request and Ride Booking form" and "Popup".
  Managed by:
  - *View Controller*
  - *Application Controller*
  - *Queue Algorithm*
  - *DBMS Controller*

- **[F4] Allow user to delete an existing ride.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Booked Rides List" and "Popup".
  Managed by:

- *View Controller*
- *Application Controller*
- *Queue Algorithm*
- *DBMS Controller*

- **[F5] Allow user to see their booking data.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Booked Rides List" and "Popup".
  Managed by:
  - *View Controller*
  - *Application Controller*
  - *Queue Algorithm*
  - *DBMS Controller*

- **[F6] After login, allow driver to accept a ride.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the section "Popup".
  Managed by:
  - *Application Controller*
  - *Queue Algorithm*
  - *DBMS Controller*

- **[F7] Allow user to delete an existing ride (in case of multiple bookings).**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Booked Rides List" and "Popup".
  Managed by:
  - *View Controller*
  - *Application Controller*
  - *Queue Algorithm*
  - *DBMS Controller*

- **[F8] Allow user to request a taxi 'booking' ride.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections

"Service Selection (Menu)", "Ride Request and Ride Booking form" and "Popup".

Managed by:

- *View Controller*
- *Application Controller*
- *Queue Algorithm*
- *DBMS Controller*

- **[F9] Allow user to consult ride's details when there are more than one booking.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Service Selection (Menu)", "Booked Rides List" and "Popup".

  Managed by:
  - *View Controller*
  - *Application Controller*
  - *Queue Algorithm*
  - *DBMS Controller*

- **[F10] Allow user to share a ride.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Service Selection (Menu)", "Ride Request and Ride Booking form" and "Popup".

  Managed by:
  - *View Controller*
  - *Application Controller*

- **[F11] Allow user to see the state of their booking queue.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Driver Area", "Booked Rides List".

  Managed by:
  - *View Controller*
  - *Application Controller*

- **[F12] Allow user to see data of every booking they had accepted.**
  In this functionality, requirements are satisfied and the explanation of how

it is possible is given in the chapter 2.3 of this document, in the sections "Driver Area", "Booked Rides List" and "Popup".

Managed by:

- *View Controller*
- *Application Controller*
- *DBMS Controller*

- **[F13] After login and (ride) acceptation, application will display to taxi drivers the taxi booking for their taxi.**
  In this functionality, requirements are satisfied and the explanation of how it is possible is given in the chapter 2.3 of this document, in the sections "Driver Area" and "Popup".

  Managed by:

  - *Application Controller*
  - *DBMS Controller*

6. References

- IEEE Standard for Information Technologies – System Design – Software Design Description
- IEEE Standard System and Software Engineering – Architecture Description
- RASD (MyTaxiService – Leonardo Turchi – Sara Pisani)
- Design I – II.pdf (Slides)

7. Hours of works

Here is the time spent for redact this document:

[sum of hours spent by team's members]

+2h (20/11)
+4h (22/11)
+10h (23/11)
+10h (24/11)
+12h (25/11)
+12h (26/11)
+4h (27/11)
+10h (28/11)
+2h (29/11)
+4h (03/12)

TOTAL ~ 70 hours

- Leonardo Turchi: ~ 35 hours
- Sara Pisani: ~ 35 hours