Design **D**ocument
“MyTaxiService” Application
A.Y. 2015-1016

Ennio Visconti (mat. 790382)
Simone Zocchi (mat. 852910)
Khanh Huy Paolo Tran (mat. 852496)

December 4, 2015

POLITECNICO
MILANO 1863

Contents

1 Introduction

1.1 Purpose

This document has the **goal** of providing an overall and complete description of *myTaxiService*'s software architectural design together with algorithms and UI.
The intended audience of this document are project managers and software developers.

1.2 Scope

myTaxiService is an application born to support an existing taxi company, especially to simplify passengers' access to the service and guarantee a fair management of taxi queues. In fact it will be GPS-based and it will be able to assign a taxi to a certain call, based on the location of the taxi and of the requesting passenger, in order to minimize the waiting time.
It will also provide an option to reserve a taxi in advance and to share the ride with other passengers to save money. All these features will be available via a freely downloadable mobile application or a web application.
It will also provide an interface for the taxi drivers, in order to communicate with them more easily and to make them able to automatically share their location over time.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Dynamic queue**: it is a list of taxis which is created whenever a request must be managed.
- **Static queue**: it is a list of taxis which is associated to every zones.

1.3.2 Acronyms & Abbreviations

- **UML**: Unified Modeling Language
- **API**: Application Programming Interface
- **MVC**: Model - View - Controller
- **DTO**: Data Transfer Object
- **PSP**: Payment Service Provider

1.4 Document Structure

The document divides into 5 parts:

- Section 1: **Introduction**. It gives a description of the document underlining his goals and gives basic information about the application.
- Section 2: **Architectural Design**. It gives the various architectural views of the application.
- Section 3: **Algorithm Design**. It gives some algorithms' descriptions contained in the application.
- Section 4: **User Interface Design**. It shows application's mockups.
- Section 5: **Requirements Traceability**. It contains solutions to the RASD requirements.
- Section 6: **References**. It contains all DD's references.

2 Architectural Design

2.1 Overview

This section contains all the architectural views with their respective low level descriptions and related diagrams.

2.2 High level components and their interactions

The following block diagram presents the overall structure of the system we are going to describe. It is composed of a front end side, with a common layer and two more specific layers: one is specific to mobile and the other is to the web. The back end side presents the main application layer that provides the public API available for third party applications which is also used for the communication with the front end. It also presents a data abstraction layer to access information stored in the DBs.

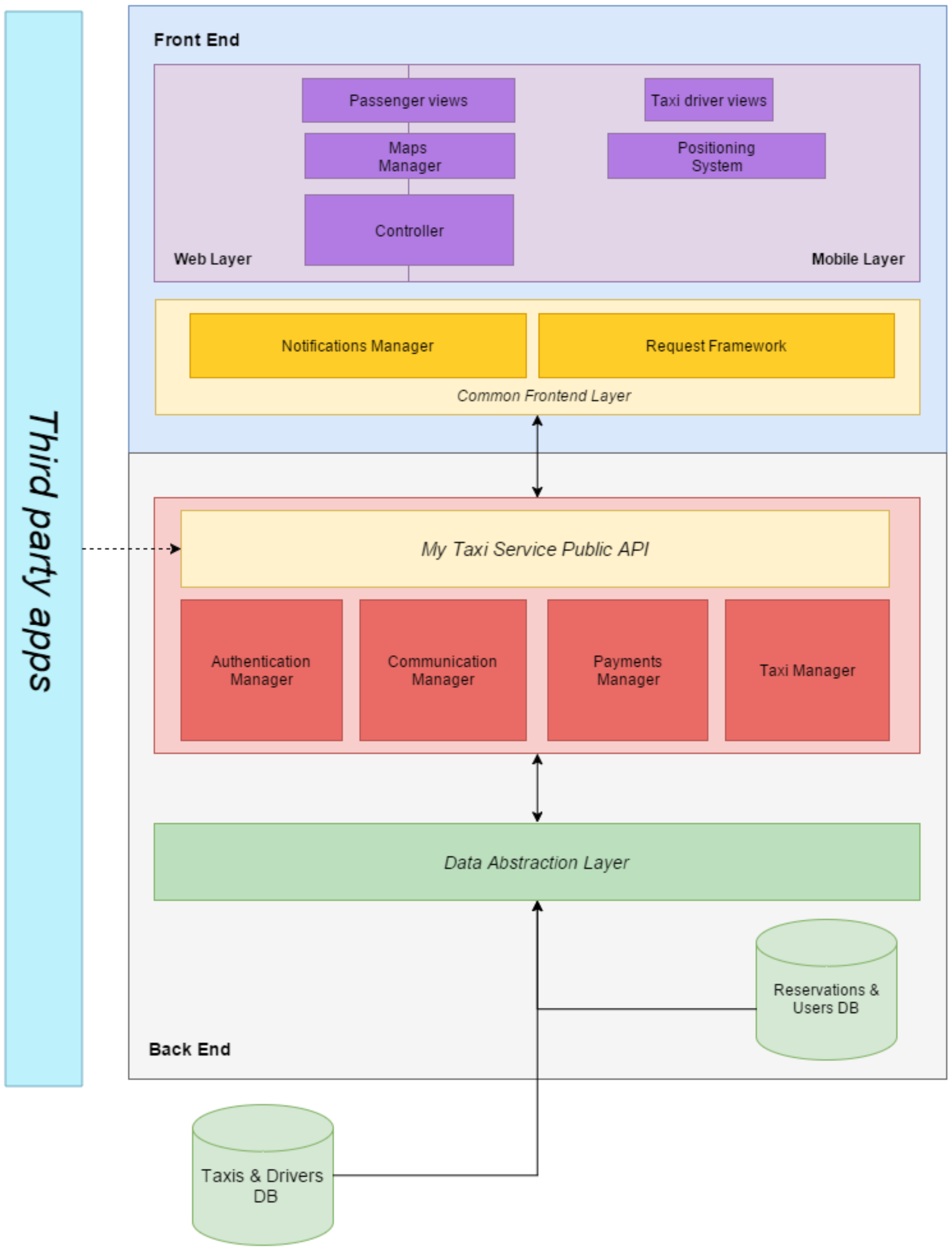
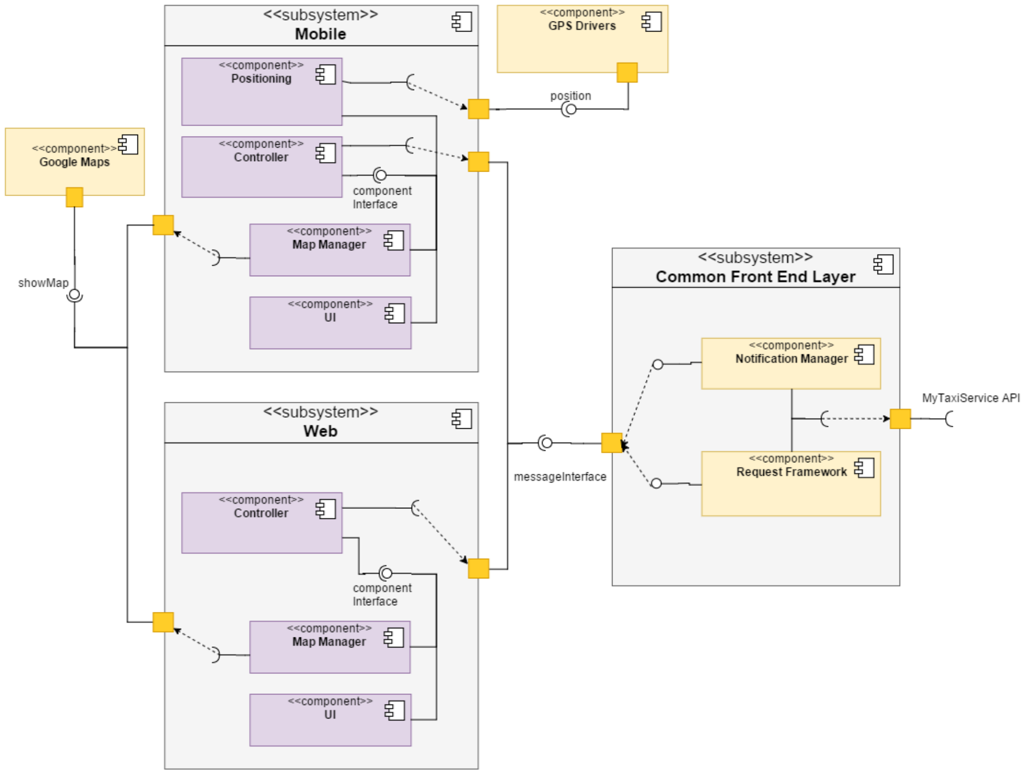


Figure 1: High Level Block Diagram

2.3 Component view

2.3.1 Front end



<<subsystem>> Mobile
<<component>> GPS Drivers
<<component>> Positioning
position
<<component>> Google Maps
<<component>> Controller
component Interface
<<component>> Map Manager
showMap
<<component>> UI
<<subsystem>> Common Front End Layer
<<component>> Notification Manager
MyTaxiService API
messageInterface
<<component>> Request Framework
<<subsystem>> Web
<<component>> Controller
component Interface
<<component>> Map Manager
<<component>> UI

Figure 2: UML Component diagram - Front end side

- **Controller**: prototype of a generic controller, it summarizes all the generic interactions between client side UI and specific components, together with routing logic and view engine.
- **UI**: view generic component. It consists of all the graphical elements which are called by the *controller* (and particularly by the view engine).
- **Maps manager**: it calls Google Maps API to show the path on screen.
- **Positioning system**: it updates the position of the device invoking the GPS drivers.

- **Request framework**: it manages the outgoing messages for the client side.
- **Notifications manager**: it manages the incoming messages for the client side.

2.3.2 Back end

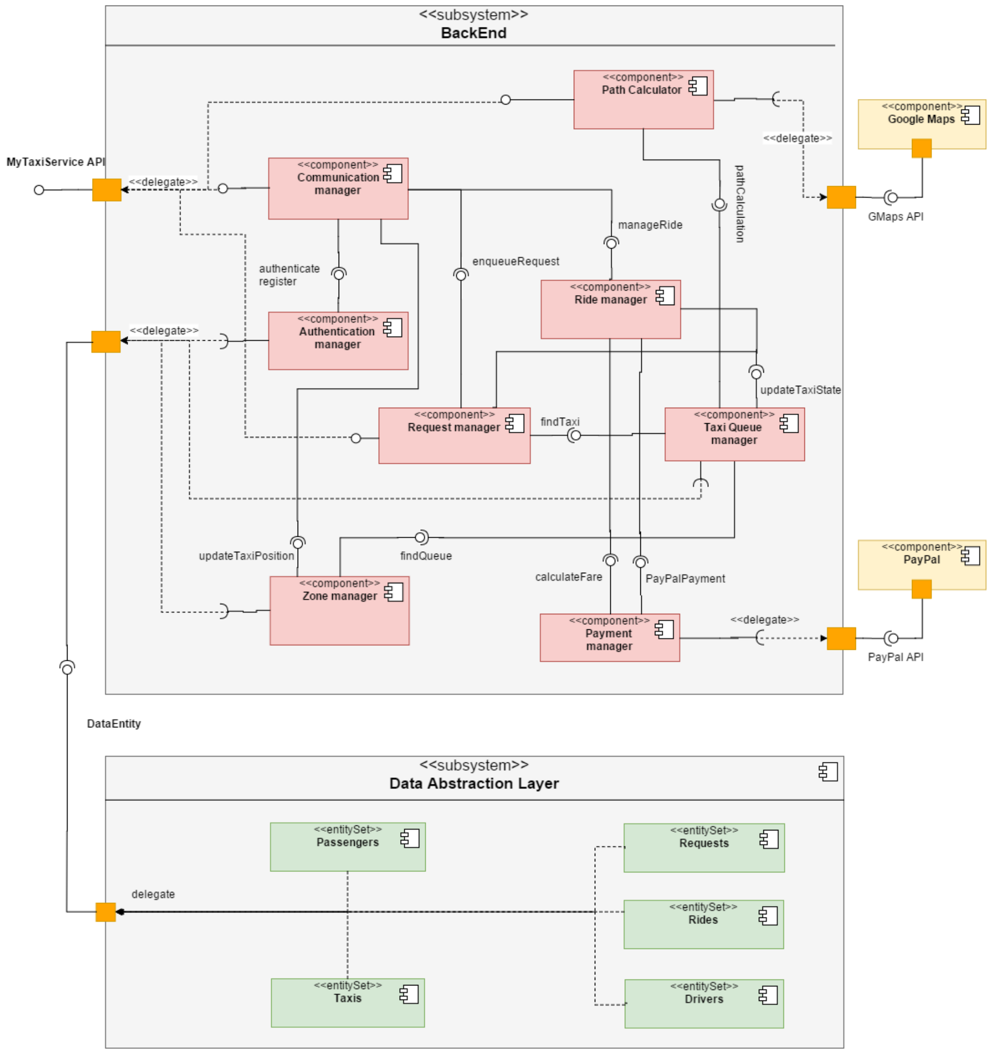


Figure 3: UML Component diagram - Back end side

- **Authentication manager**: it's delegated to execute users' login and to authenticate incoming messages. It also has to manage the registration process itself.
- **Communication manager**: it manages all the ingoing and outgoing notifications and messages to and from taxi drivers and passengers.
- **Payments manager**: it is held responsible for calculating the fare for the ride and for using the PayPal APIs for payments through the PSP.
- **Taxi manager**:
  - **Taxi queue manager**: it has to build and manage dynamic queues and to change taxis' states.
  - **Rides manager**: it manages all the ongoing rides, the start and stop process.
  - **Request manager**: it manages all rides' requests from passengers. It contains queues for immediate and future requests.
  - **Path calculator**: it manages the interaction with the Google Maps APIs to compute paths and tries to match a shared ride path with a new shared ride request's path.
  - **Zone manager**: it manages the static queues. It collects the taxi positions and it puts them in the correct queue.
- All the components in the **Data Abstraction Layer** represent the main entity of the DB.

2.4 Deployment view

In this section it will be shown a software's Deployment Diagram that describes the hardware level.
It has been adopted a 3-tier architectural model. There is a data tier with two different databases: a legacy one which contains taxis and drivers and a new one which contains reservations and passengers.
The middle layer is the logic tier which contains all the application logic. It runs on Windows Server 2008 R2.
The presentation for the mobile application is entirely on the client device whereas for a web access, it is split between the client device and on the web server. So the presentation tier lies between the client layer (device applications) and the web server layer (application server).

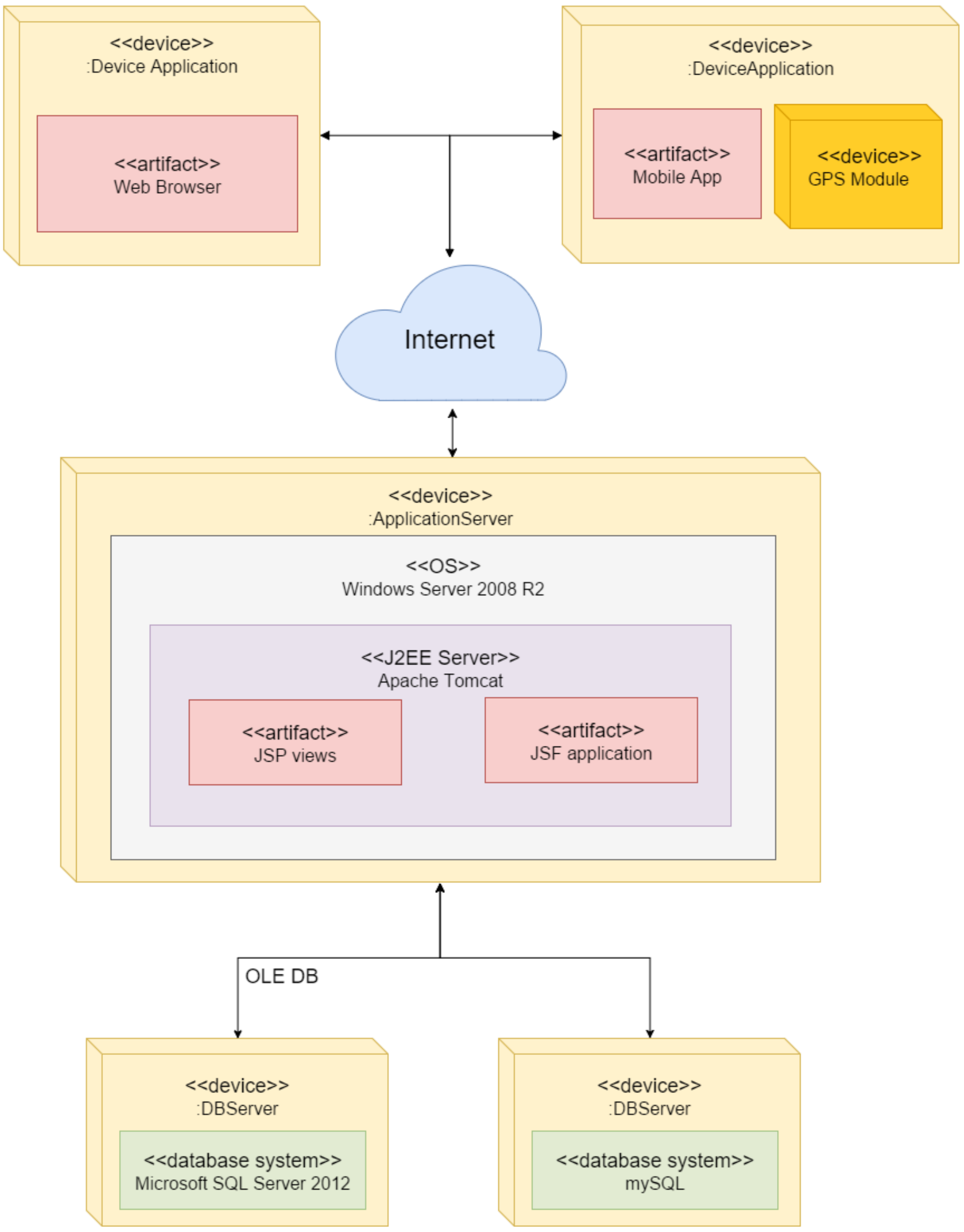


Figure 4: UML Deployment Diagram

2.5 Runtime view

This section contains some Sequence Diagrams of the application.
In particular it shows what happens whenever:

- A request arrives in the server;
- the taxi driver receives a notification for a ride;
- a request has to be handled by the request manager.

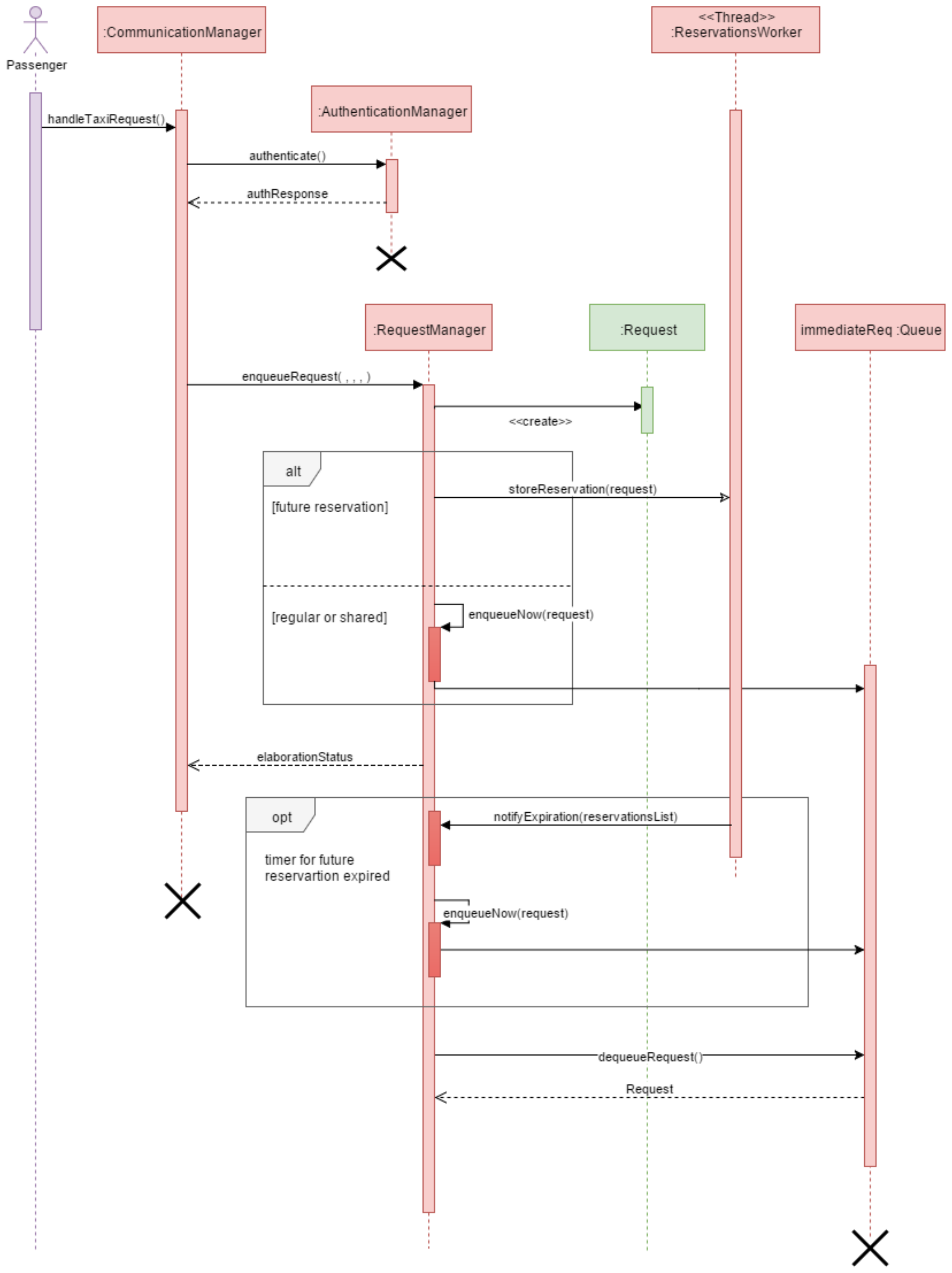


Figure 5: UML Sequence Diagram: Request arrival

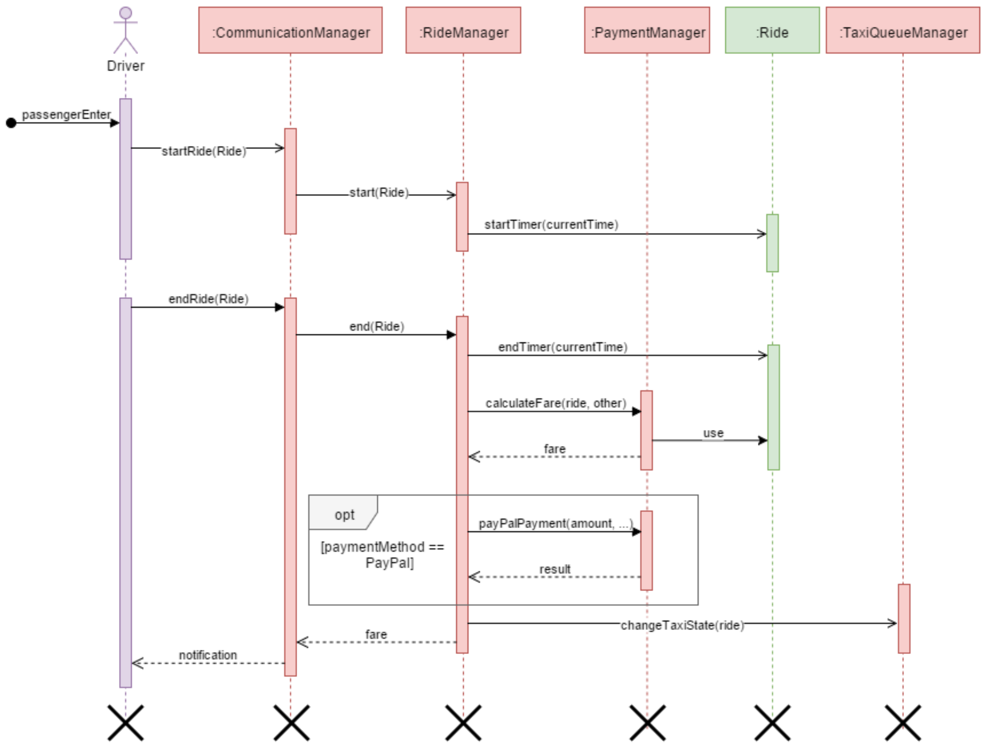


Figure 6: UML Sequence Diagram: Ride notification

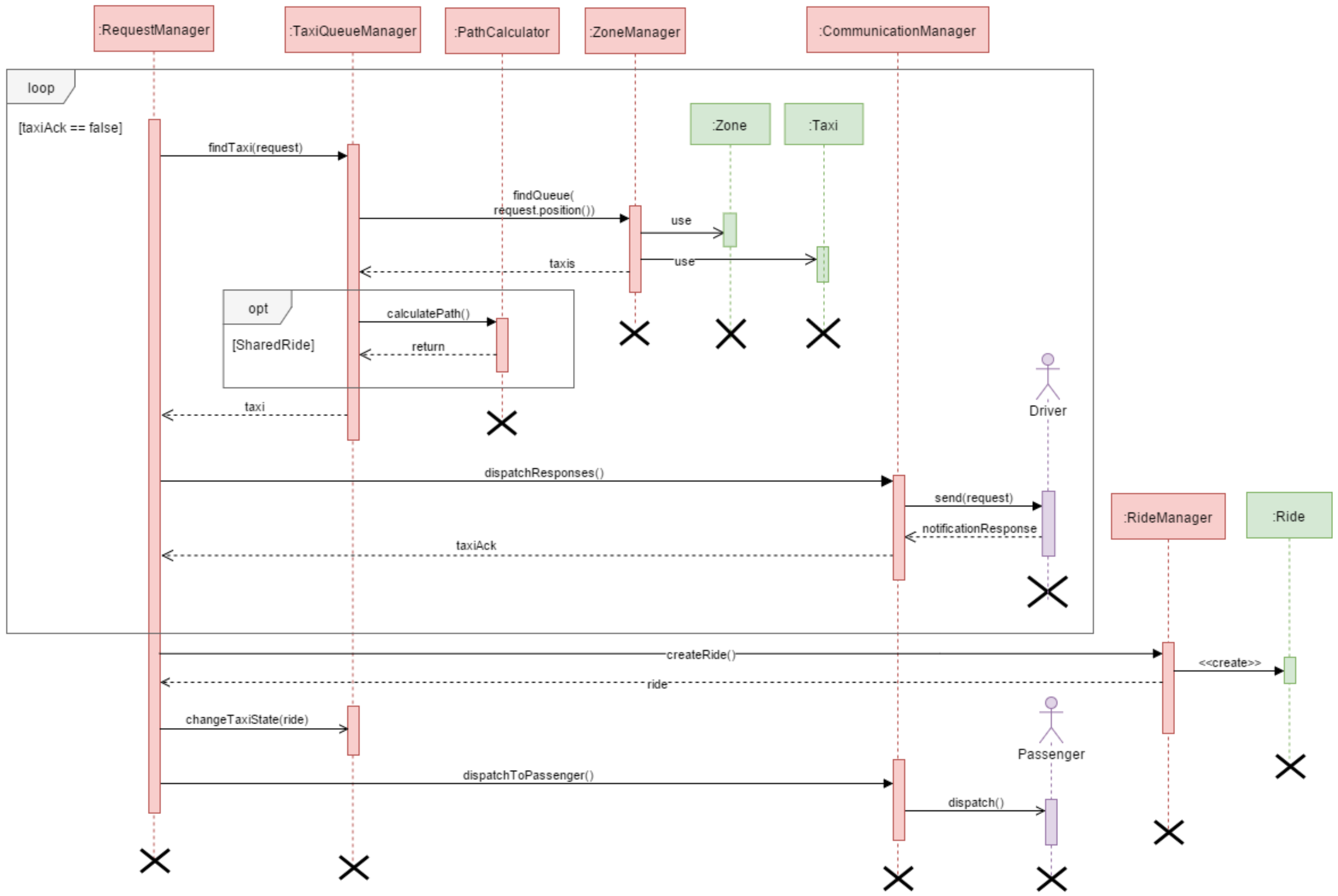


Figure 7: UML Sequence Diagram: Request handling

2.6 Component interfaces

2.6.1 Internal interfaces - Front End

- **Request framework**:
  Requires:
  - *messageInterface*: possibility to send messages from the back end

  Provides:
  - *myTaxiServiceAPI*: method to communicate with the mobile and web subsystems
- **Notification manager**:
  Requires:
  - *messageInterface*: possibility to receive messages from the back end

  Provides:
  - *myTaxiServiceAPI*: method to communicate with the mobile and web subsystems
- **Controller (web or mobile)**:
  Requires:
  - *messageInterface*: an interface to the common front end layer in order to send or receive message from the back end
- **Map manager**:
  Requires:
  - *showMap*: access to GoogleMaps API to display the maps and the paths
- **Positioning**:
  Requires:
  - *position*: access to the device position
- **GPS drivers**:
  Provides:
  - *position*: method that gives the device position

2.6.2 Internal interfaces - Back End

- **Taxi queue manager**:
  Requires:
  - *findQueue*: access to the static queues

  Provides:
  - *findTaxi*: method to build a new dynamic queue and look for a taxi
  - *updateTaxiState*: method to change a taxi state
- **Request manager**:
  Requires:
  - *findTaxi*: access to taxi list in order to look for a taxi
  - *updateTaxiState*: access to taxi list in order to update a taxi state at the ride's creation

  Provides:
  - *enqueueRequest*: method to create a request and enqueue it
- **Path calculator**:
  Provides:
  - *pathCalculation*: method to calculate a path for a shared ride
- **Ride manager**:
  Requires:
  - *updateTaxiState*: possibility to change the taxi state after a ride's ending
  - *PayPalPayment*: possibility to pay with PayPal
  - *calculateFare*: possibility to calculate a ride's fare

  Provides:
  - *manageRide*: method in order to start and end a ride
- **Authentication manager**:
  Provides:
  - *authenticateRegister*: method to authenticate a request, log in and register visitors
- **Communication manager**:
  Requires:

  - *authenticateRegister*: possibility to authenticate a request, log in and register visitors
  - *updateTaxiPosition*: possibility to know a taxi position
  - *enqueueRequest*: possibility to create and enqueue a request
  - *manageRide*: possibility to start and end a ride

- **Payment manager**:
  Provides:
  - *calculateFare*: method to calculate a ride's fare
  - *PayPalPayment*: method to pay with PayPal

- **Zone manager**:
  Provides:
  - *updateTaxiPosition*: method to update a taxi position
  - *findQueue*: method to find all the zones' queues

2.6.3 Public API

Method	**URL**	**Usage**	**Return**
PUT	/request?x={xcoord}& y={ycoord}&time={time}& seats={nseats}	Insert a new request	-
GET	/request/{id}	Retrieves a request by id	request
POST	/request/{id}/edit	Edits the request	-
DELETE	/request/{id}/delete	Delete the request	-
GET	/me	Retrieves the user's profile	user
POST	/me/edit	Edits the user's profile	-
GET	/ride/{id}/fare	Calculate the fare for the given ride	fare
GET	/taxi?xcoord={x}& y={ycoord}	Searches a taxi for the given location	taxi
GET	/taxi/shared?x={xcoord}& y={ycoord}	Searches a shared taxi for the given location	taxi
POST	/api/token	Gets an access token	token

2.7 Selected architectural styles and patterns

List of selected architectural styles:

- **Client - Server**: the application is based on a client-server architecture. The requests are sent by the client which is the passenger. Then they are processed by the server which will communicate with a taxi driver, considered as a client. Rides management are also based on client - server architecture (taxi driver will confirm the begin and the end of a ride to a server).
- **Publisher - Subscriber**: the notification system is based on this style. The passenger and driver sides are subscribed and the notifications are sent asynchronously to them.
- **MVC**: the application's design is based on a MVC pattern. This makes the separation of concerns possible. More in details it builds a clear distinction between the data models with their database interactions and the logic part.
- **DTO**: the application is also based on a DTO architectural pattern. It is used for the request handling.

List of selected design patterns:

- **Singleton**: the *request manager* will be instantiated as a singleton.
- **Observer**: the observer pattern is used upon request handling and calling the ride manager. In particular is useful at a reservation timer's ending.
- **Factory method**: the factory method pattern is used to create new requests and new rides. It helps to not specify the request type immediately.
- **Visitor**: this pattern permits to select the right algorithm during a taxi search for based on the request kind. Specifically it's used between the *requirements manager* and the *taxi queue manager*.

2.8 Other design decisions

The taxi management is implemented using two different types of queues: static and dynamic.

- The *static queues* are associated to a specific zone and contain taxis that are physically in the relevant city area. These queues are managed by the *Zone Manager* component that updates in real time the taxi position inside the queues gathering their GPS position.
- The *dynamic queues* are built when a request is dispatched to the system from the *Taxi Queue Manager* component. The area associated to the zone is of 2 km^2 around the position indicated in the request. The taxis for the queue are taken from the static queue that overlap the dynamic area just built.

The requests are managed using queues as well.
If the incoming request is a reservation, it is stored in a specific reservations queue. A thread puts the reservation in the immediate requests list when there are 10 minutes left to reservation time.
The immediate requests are saved in another immediate requests queue and whenever an element is added to the latter it's popped and managed.

3 Algorithm Design

This sections present some algorithms used within the application.

3.1 Taxi search for algorithm

3.1.1 Standard Taxi Request

The following algorithm looks for all the taxis nearby the request's location and chooses the nearest available, preferring the ones which are in the request's zone.

Algorithm 1 Request Handling Algorithm

```
function StandardTaxiRequest(passenger, r)
    $b$ = CircularBall($passenger.\mathbf{x}, r$)    ▷ returns the list of close enough and available taxis
    if b.isEmpty() then
        return StandardTaxiRequest(passenger, $r + \epsilon$)
    else
        for t in b do
            if t.zone == passenger.zone then
                return t
            end if
        end for
        return b[0];
    end if
end function
```

3.1.2 Shared Taxi Request

The following algorithm given the ideal path to passenger's destination, for each point of the path it calculates the distance from every shared taxi's path.
At the end it sorts the list of available shared taxis by decreasing distance and it takes the first if the taxi doesn't exceed the maximum passenger's number allowed. If it does and there are no other shared taxi in the list, the regular request algorithm is called instead.

Algorithm 2 Shared Request Handling Algorithm

```
 1: function SharedTaxiRequest(passenger, r)
 2:     $s$ = SharedTaxisAvailable ▷ returns the array of all available shared taxis
 3:     $ip$ = IdealPath($passenger.\mathbf{x}, passenger.destination.\mathbf{x}$)
 4:     for p in ip do
 5:         $b$ = CircularBallShared($p.\mathbf{x}, r$)
 6:         for t in b do
 7:             $s[t] = s[t]$ + Distance($p, t.path$)
 8:         end for
 9:     end for
10:     s.SortDescending
11:     for t.distance in s do
12:         if t.distance <= MAX_DISTANCE && t.distance != 0 then
13:             return t
14:         end if
15:     end for
16:     return StandardTaxiRequest(passenger, r)
17: end function
```

3.2 Fare calculation for taxi rides

The following algorithm is used to calculate the fare for a ride when it's terminated.
currentRide is a parameter which represents the terminated ride for which the fare calculation is required.
otherRide is a list containing the other shared rides of the taxi's current trip that hasn't ended before the beginning of *currentRide*. It is empty in a regular ride's case.
A ride has the starting and the ending time (if ended).
next function finds the following instant to the one passed as parameter. In this particular algorithm the temporal instants are just starting or ending time of a ride.
The algorithm is based on the number of passengers on board and the time interval of the ride.

Algorithm 3 Fare Algorithm

```
function CalculateFare(currentRide, otherRides:List)
    $totPass = currentRide.pass$
    $fare = 0$
    $currT = currentRide.start$
    $nextT = \text{NEXT}(currT)$
    for $ride$ in $otherRides$ do        ▷ Update totPass to the number of passenger onboard at the beginning of the ride in object
        if $ride.start < currentRide.start$ then
            $totPass = totPass + ride.pass$
        end if
    end for
    while $nextT! = currentRide.end$ do
        $c = \text{Cost}(currT)$        ▷ Return the cost per minute depending on specific polices
        $fare = fare + (nextT - currT) * c * currentRide.pass/totPass$
        $fare = fare + fare * k$
        if $nextT$ is the end time of one of the ride $r$ in otherRide then
            $totPass = totPass - r.pass$
        end if
        if $nextT$ is the start time of one of the ride $r$ in otherRide then
            $totPass = totPass + r.pass$
        end if
        $currT = nextT$
        $nextT = \text{NEXT}(currT)$
    end while
    $fare = fare + (nextT - currT) * c * currentRide.pass/totPass$
    return $fare$
end function
```

4 User Interface Design

The mock-ups were already described in the RASD document. In this section are presented two UX diagrams that describe the whole user experience.

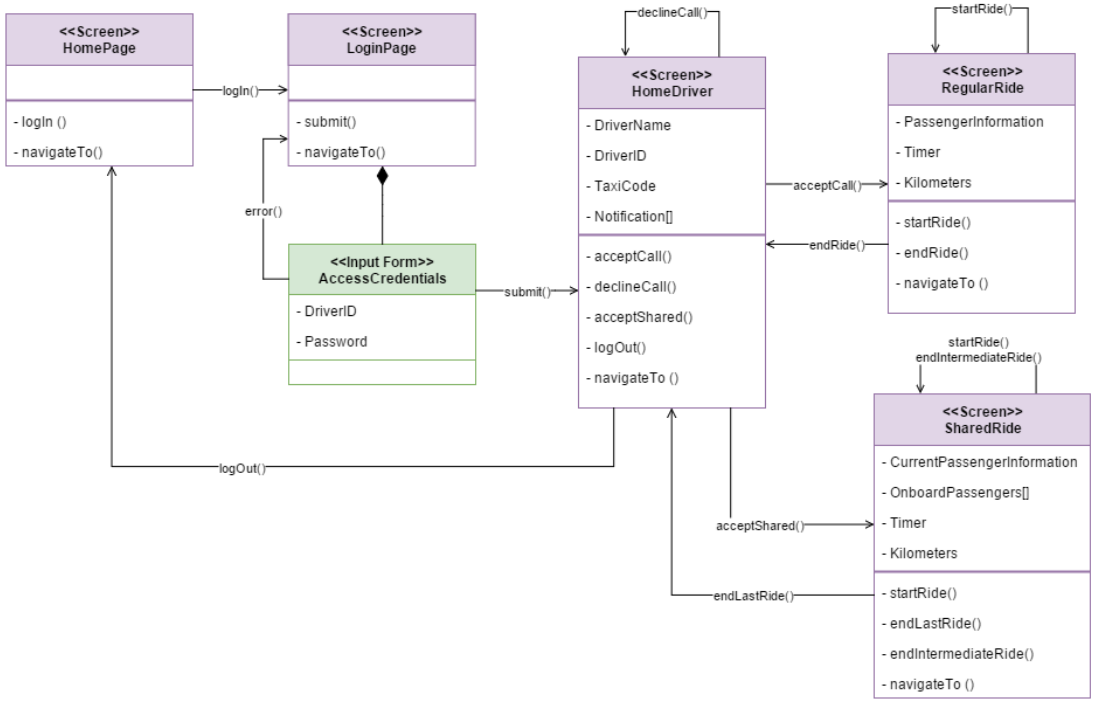


Figure 8: UX Diagram for driver interface

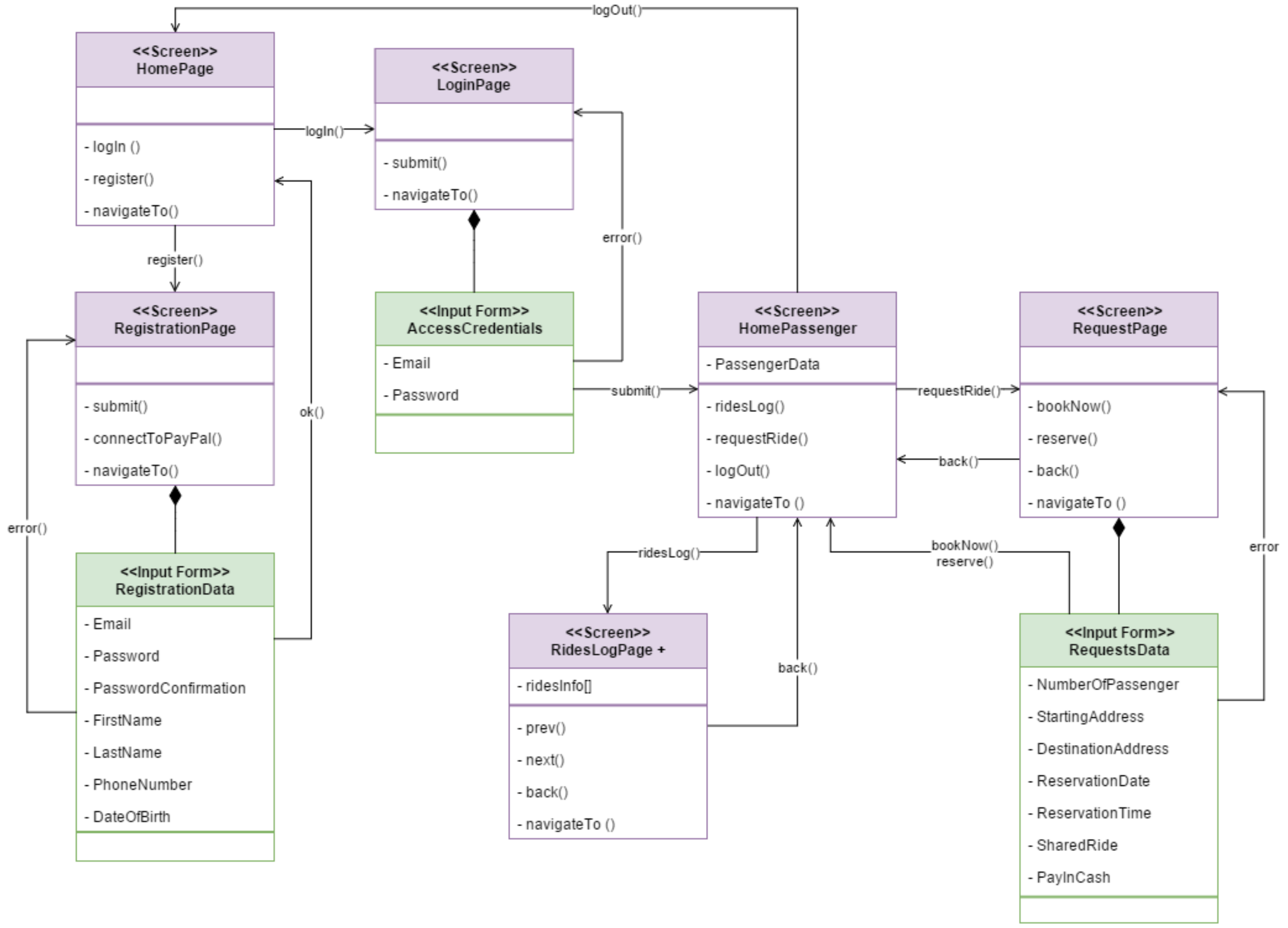


Figure 9: UX Diagram for passenger interface

5 Requirements Traceability

5.1 Visitor requirements

- ***Registering***
  - *Visitor must enter a valid email*: the application checks if it's possible to send a message to the specified email.
  - *Visitor must enter a valid phone number*: the application checks through a line test: an aborted call is made to the subscriber line.
  - *Visitor must be adult in order to register*: the application checks the age calculating it with the inserted date of birth.

5.2 Passenger requirements

- ***Pay with PayPal***
  - *Passenger must have connected his account with PayPal*: the application checks in the database if a passenger is related to a PayPal account.
- ***Request a new taxi ride***
  - *A passenger must have not requested another taxi at the same time*: the application has the list of current active requests and they must be related to different passenger. During a new reservation's creation, the application checks if there are reservations for the same time and it denies the creation of it if one belongs to the same passenger.
- ***Look rides log***
  - *Passenger must have booked at least one ride*: the application looks in the passenger's ride, reservations and requests lists and if there are no element linked to the passenger, then it denied this functionality.

5.3 Taxi driver requirements

- ***Manage a ride notification***
  - *The taxi related to the driver is not busy or shared*: the application checks if the taxi's status related to the driver isn't busy or shared. In fact every taxi has a status attribute.
  - *The taxi driver must answer the call otherwise after 15 seconds the call is automatically refused*: the application starts a timer when it sends the request to the taxi driver. If it doesn't receive an answer, it considers the request as refused.

- ***Manage a ride notification - Accept a shared ride notification***
    - *No passenger must be in the car. Besides the first shared ride notification, the others are automatically accepted until the end of the ride*: the application checks the taxi's status; it must be *free*.
- ***Manage a ride notification - Decline a ride notification***
    - *Taxi must not be shared*: the application checks the taxi's status; it must be not *shared*.
- ***Start a ride***
    - *There must be a pending request from the passenger*: the application checks if there is a ride linked to the taxi with the ride's timer not already started.
- ***Finish a ride***
    - *The taxi related to the taxi driver is busy or shared*: the application checks the taxi's status; it must be *busy* or *shared*.
    - *The ride must have been started*: the application checks if there is a ride linked to the taxi with the ride's timer already started.

5.4 Queue management requirements

- *A taxi is stored in one and only one queue*: the application checks the queues and verifies they don't contain taxi duplicates.
- *A taxi has one and only one status*: every taxi has only a status attribute.

6 References

- IEEE Std 1016-2009, IEEE Standard for Information Technology-Systems Design-Software Design Descriptions
- Requirements Analysis & Specification Document - ”myTaxiService” Application, AY 2015-2016; Ennio Visconti, Simone Zocchi, Khanh Huy Paolo Tran
- Specification document: Software Engineering 2 Project, AY 2015-2016