POLITECNICO
MILANO 1863

Software Engineering 2 Project

MyTaxiService

PART 2:

Design Document

Valeria Deciano 858479
Fabio Calabretta 852717

1/12/2015

SUMMARY

1. INTRODUCTION

1.1 Purpose

The purpose of this document is to present the architecture of the system analyzed, defining the various components and how they interact each other.

1.2 Scope

The scopes of this document are the design and architectural choices. We analyze in particular the application structure, starting from the high level architecture to the determination of the individual components and the patterns.

1.3 Definitions and Acronyms

As in the RASD, we want to explain the specific meaning that some words have in our document.

RASD: requirements analysis and specification document

DD: design document

User or Client: is a generic client that uses the taxi service.

Unregistered user: is a user who is not signed-up into the system; nevertheless he/she can ask for a taxi, but he/she cannot make reservations.

Registered user: a signed up user; the system associates to this kind of customer some information such as username, password, name, surname, email, phone number and credit card.

Authenticated user: is a user who has logged into the system.

Non-authenticated user: is a user who is not logged into the system, but he could be registered or not.

Queue: it represents the list in which each taxi is inserted as soon as it becomes available. The queue is relative to a specific area.

Zone: is a city area of approximately 2 km^2 in which the taxis are located. Every zone has its taxi queue.

Reservation: is only used for requests that have the following characteristics: departure, destination and schedule time.

Request: is used to indicate services that are required in real time.

Route: is the path taxi do to transport the passenger to a destination point.

1.4 Reference Documents

The Reference Document we used to write the Design Document is RASD written previously.

1.5 Document structure

Five chapters compose the document

1. Introduction: in this chapter there is a description of the scope and purpose of this document and, also in general of the system. There is some general information, like the specification of some words used, the reference documents and the structure of the document.
2. Architectural design: in this section we describe the system and its decomposition in sub-systems. Moreover, we have also described the interaction between them.
3. Algorithm design: in this section, we want to present some important algorithms.
4. User interfaces design: in this part there is a description of user interfaces.
5. Requirements traceability: in this section we want to explain how the requirements, we have previously characterized in RASD, map into the design element that we defined.

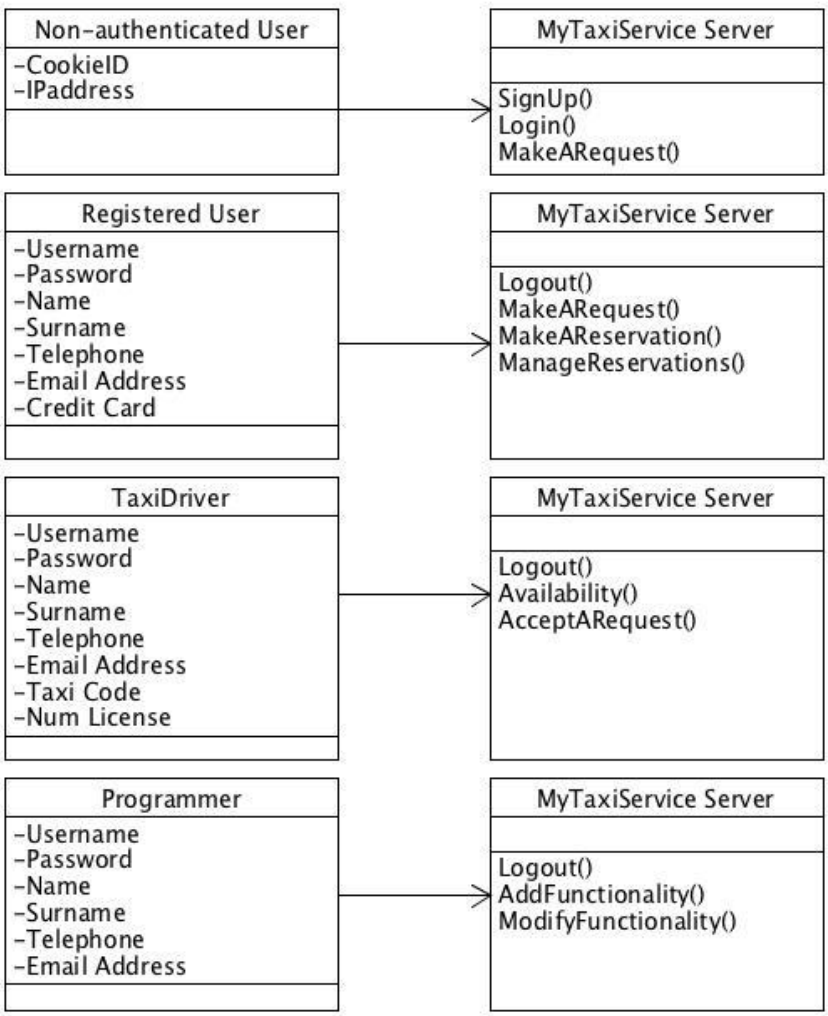
2. ARCHITECTURAL DESIGN

2.1 Overview

In this chapter we want to describe our architectural choices to guarantee the functional specifications of the system. Consequently, the system architecture is split into levels in order to determine the sub-components forming the system and the interactions between them.

2.2 High level components and their interaction

The following UML shows the main high-level components of the architecture and how each of them interacts with the server system. A list of attributes is associated to each component, while on the server side, there's a list of possible methods that can be invoked according to the specific kind of user who is active at that time. We supposed that the communication is synchronous because the user, after a request to the server, attends the answer.

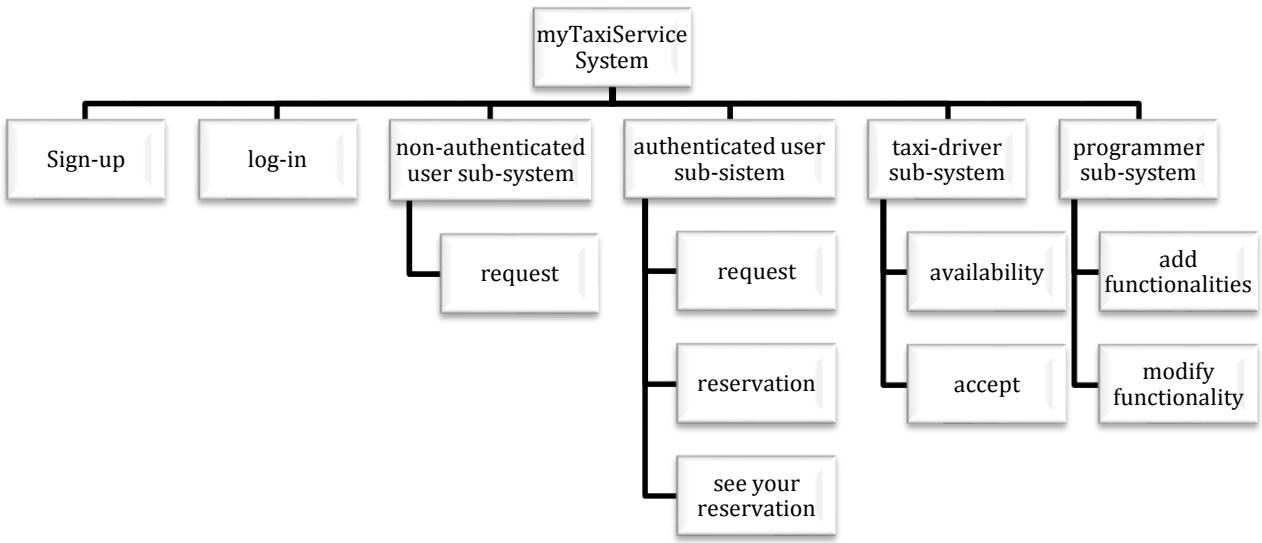


2.3 Component view

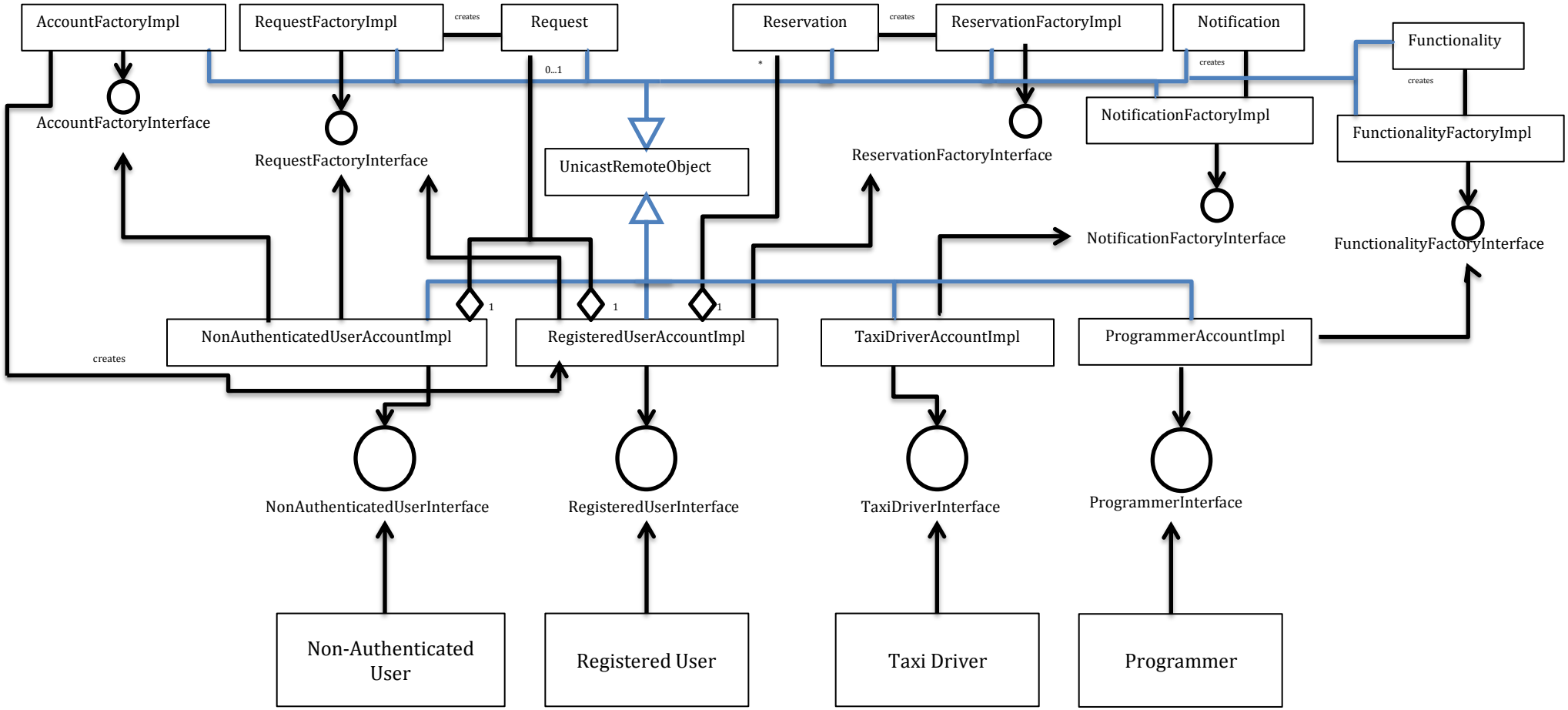
We have decomposed the system to define some sub-systems. In this way the functionalities of the system are logically separated. Besides, we define the relation among the sub-systems.

We have these sub-systems:

- Sign-up sub-system
- Log-in sub-system
- Non authenticated user sub-system
    - Request sub-system
- Authenticated user sub-system
    - Request sub-system
    - Reservation sub-system
    - See Your Reservations sub-system
- Taxi-driver sub-system
    - Availability sub-system
    - Accept sub-system
- Programmer sub-system
    - Add functionality
    - Modify functionality

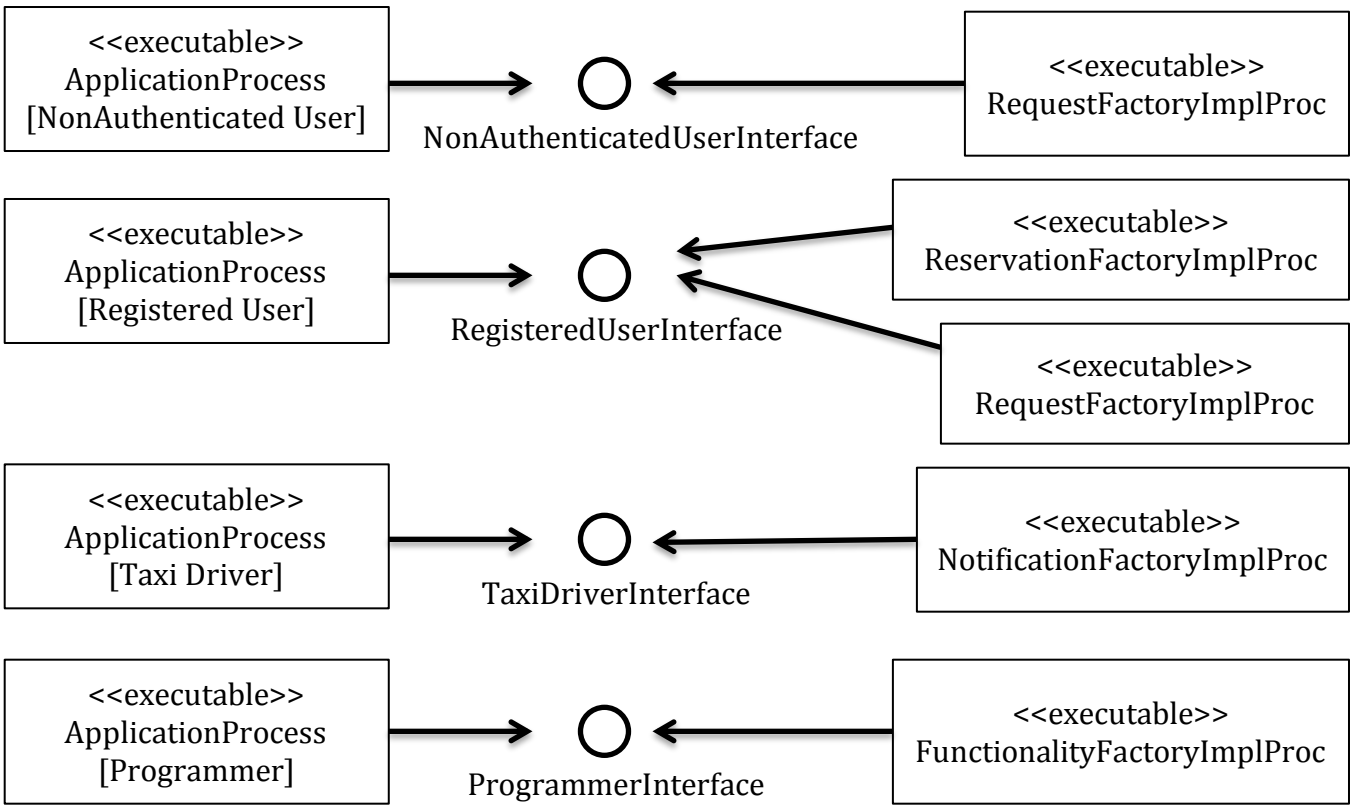


The following diagram presents all the interfaces provided for each kind of user and how the sub-components are related to each other.


AccountFactoryImpl
RequestFactoryImpl
creates
Request
Reservation
creates
ReservationFactoryImpl
Notification
Functionality
creates
creates
0...1
*
AccountFactoryInterface
RequestFactoryInterface
UnicastRemoteObject
ReservationFactoryInterface
NotificationFactoryImpl
FunctionalityFactoryImpl
NotificationFactoryInterface
FunctionalityFactoryInterface
1
1
1
NonAuthenticatedUserAccountImpl
RegisteredUserAccountImpl
TaxiDriverAccountImpl
ProgrammerAccountImpl
creates
NonAuthenticatedUserInterface
RegisteredUserInterface
TaxiDriverInterface
ProgrammerInterface
Non-Authenticated User
Registered User
Taxi Driver
Programmer

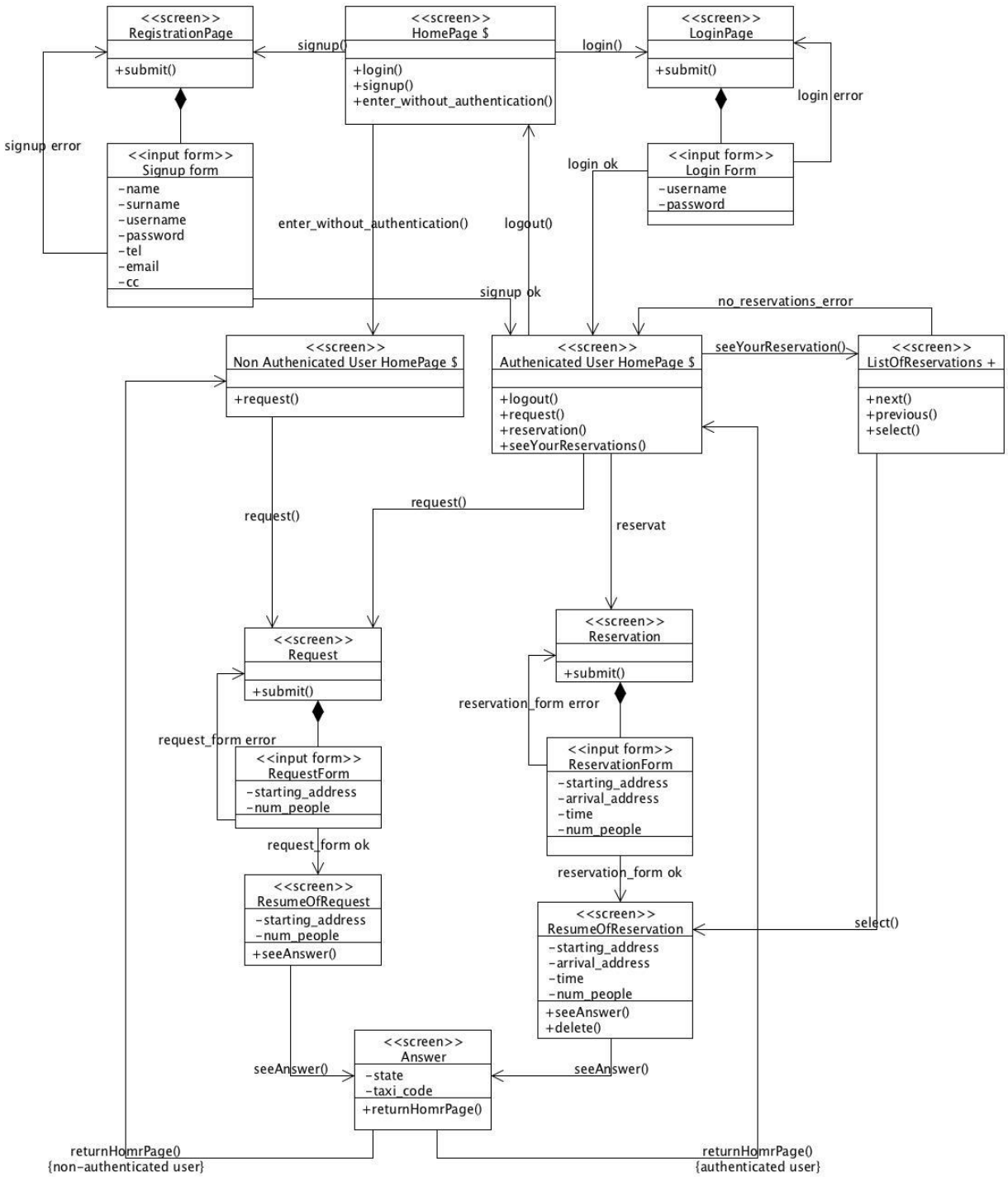
2.4 Deployment view

In this section we have identified the artifact that need to be deployed in order to make the system work. In our case, the customer, to use the service, must run an application, that could be mobile or web, depending on which device is using. According to the functionality used by the user, the process related will automatically run on the server side.

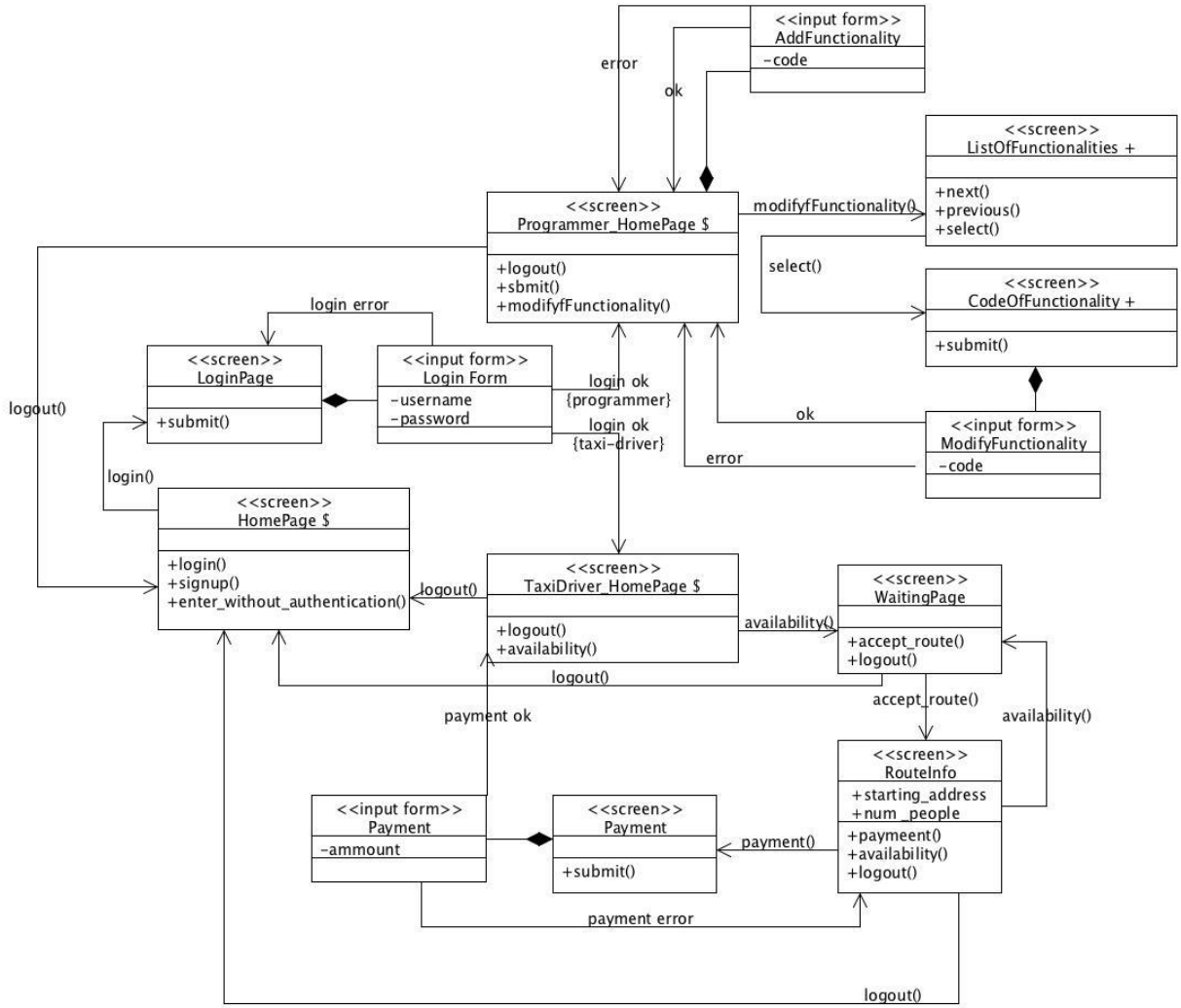


2.5 Runtime view

The two diagrams below are UX diagrams and they show the navigation in MyTaxiService website or application.
This first diagram shows all the screens that a generic user can navigate and all connections between them.



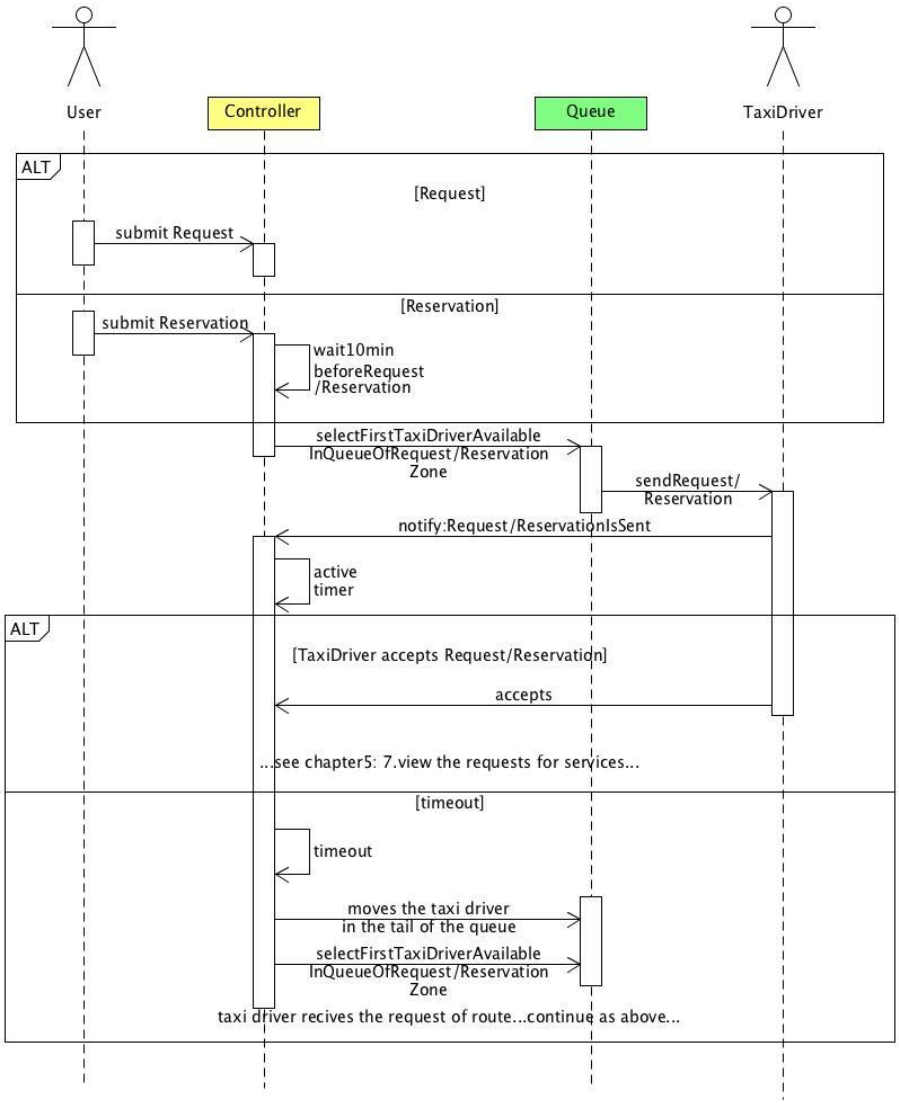
The same thing is done for the programmers and taxi drivers



In this UX diagram, the Home Page screen is the same of the previous one, in fact we don't explain some methods like: enter_without_authentication() or signup().
Moreover, we assume that these two kinds of users (taxi-driver and programmer) are already registered into the system because we must be sure about their identity. Thus, if there's a new taxi-driver who wants to enroll, he can asks to the senior management of the company, and if he is qualified, a programmer can insert him into the database and gives him the credentials to access into the system.

The screens with the stereotype <<screen>> represent all the possible screens.
The screens with the stereotype <<input form>> represent the information we put into the system.
Also, in a correct UX diagram we should have insert, in all screens a method calls NavigateTo(), that we assume to be implicit.

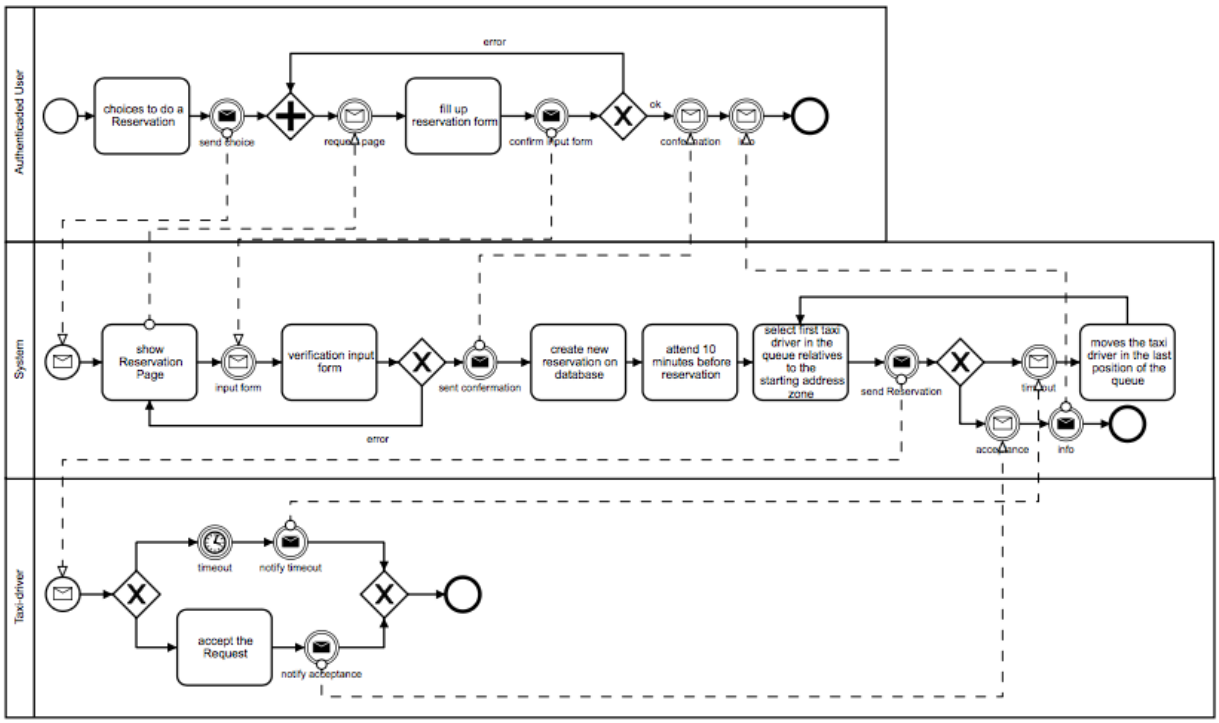
We want to show, through a sequence diagram, the interaction between the user and the taxi driver:



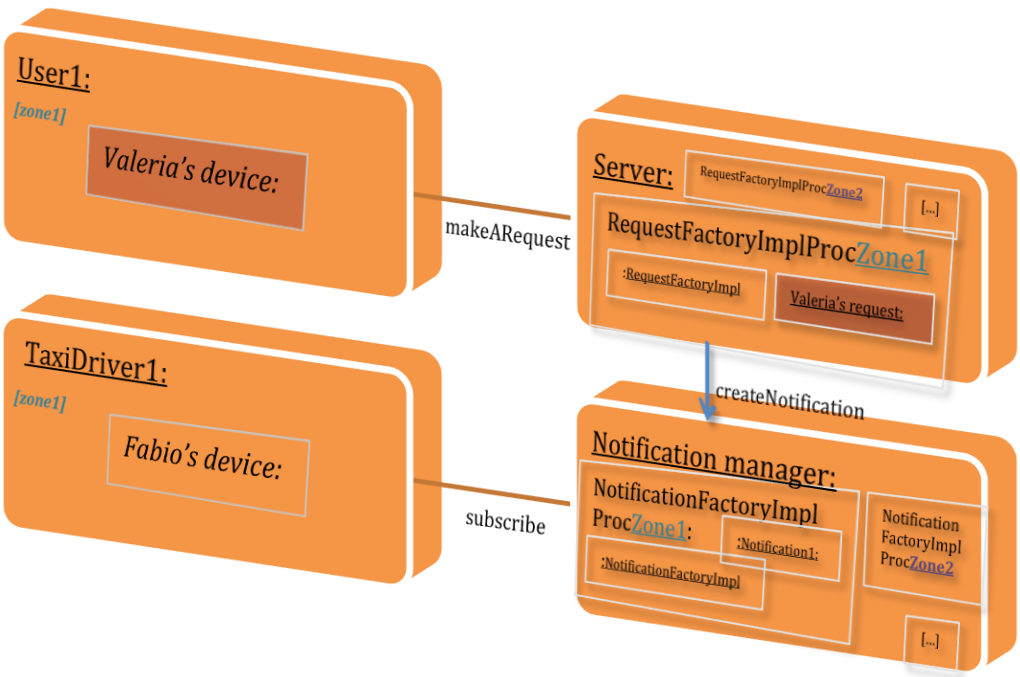
Other sequence diagrams are in chapter 5

In the following BPMN diagram we show the interaction between Authenticated User, System and Taxi driver if an Authenticated User do a Reservation.

But, a similar diagram can be done also if a generic User wants to do a Request.



Here there's an example of how the components behave at runtime in order to accomplish a request by a user. After receiving the request, the server creates a notification and forwards it to the taxi driver, which is registered virtually in the "notification manager", to ensure that can be notified in time.



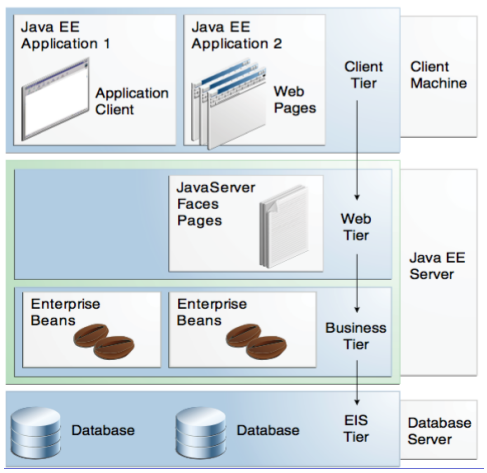
2.6 Component interfaces

Now is shown the specification of interfaces:

```
public interface NonAuthenticatedUserInterface{
      UserID signUp(String nickname, String password, String email);
      UserID login(String nickname, String password);
      boolean makeARequest(String address, String telephone);
}

public interface RegisteredUserInterface{
      boolean logout();
      boolean makeARequest(String address);
      boolean makeAReservation(String address, String day, String time);
      List manageReservations();
}

public interface TaxiDriverInterface{
      boolean logout();
      boolean acceptARequest();
}


public interface ProgrammerInterface{
      boolean logout();
      void addFunctionality();
      void modifyFunctionality();
}
```

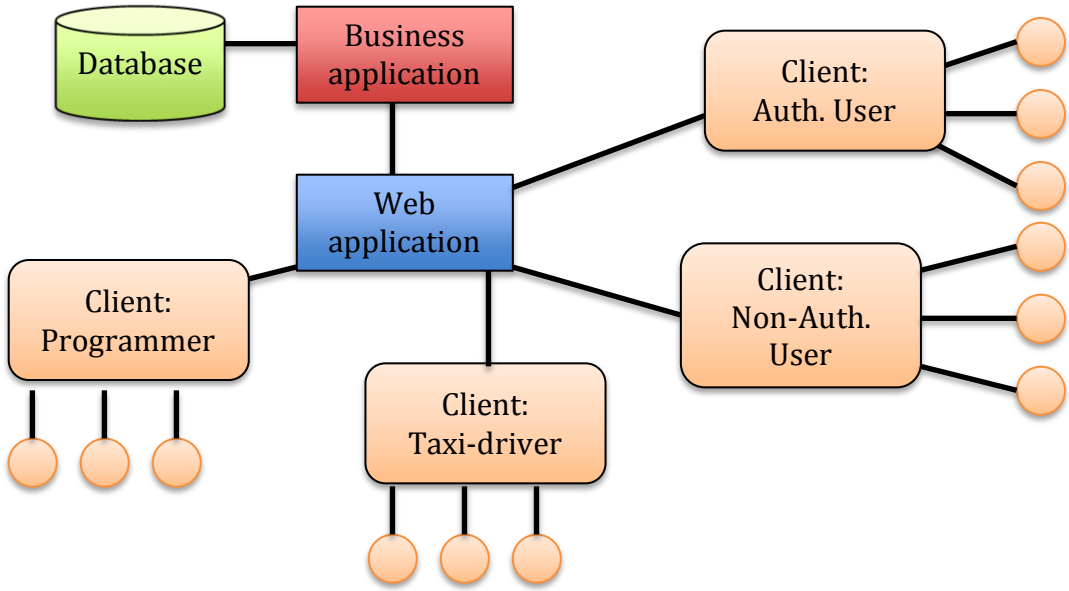
2.7 Selected architectural styles and patterns

We choose to use the Java Enterprise Edition (JEE) Architecture that has four tiered:



- **Client Tier**: that contains the Application Clients and Web Pages. This part communicates with users who can use web browsers and also mobile application.
- **Web Tier**: in this level, the data contained in the client tier requests are forwarded to the business tier where they are elaborated and after sends to the client.
- **Business Tier**: it contains the business logic of the application
- **EIS Tier**: it contains the data source, that in our case is a database

So, we want to find artifacts that provide the desired system working.

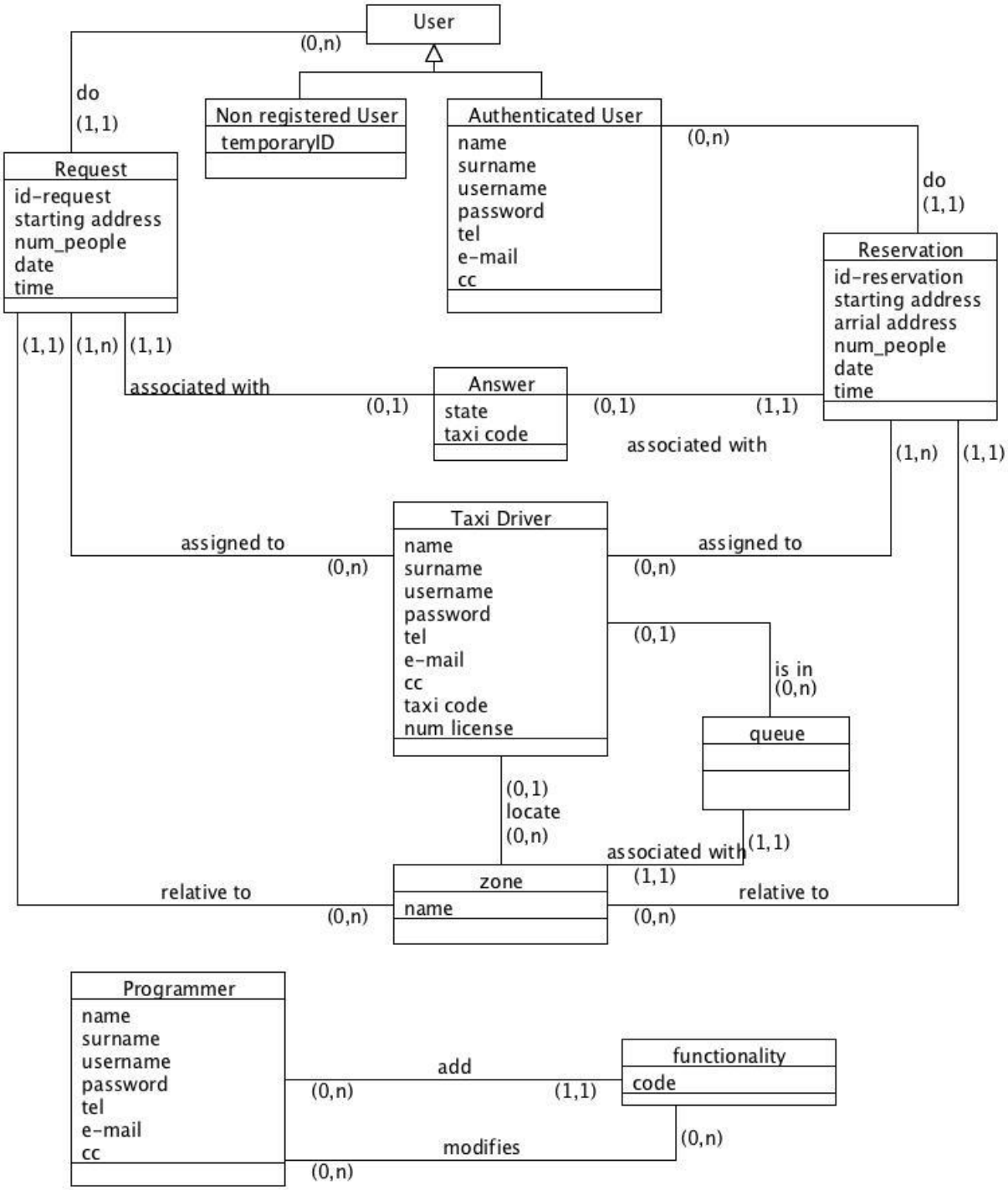


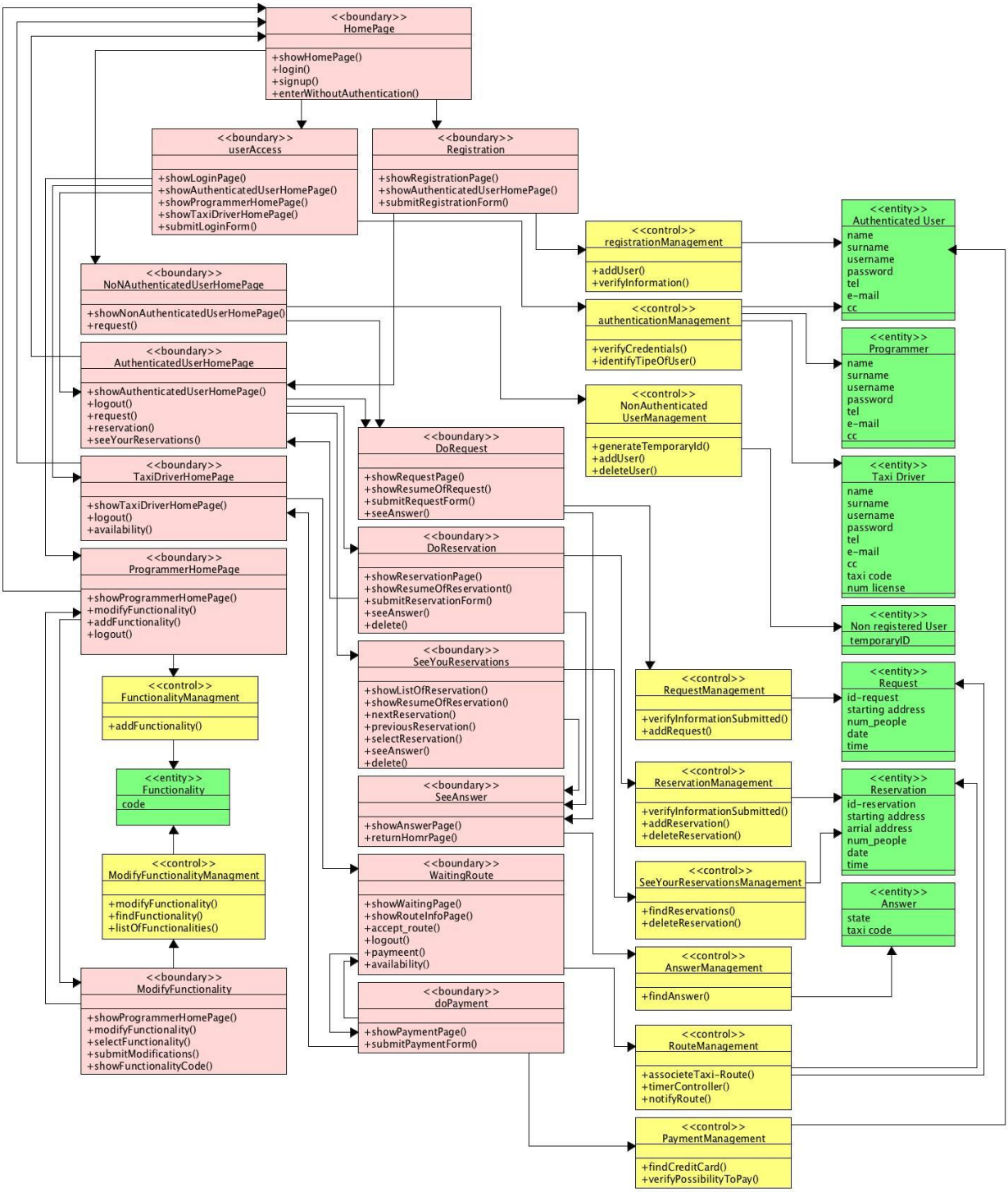
We decide to use the **MCV pattern** (Model-View-Controller) that adapts itself perfectly with the architecture we chosen. Using this pattern there are some advantages:

- The user interfaces are separate by application logic, so we can modify one of them without modify the other one.
- The data are separated by application logic, and this guarantees more reliability.

This distinction between Model, View and Controller can be represented by a BCE diagrams, where there are the BOUNDARIES, that represent the functionalities given to users, the ENTITIES, that are the data, and the CONTROL CLASSES, that are mediators between interfaces and data.

Before illustrate this model we want to give a representation of the data using a Class diagram.


User
(0,n)
do
(1,1)
Non registered User
temporaryID
Authenticated User
name
surname
username
password
tel
e-mail
cc
(0,n)
do
(1,1)
Request
id-request
starting address
num_people
date
time
Reservation
id-reservation
starting address
arrial address
num_people
date
time
(1,1)
(1,n)
(1,1)
associated with
(0,1)
Answer
state
taxi code
(0,1)
(1,1)
associated with
(1,n)
(1,1)
assigned to
(0,n)
Taxi Driver
name
surname
username
password
tel
e-mail
cc
taxi code
num license
assigned to
(0,n)
(0,1)
is in
(0,n)
queue
(0,1)
locate
(0,n)
associated with
(1,1)
(1,1)
relative to
(0,n)
zone
name
relative to
(0,n)
Programmer
name
surname
username
password
tel
e-mail
cc
add
(0,n)
(1,1)
functionality
code
(0,n)
modifies
(0,n)

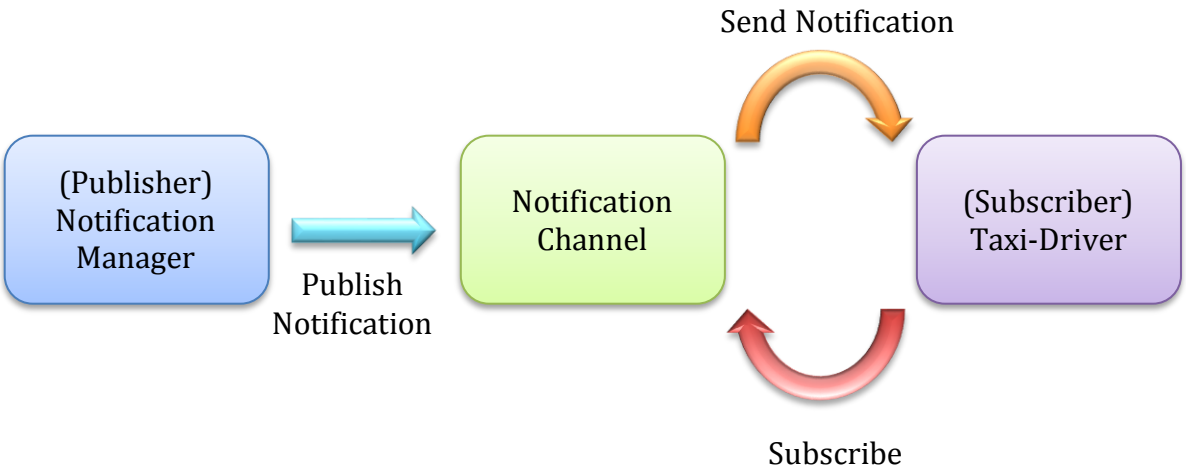

<<boundary>>
HomePage
+showHomePage()
+login()
+signup()
+enterWithoutAuthentication()
<<boundary>>
userAccess
+showLoginPage()
+showAuthenticatedUserHomePage()
+showProgrammerHomePage()
+showTaxiDriverHomePage()
+submitLoginForm()
<<boundary>>
Registration
+showRegistrationPage()
+showAuthenticatedUserHomePage()
+submitRegistrationForm()
<<control>>
registrationManagement
+addUser()
+verifyInformation()
<<entity>>
Authenticated User
name
surname
username
password
tel
e-mail
cc
<<boundary>>
NoNAuthenticatedUserHomePage
+showNonAuthenticatedUserHomePage()
+request()
<<control>>
authenticationManagement
+verifyCredentials()
+identifyTipeOfUser()
<<entity>>
Programmer
name
surname
username
password
tel
e-mail
cc
<<boundary>>
AuthenticatedUserHomePage
+showAuthenticatedUserHomePage()
+logout()
+request()
+reservation()
+seeYourReservations()
<<control>>
NonAuthenticated
UserManagement
+generateTemporaryId()
+addUser()
+deleteUser()
<<boundary>>
DoRequest
+showRequestPage()
+showResumeOfRequest()
+submitRequestForm()
+seeAnswer()
<<entity>>
Taxi Driver
name
surname
username
password
tel
e-mail
cc
taxi code
num license
<<boundary>>
TaxiDriverHomePage
+showTaxiDriverHomePage()
+logout()
+availability()
<<boundary>>
DoReservation
+showReservationPage()
+showResumeOfReservationt()
+submitReservationForm()
+seeAnswer()
+delete()
<<entity>>
Non registered User
temporaryID
<<boundary>>
ProgrammerHomePage
+showProgrammerHomePage()
+modifyFunctionality()
+addFunctionality()
+logout()
<<boundary>>
SeeYouReservations
+showListOfReservation()
+showResumeOfReservation()
+nextReservation()
+previousReservation()
+selectReservation()
+seeAnswer()
+delete()
<<control>>
RequestManagement
+verifyInformationSubmitted()
+addRequest()
<<entity>>
Request
id-request
starting address
num_people
date
time
<<control>>
FunctionalityManagment
+addFunctionality()
<<boundary>>
SeeAnswer
+showAnswerPage()
+returnHomrPage()
<<control>>
ReservationManagement
+verifyInformationSubmitted()
+addReservation()
+deleteReservation()
<<entity>>
Reservation
id-reservation
starting address
arrial address
num_people
date
time
<<entity>>
Functionality
code
<<boundary>>
WaitingRoute
+showWaitingPage()
+showRouteInfoPage()
+accept_route()
+logout()
+payment()
+availability()
<<control>>
SeeYourReservationsManagement
+findReservations()
+deleteReservation()
<<entity>>
Answer
state
taxi code
<<control>>
ModifyFunctionalityManagment
+modifyFunctionality()
+findFunctionality()
+listOfFunctionalities()
<<control>>
AnswerManagement
+findAnswer()
<<boundary>>
ModifyFunctionality
+showProgrammerHomePage()
+modifyFunctionality()
+selectFunctionality()
+submitModifications()
+showFunctionalityCode()
<<boundary>>
doPayment
+showPaymentPage()
+submitPaymentForm()
<<control>>
RouteManagement
+associeteTaxi-Route()
+timerController()
+notifyRoute()
<<control>>
PaymentManagement
+findCreditCard()
+verifyPossibilityToPay()

2.8 Other design decisions

We used a top-down approach to define the architecture of the system. Firstly, we designed the high level structure of the system; secondly we worked gradually down in the low level structure. Using this approach we are focused on building a solid structure rather than the reusability of the system.

We use in some cases the factory pattern, complying with the information hiding principle, reducing, in this way, dependencies on things that could change in future.

Therefore, we use the Publish/Subscribe pattern, which manages the communication between taxi-drivers and notification manager



3. ALGORITHM DESIGN

The following Java code concerns the creation of a new request by a non-authenticated user and the notification forwarding to the taxi driver. User and system are linked through the remote invocation of the method "makeARequest". Finally, the notification manager takes care of the redirection to the first taxi driver in the queue of the zone where the user is situated.

```
1.  /**
2.  * Class that implements the NonAuthenticatedUserInterface.
3.  * Represents the account of a non-authenticated user and his
4.  * related functionalities that can do.
5.  **/
6.  public class NonAuthenticatedUserAccountImpl implements
    NonAuthenticatedUserInterface{
6.       ...
7.       boolean makeARequest(String address, String telephone){
8.            //remote invocation
9.            RemoteMyTaxiService.makeARequest(address,telephone);
10.      }
11. }
12.
13. /**
14. * Class that implements the RemoteMyTaxiService interface.
15. * It contains all the functionalities of the system can be
16. * invoked remotely
17. **/
18. public class myTaxiServiceImpl extends UnicastRemoteObject
    implements RemoteMyTaxiService{
19.           ...
20.
21.      public makeARequest(String address, String telephone){
22.
23.           Request r;
24.           Zone zone;
25.           TaxiDriver td;
26.
27.           //compute the correlated zone of the address
28.           zone = calculateZone(address);
29.           //creates a new request
30.           r = new Request(address, telephone);
31.
32.           try{
33.                //return the first taxidriver in the queue
34.                td = zone.getFirstTdInQueue();
35.           //if there's no taxidriver in the queue, it calls an
36.           exception that takes a taxidriver from an adjacent
37.           queue
38.           } catch (NoTaxiDriversException e) {
39.              e.printStackTrace();
```

```
40.             }
41.             //creates a new notification
42.             n = new Notification(request, td);
43.             //the notification manager provides to properly
44.             forward the notification to the related taxidriver
45.             NotificationManager.addNotification(n);
46.         }
47.   }
```

4. USER INFERFACE DESIGN

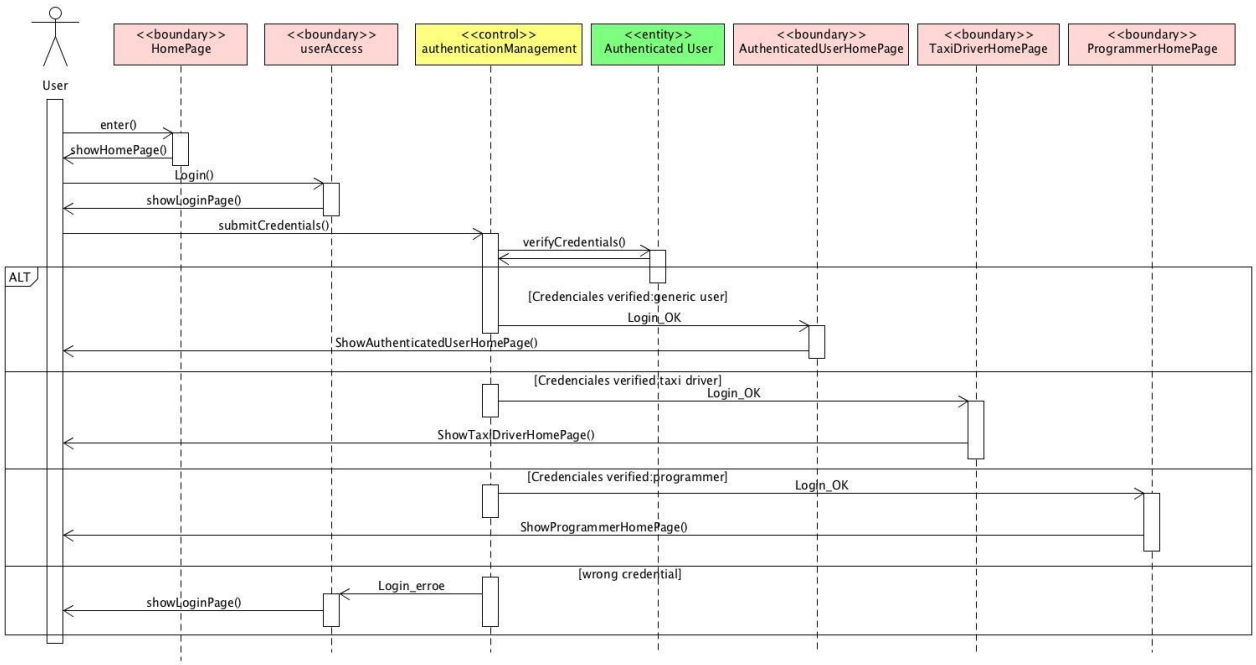
See Chapter 3 of the RASD document.

5. REQUIREMENTS TRACEABILITY

In the RASD document we determine some requirements:

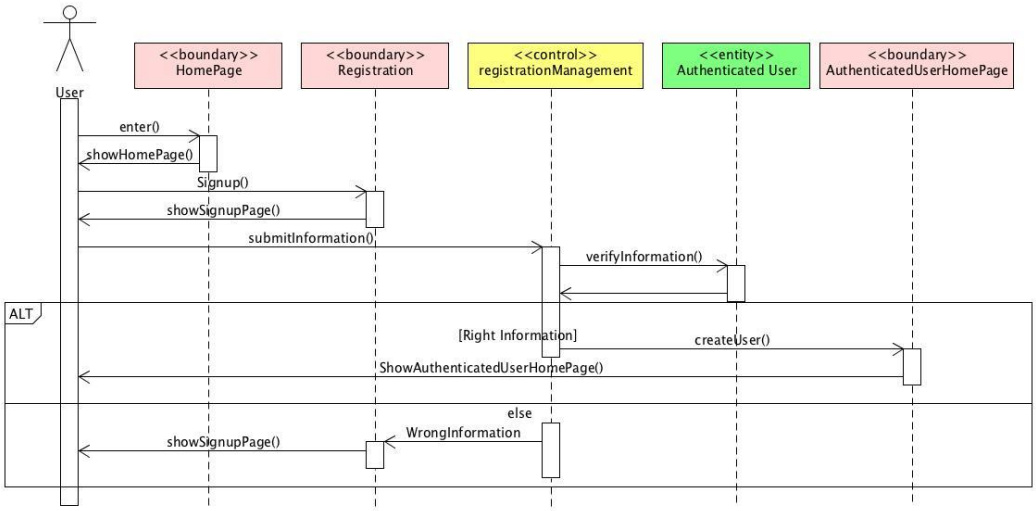
1. Login into the system

When a person enters into the system, the Home Page is shown to him by the View. Now he has three options: login, sign-up, or enter without authentication.
If he chooses to authenticate himself into the system then the View loads a Login Page.
At this point the user can submit their credentials, which are verified by the Controller querying the database. In this way the user is recognized and the View shows him the right Home Page otherwise it shows again the Login Page.



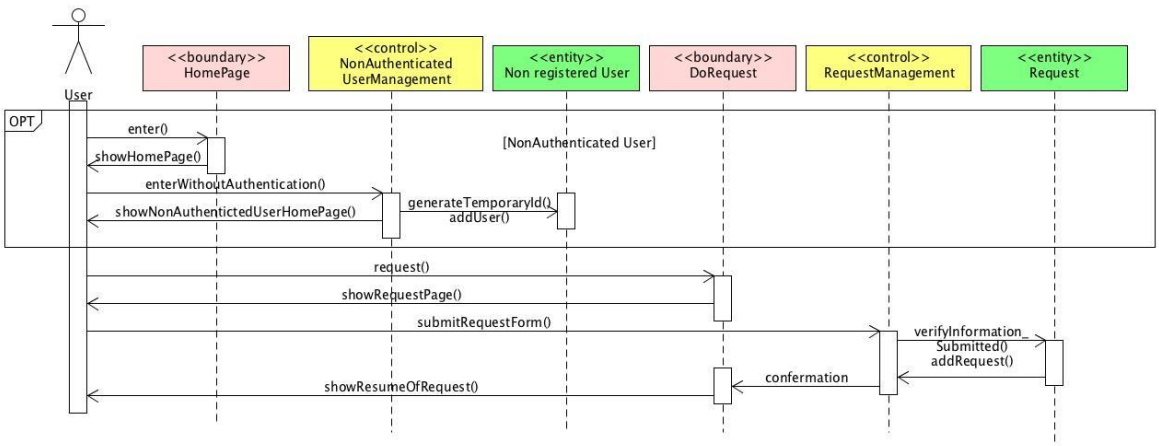
2. Registration of a person in the system

When a person enters into the system and choices “sign-up”, the View loads a Signup Page. Thus, the user can submit his data and choice his credentials. The Controller verifies the information submitted querying the database. If these are correct, the View shows the Authenticated User Homepage, and a new User is added in the Database, otherwise it shows again the Signup Page.



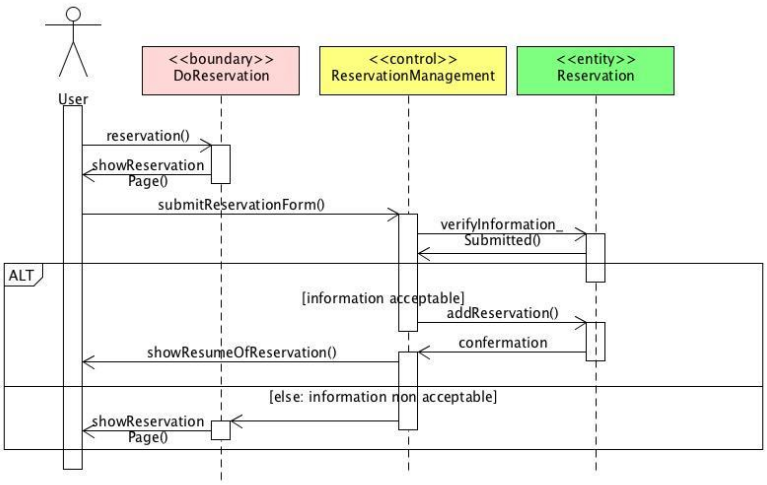
3. Request a taxi service

The request can be done by an authenticated user or non-authenticated user.
When the user enters into the system and chooses “enter without authentication”, the View shows him the Non Authenticated User Homepage, and in the database is added a non-authenticated user with associated only a temporary-Id. He can do only a request.
If the user is authenticated, he can choose to do a request. In both cases, the View shows him the Request Page and the user submits information about the route. The Controller verifies the information submitted and a new request is created in the database.



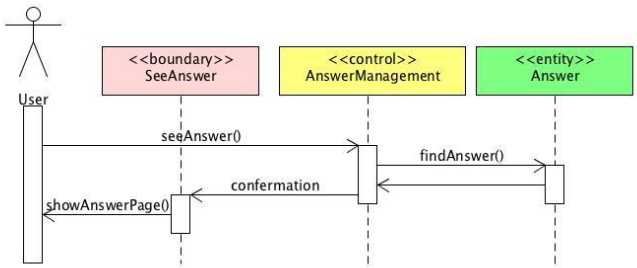
4. Reservation of a taxi

A user, authenticated into the system, can choose to do a reservation. In this case the View shows him the Reservation Page and then the user can submit the Registration form. The Controller verifies the data submitted and, if they are valid, a new reservation is created in the database. As a result, the user visualizes the Resume of reservation. If the submitted Form is not valid, the View shows again to the user the Registration Page.



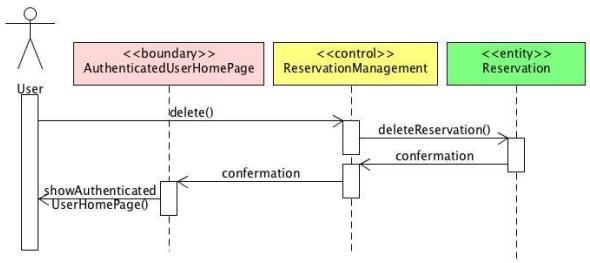
5. Display the confirmation and information about taxi

When a user made a request or a reservation is showed him, by the View, the resume of reservation. The user can choose to see the Answer. In this case, the Controller, asking the information from database, shows him the answer and the state of the route.



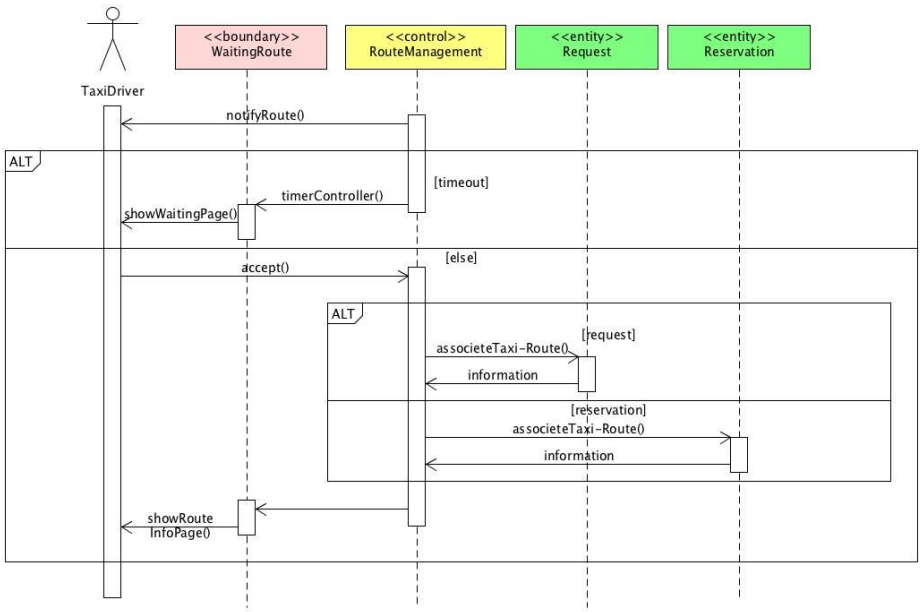
6. Cancellation of a reservation

An Authenticated user, who made a reservation, can see the resume and, then, he can choose to delete the reservation. So, the Controller asks database to delete it.



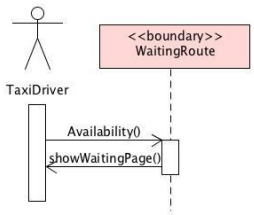
7. View the requests

When a request arrives to a taxi driver, he has a specific period of time to accept this. If he doesn't accept during this period, the View shows him the Waiting page; on the contrary, if he accepts the request, the View shows him the information of the route took by Database with the mediation of Controller. The latter manages the timer.



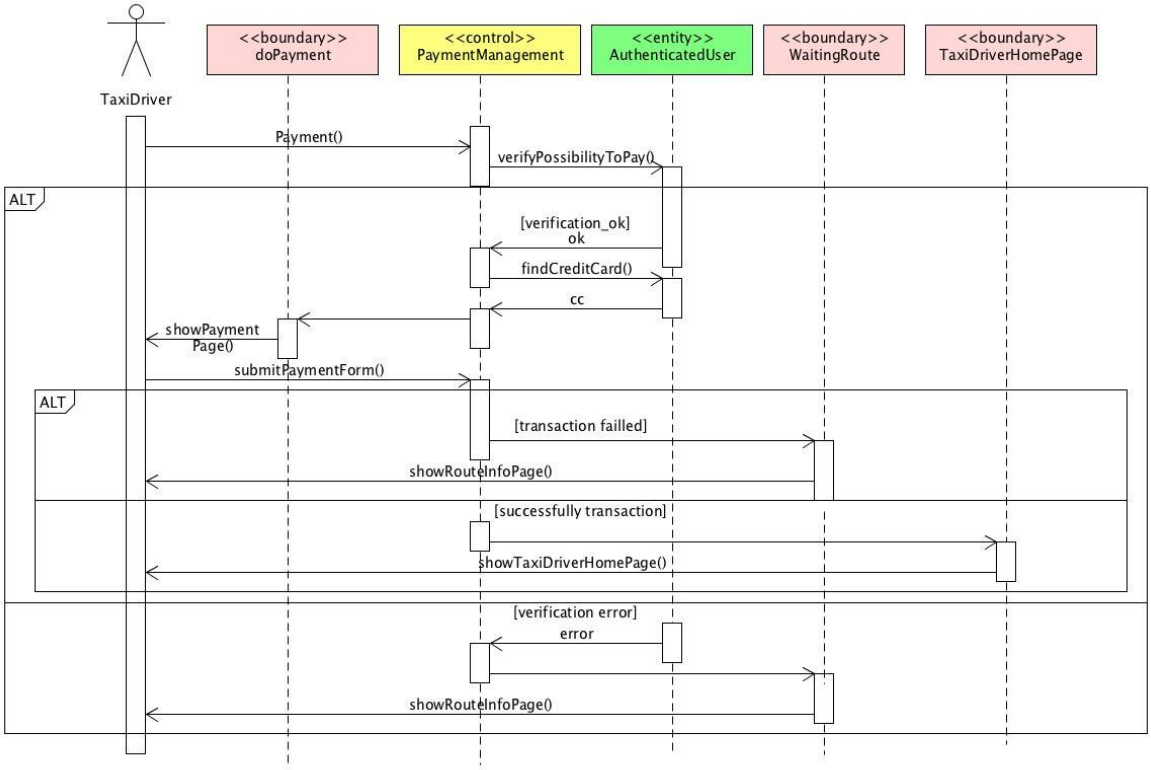
8. Communication of taxi availability

The system, through the login, can recognize the taxi-driver using the information on database, so the Taxi Driver Home Page is shown to him by the View. A taxy driver can communicate his availability, and the View shows him the Waiting Page.



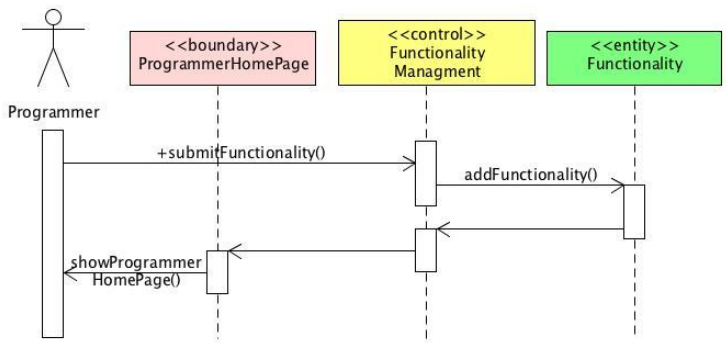
9. Travel payment

If the user is authenticated and he expresses the desire to pay through the system (if he has registered his credit card), the taxi driver can choose “payment”, at first the Controller verify the possibility to do a payment. If it is possible, the Controller keeps the credit card code asking to the Database and the View shoes to the Taxi driver the Payment Page where he can submit the amount. If the transaction ends successfully the View shows to the taxi driver his Homepage, where he can communicate his availability, else, if the transaction doesn’t end well, he returns to Route Info Page.



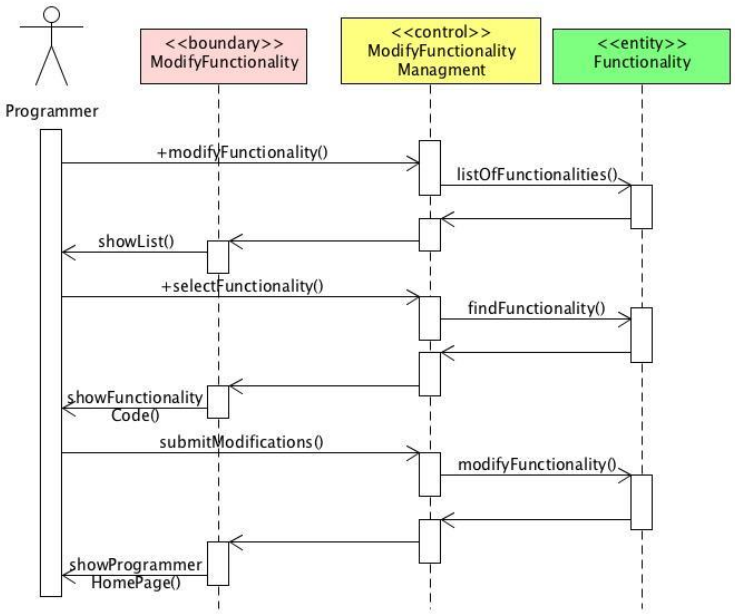
10. Add new functionalities of the system

The programmer can add new functionalities to the system. To do this, he chooses "add functionality" from his Home Page, submitting the code. As a consequence, in the database is added a new functionality that can be use through the system.



11. Modify a functionality

When a programmer wants to modify functionality, he chooses the related option in his Home Page. Afterwards, the View shows him a list of functionalities that he can modify. The controller, querying the Database, creates this list. After selecting the functionality, the Controller asks to the Database the code of this one and this is showed by the View to the programmer. He makes the changes and then submits the modified code, which is updated in the database. Finally, the View shows him the Programmer Homepage.


Programmer
<<boundary>>
ModifyFunctionality
<<control>>
ModifyFunctionality
Managment
<<entity>>
Functionality
+modifyFunctionality()
listOfFunctionalities()
showList()
+selectFunctionality()
findFunctionality()
showFunctionality
Code()
submitModifications()
modifyFunctionality()
showProgrammer
HomePage()