POLITECNICO DI MILANO

Master Degree course in Computer Science Engineering

Design Document

Instructor: Prof. Elisabetta Di Nitto

Authors:

Brunato Andrea	851413
Costantini Lorenzo	852599
Dell'Orto Alessandro	853050

Academic year 2015-2016

TABLE OF CONTENT

1. INTRODUCTION

1.1. Purpose

The purpose of the design document is to help software engineers to choose the architecture of the future system and, once the document is completed, to have a unique resource to which all developers can rely for coding.
We have presented various types of diagrams that, in addition with the ones in the RASD, will be useful for all the people involved in the realization of the software in order to avoid misunderstanding on requirements and design decisions.

1.2. Scope

This document will describe how to the EasyTaxi application will be developed. Component, Deployment, Sequence, ER and Class Diagrams are provided in order to fully describe the track the project has to follow.

C. Acronyms

PK: in the Entity Relationship model it means primary key.

UX: User eXperience

UML: Unified Modeling Language

DBMS: Database Management System

EIS: Enterprise Information System

JPA: Java Persistence API

ER: Entity Relationship

SOA: Service Oriented Architecture

EJB: Enterprise Java Beans

RASD: Requirement Analysis And Specification Document

D. Reference Documents

- Specification Document: Software Engineering 2 Project, AA 2015-2016 Assignments 1 and 2
- Easy Taxi Requirement Analysis And Specification Document (RASD)

E. Document Structure

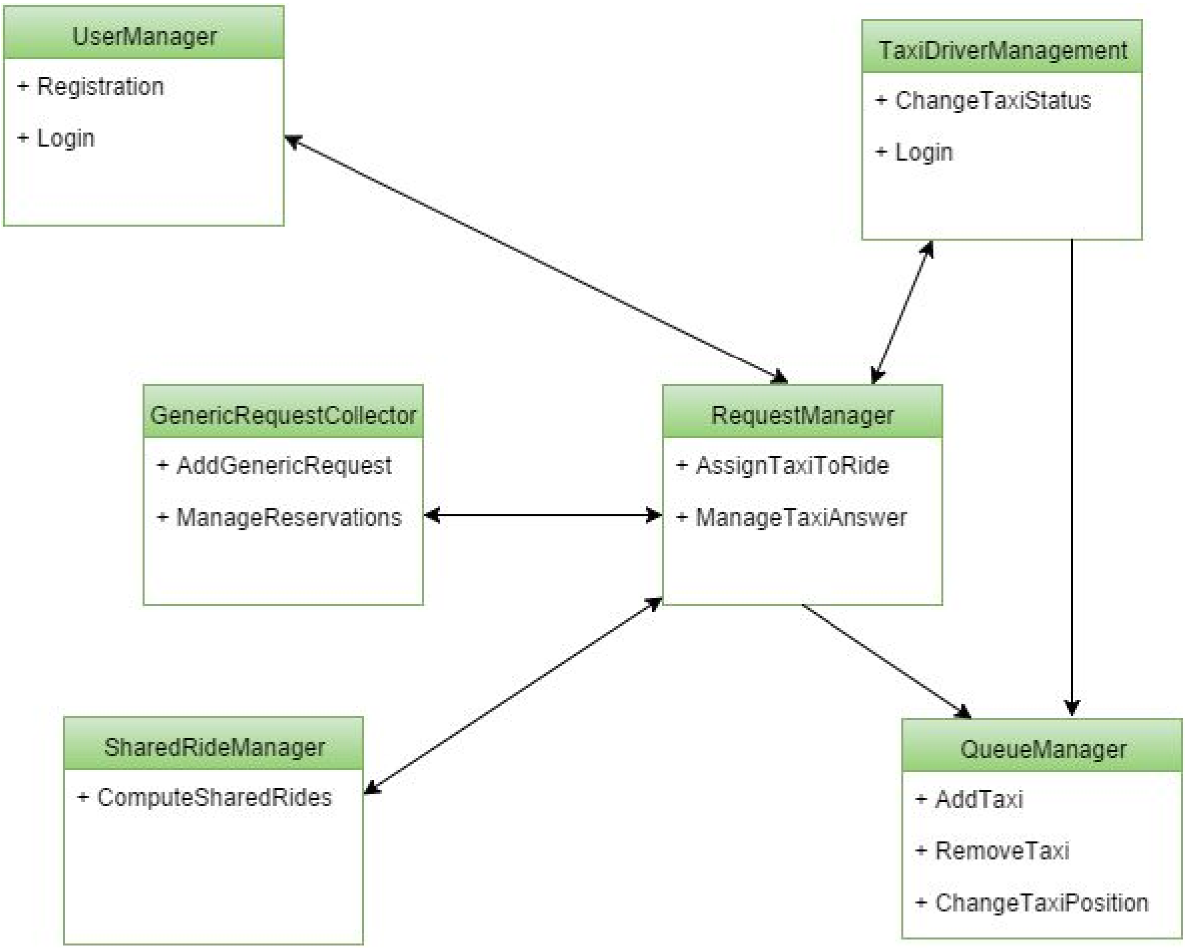
1) **Introduction**: This section contains general informations about our document, the purpose of the realization of the DD and his goals, plus the list of definitions and acronyms used in the following sections of the document.
2) **Architectural Design**: This part describes the most important design and architectural styles chosen: this is the core of our document.
3) **Algorithm Design**: This section describes via pseudocode the main and crucial system's algorithms.
4) **User interface Design**: This part shows how the mockup already presented in the RASD document are reachable between each other through an UX diagram
5) **Requirements traceability**: This part reports how the design decisions match the functional requirements expressed in the RASD document.
6) **References**: This part lists the tools used in making this document.

2. ARCHITECTURAL DESIGN

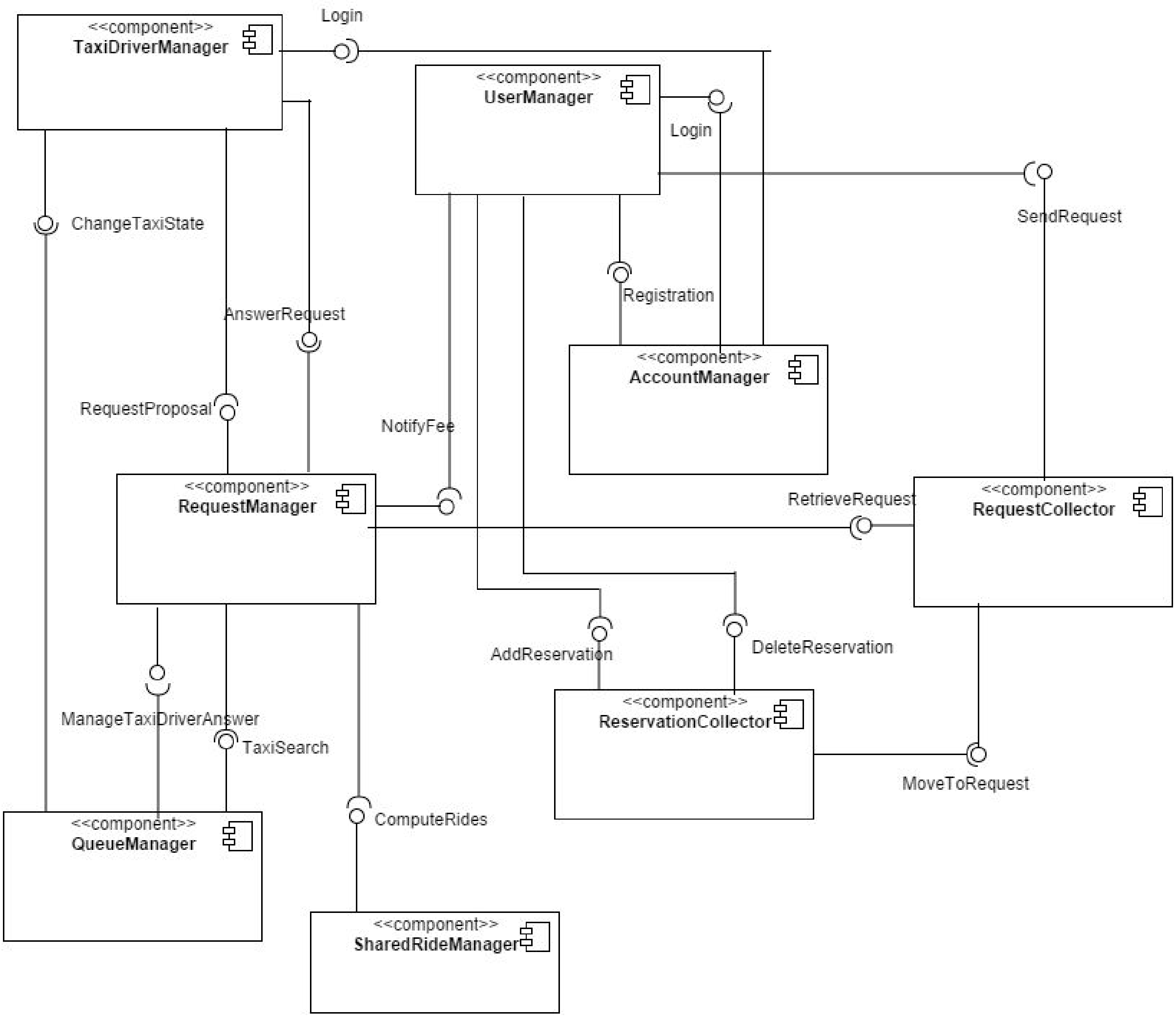
A. Overview

The section B shows the high level components of our system concerning the model of our application. Those components will be more deeply analyzed in the section C through a component diagram, which also specifies the interfaces between them. The meaning of all interfaces can be read in section F.
Section D contains the deployment of our application showing the interaction between tiers and the components in every tier.
Section E contains sequence diagrams that describes the communication between components for the realization of the functionalities of the system. These sequence diagrams are more detailed than the ones already presented in the RASD.
Section G descibes how the architectural styles we have adopted have been implemented and why we have chosen them.
Section H contains other useful diagrams for a better understanding of the future implementation.

B. High level components and their interaction



C. Component view



Components Description:

The following components will be implemented as Stateless Session EJB with the exception of the Queue manager which is a Singleton Session Bean.

Request Manager: This component is the core part of the system and deals with every request sent by users.
Once a new request has been retrieved by the Request Manager from the Request Collector, it checks whether is a Shared or Simple Request and processes it in different ways on top of this distinction interacting with the Queue Manager and the Taxi Driver Manager and with the Shared Ride Manager if the current processed request is shared.

Its dynamic behavior in each case can be easily seen in the Sequence Diagram presented in the Runtime view part of this document. It also invokes SMS and E-Mail services to notify the user about the confirmation of the successful processing of his request, providing also the notification of the fees and the waiting time for the taxi.

QueueManager: This component deals with the taxi drivers queue and continuously keeps updating it whenever a new taxi driver logs into the system or change his status or changes his zone while available.
It checks if the position given by the Taxi Driver Manager (thanks to the GPS) is inside a certain zone and acts in consequence.
It can easily report, when asked by other components, which is the first available taxi.
Because of its coordination role above all taxi interacting with the system this component will be implemented ad a Singleton bean.

ReservationCollector: This component deals with the reservations. His main functionalities are the creation/ deletion of new tuples representing reservations into the database and moving a reservation that is going to start in 10 minutes into the Request Collector queue.

Request Collector: This component deals with the requests. His main functionalities are the creation/ deletion of new tuples representing request into the database and maintaining a request queue that keeps all pending requests.

AccountManager: This component deals with the access methods for the client of the system. Whenever a user is attempting to log into the system this component checks his credentials with the ones saved in the database and let the operation go on if consistency is found. It also deals with the registration mechanism and let the creation of new tuples representing the users in the database.

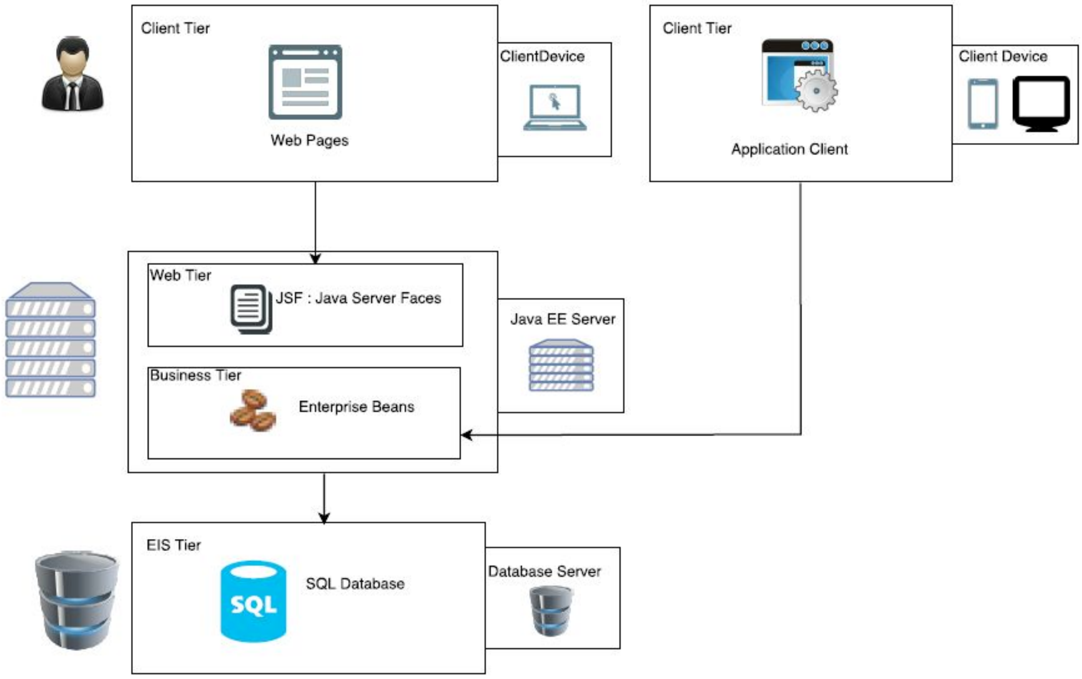
Shared Ride Manager: This component deals with the organization for the shared taxi rides and computes whether two given rides are compatible or not. It invokes Google Maps API methods in order to find the optimal route path to the one given by the Request Manager.

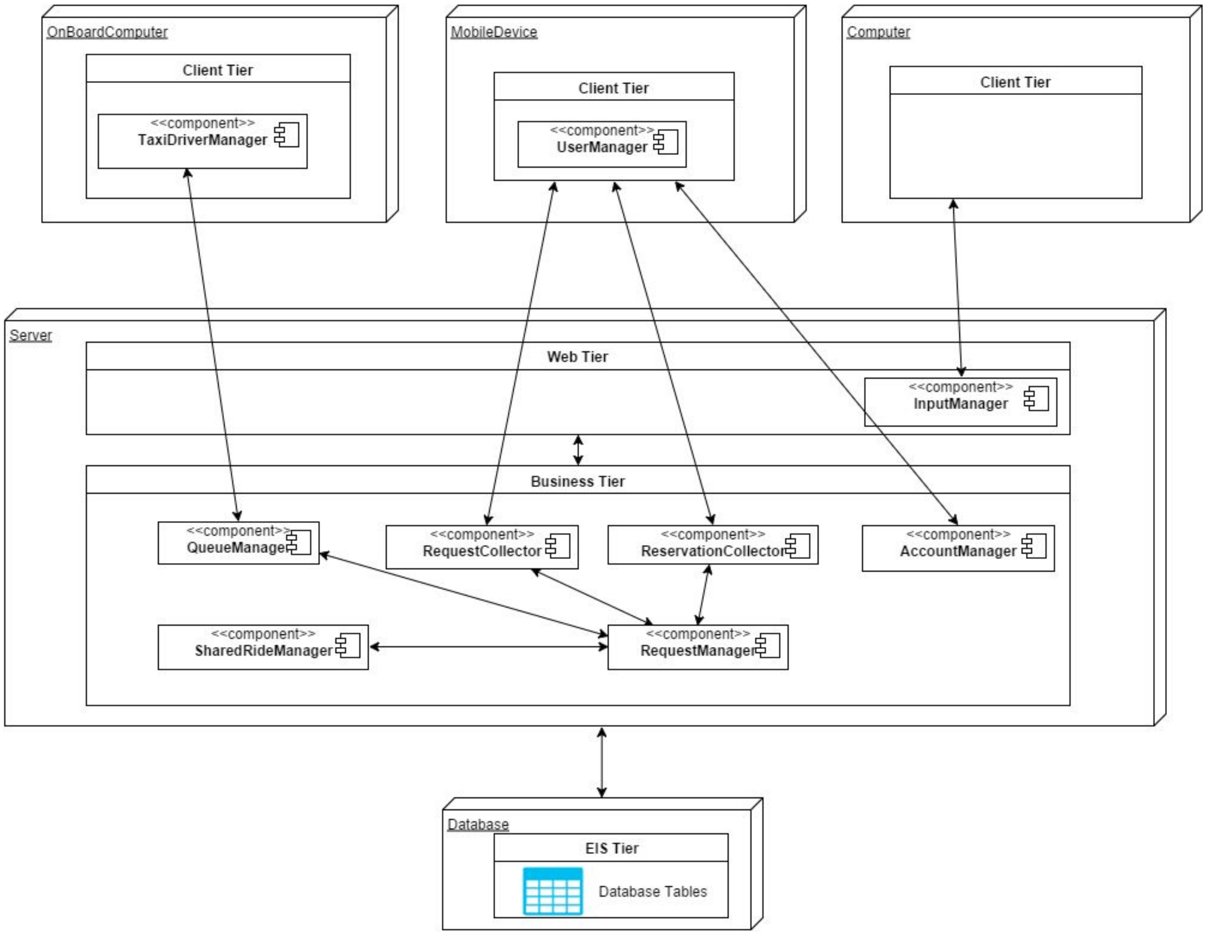
User Manager: This component is completely devoted to deal with the user interaction with the system. It offers a GUI in which the user can select all the operations implemented by the system,retrieves the user position through GPS and can send data through the internet.

Taxi Driver Manager: This component is completely devoted to deal with the user interaction with the system. It offers a GUI in which the taxi driver can select all the operations implemented by the system, retrieves the taxi driver position through GPS and can send data through the internet.

In the deployment view it appears also an **Input Manager** component that is a set of all utility classes designed to accept input forms from the web page layer. These classes belongs to the above descripted components that interact with the User Manager.

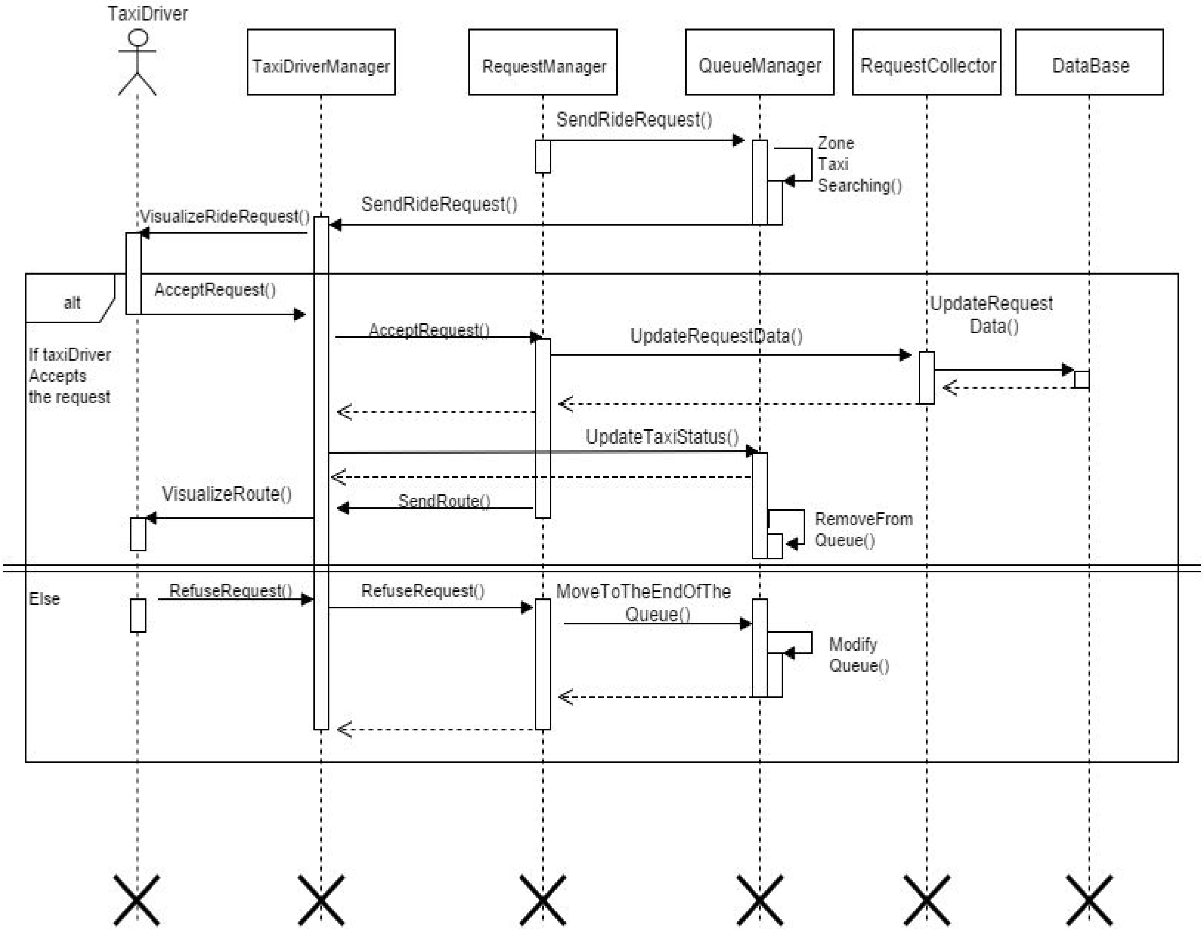
D. Deployment view



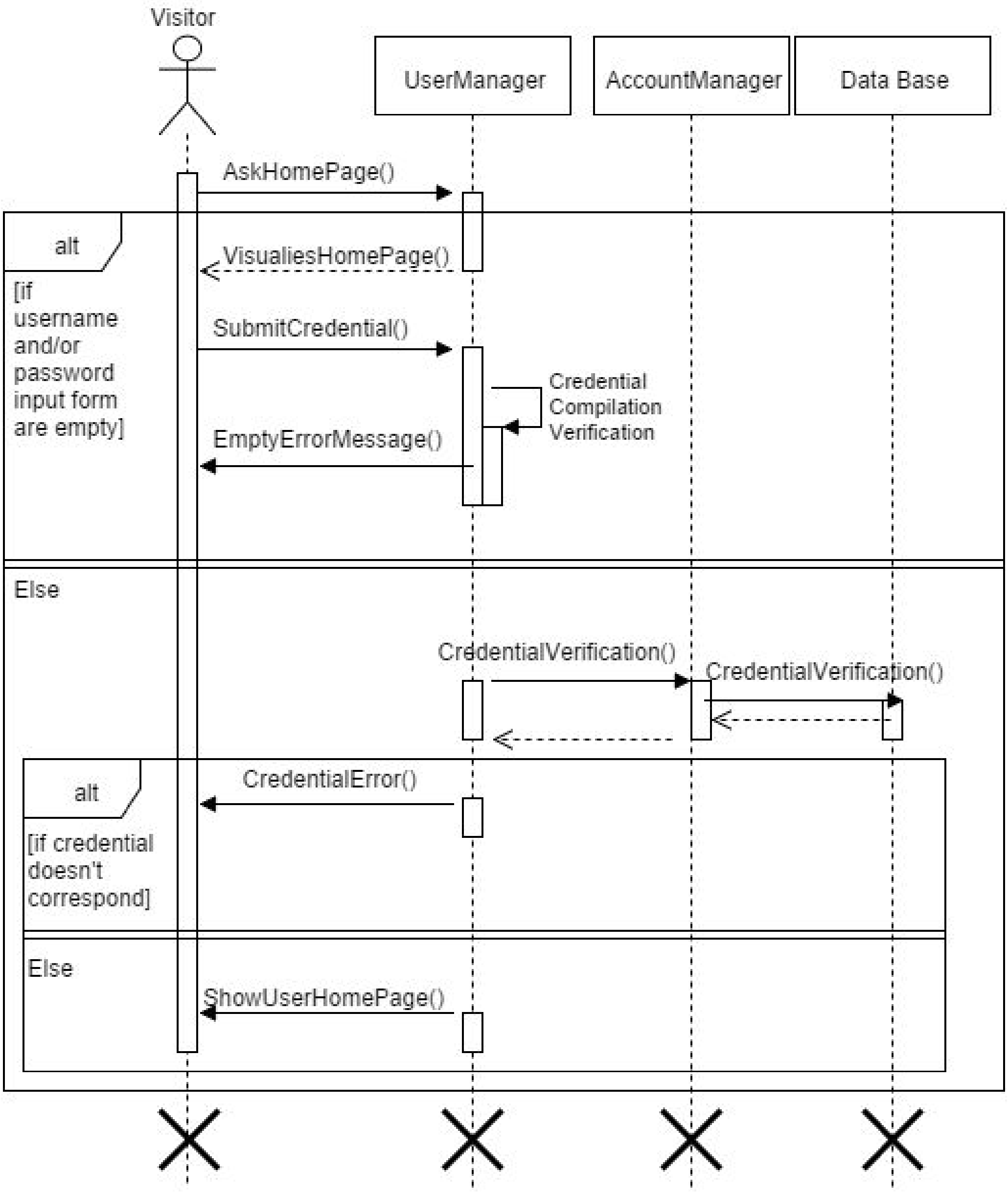

OnBoardComputer
Client Tier
<<component>>
TaxiDriverManager
MobileDevice
Client Tier
<<component>>
UserManager
Computer
Client Tier
Server
Web Tier
<<component>>
InputManager
Business Tier
<<component>>
QueueManager
<<component>>
RequestCollector
<<component>>
ReservationCollector
<<component>>
AccountManager
<<component>>
SharedRideManager
<<component>>
RequestManager
Database
EIS Tier
Database Tables

E. Runtime view

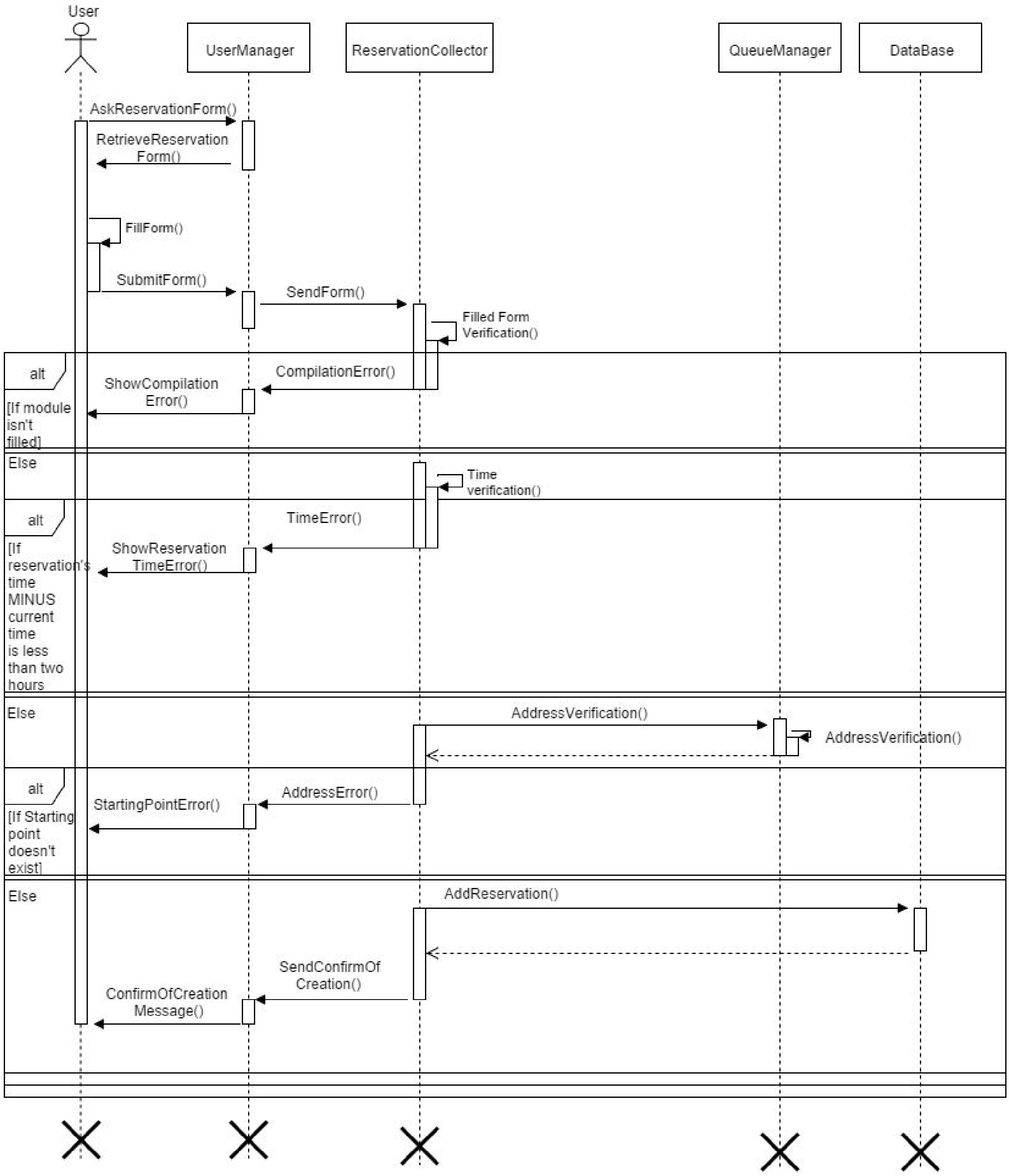
1.Answer to Request



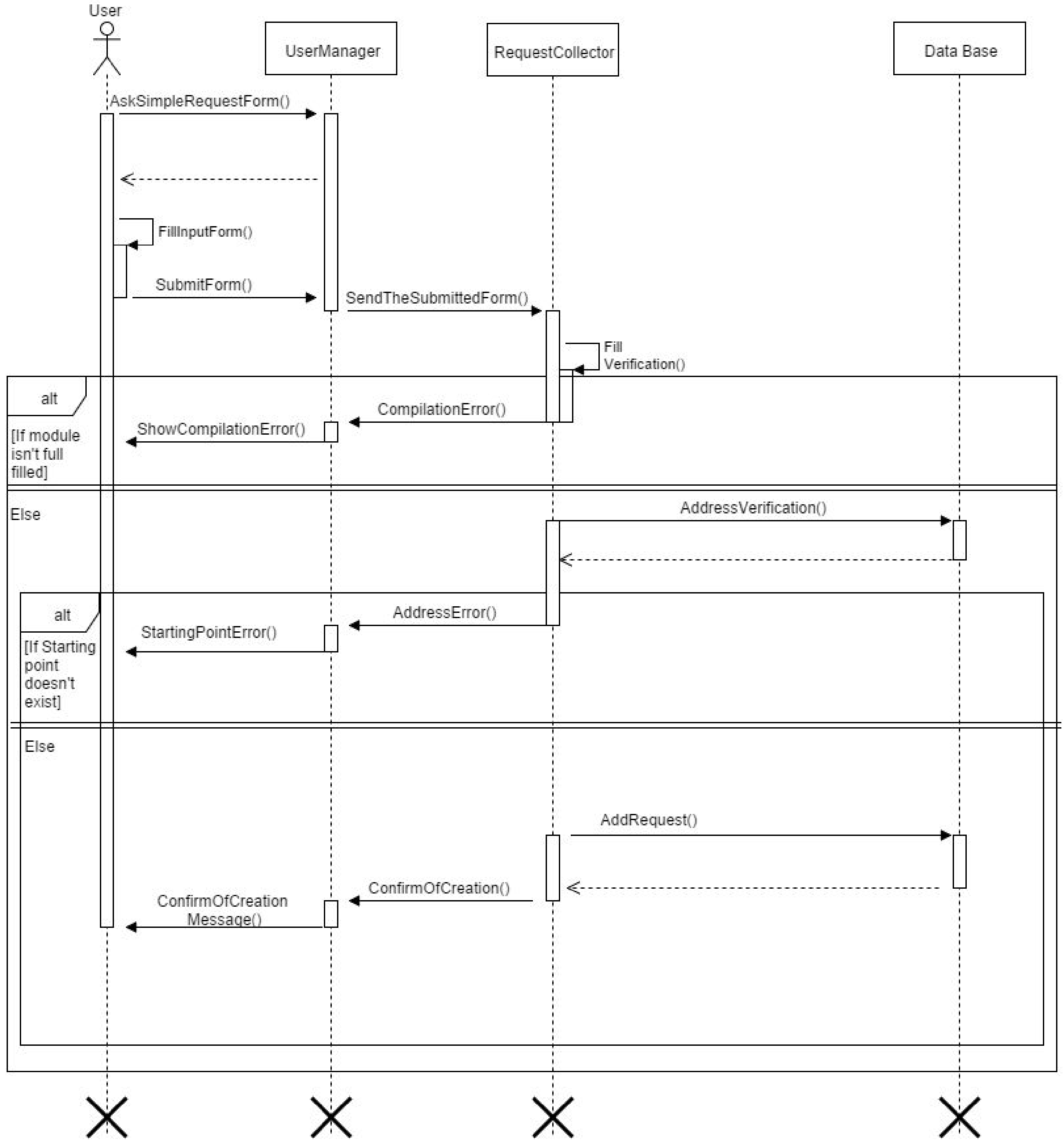
2.Client Login



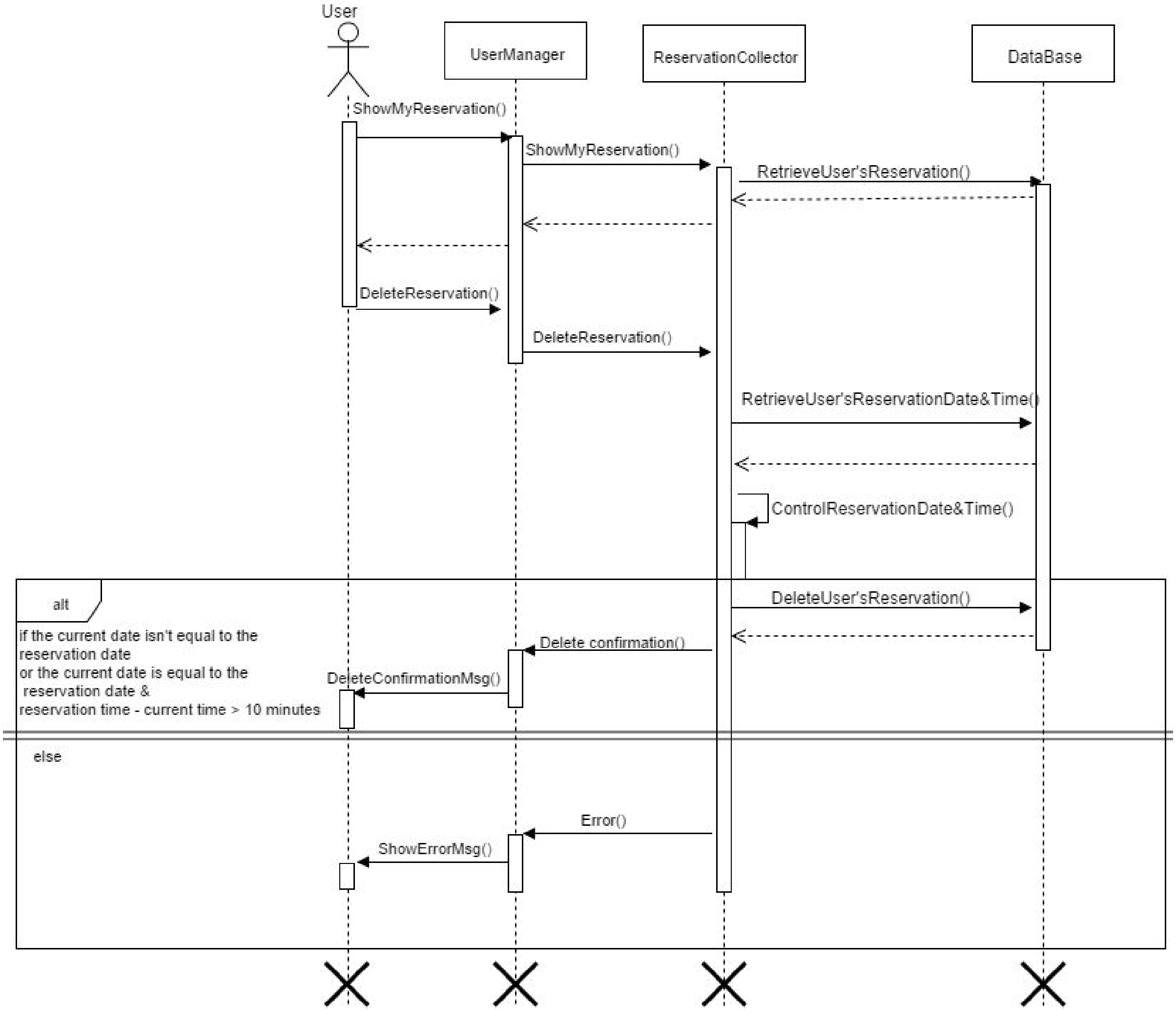
3.Create Reservation



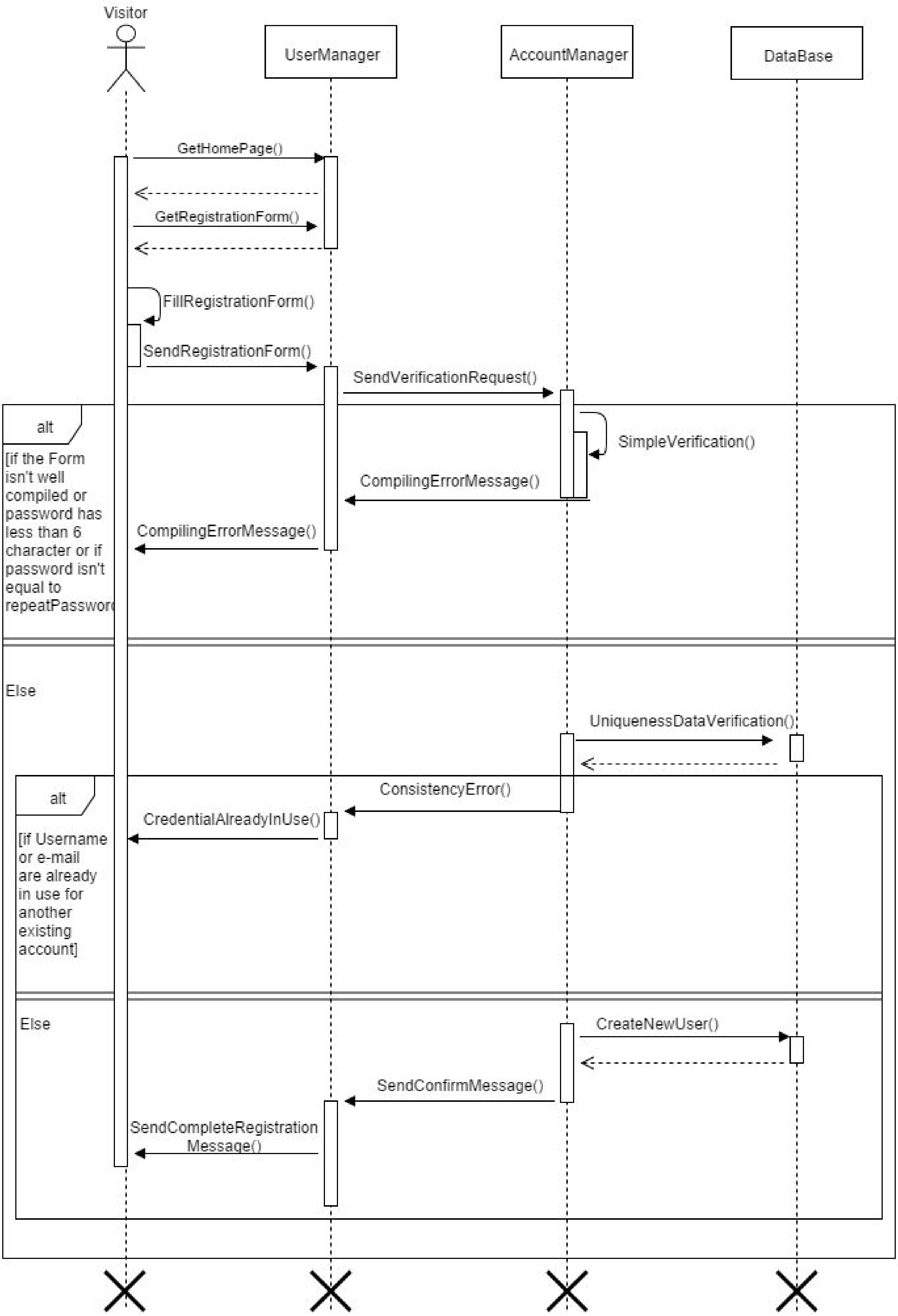
4.Create Simple Request



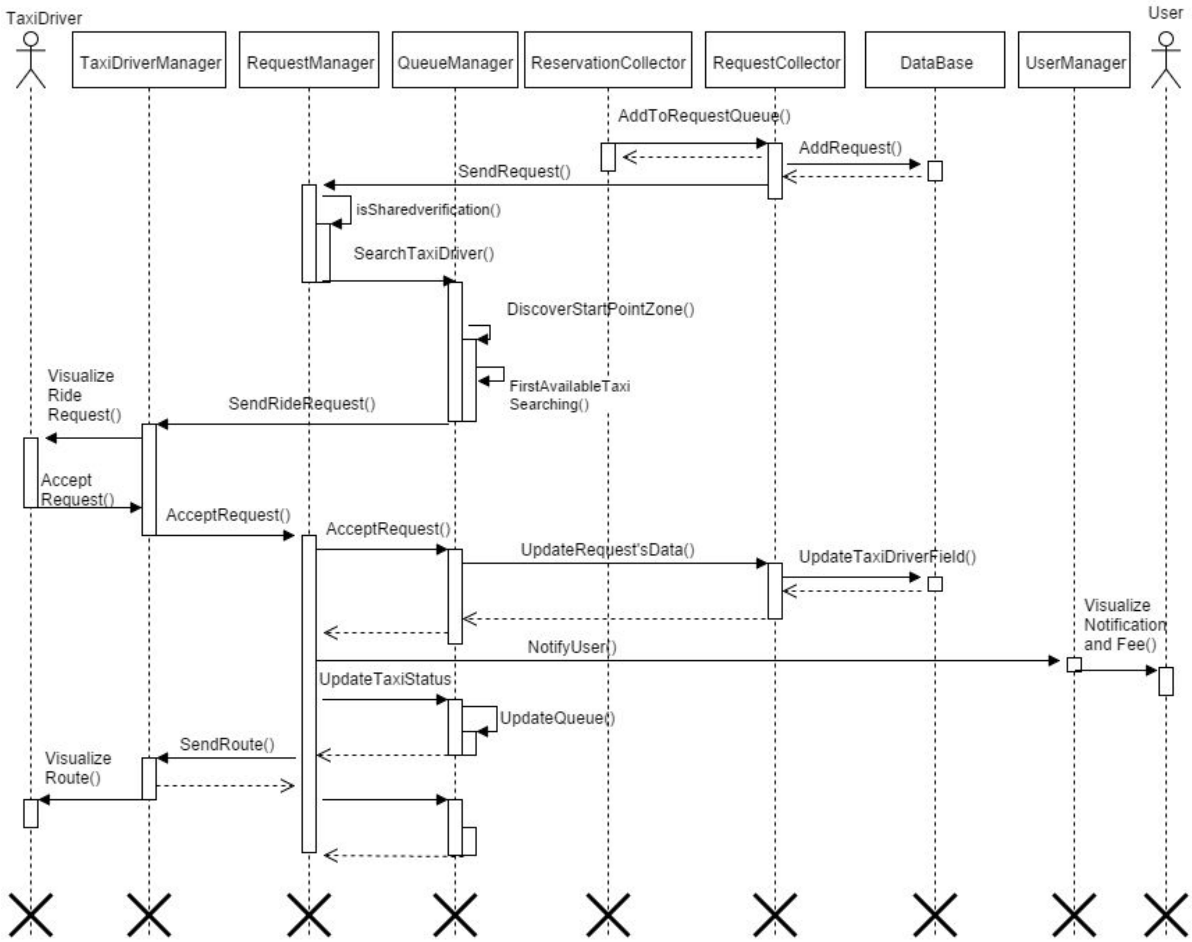
5.Delete Reservation



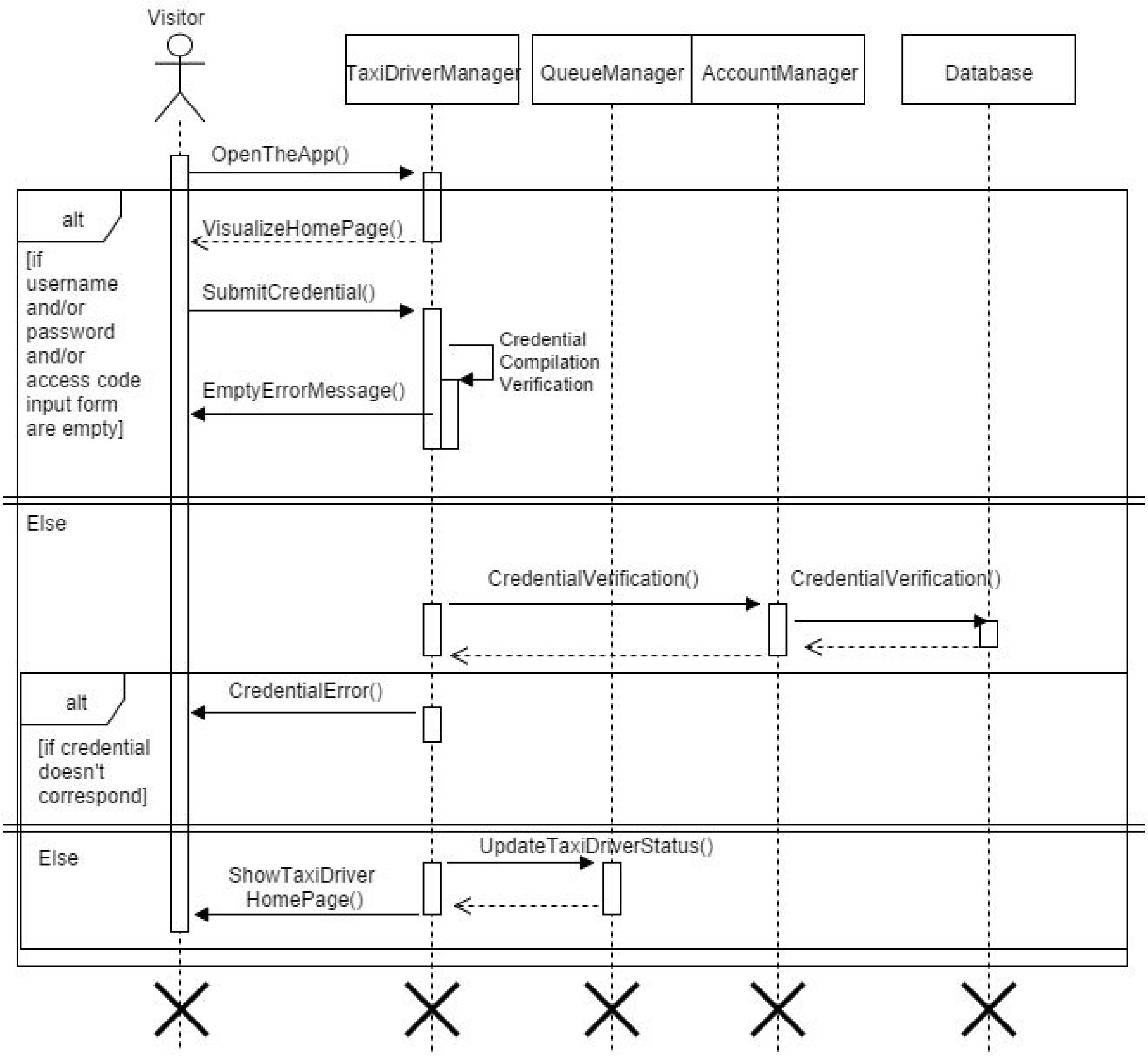
6.Registration



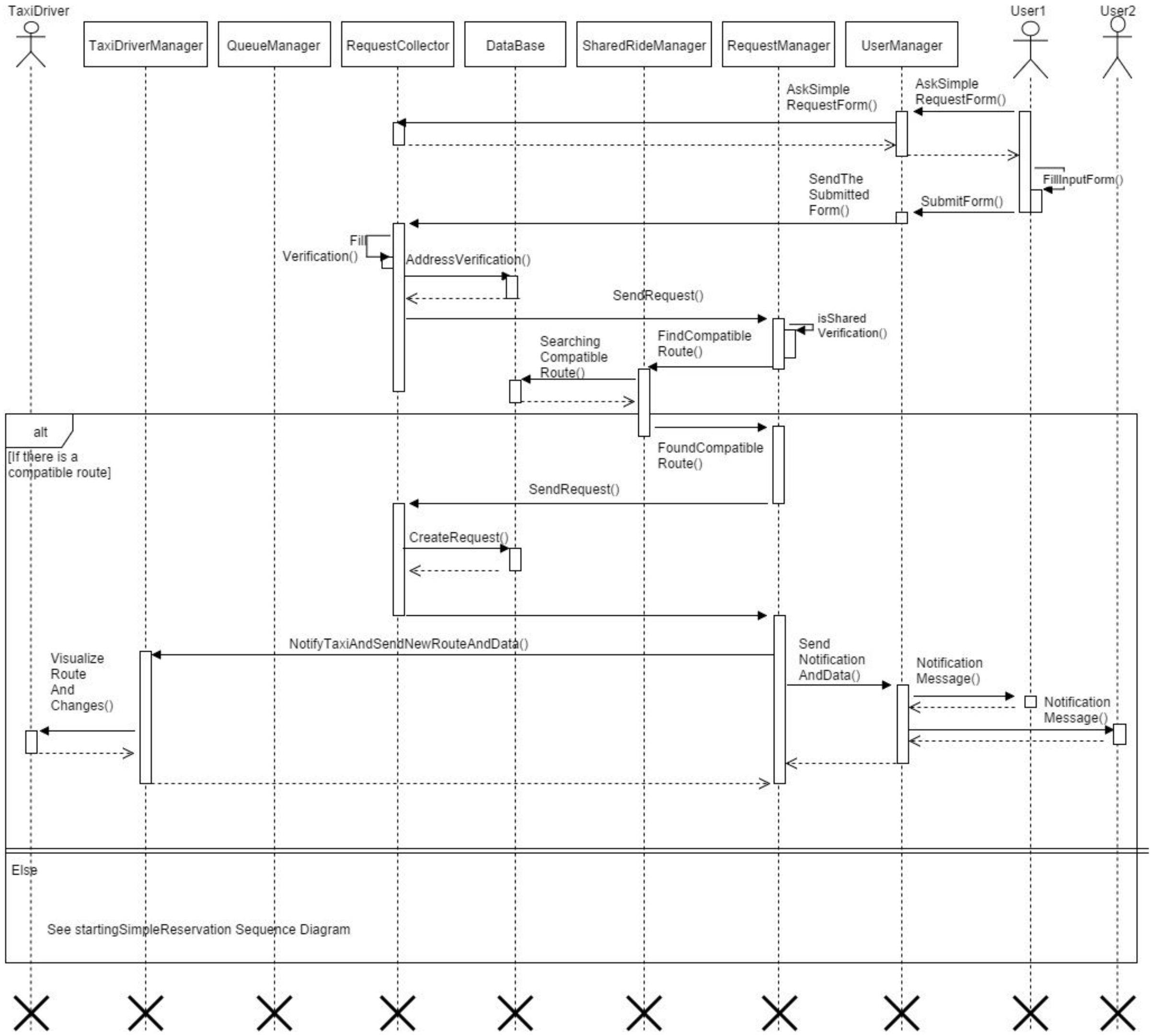
7.Starting Simple Reservation



8.Taxi Driver Login



9.Creation of Shared Request



F. Component interfaces

Each components expones the following interfaces:

- **TaxiDriverManager**:
  - ***ChangeTaxiState***: This interface is provided to the Queue Manager component. It lets the taxi driver to change his current status from Busy to Available and conversely. The new status will affect the taxi driver position in the queue present in a class of the Queue Manager component: if taxi driver set himself to avalaible, he is added to the correspondent queue, while if he set himself to busy he is removed from that queue.
  - ***Login***: This interface is provided to Account Manager component: it allows the taxi driver to log into the system. The Account Manager will be responsible of checking the correspondence between username, password and access code.
  - ***AnswerRequest***: This interface is provided to the RequestManager. It implements everything to let the taxi driver answer to a request (process the acceptance of the answer input of taxi driver, control of the passing of 15 seconds). The Request Manager will use the informations of the driver's answer to interact with the Queue Manager (as we can see in the next point) and with the Request Collector.
- **RequestManager**:
  - ***ManageTaxiDriverAnswer***: This interface is provided to the Queue Manager. Based from the answer of a taxi driver received by the Request Manager, the Queue Manager will delete the taxi from the queue or it will put the taxi still avalaible at the end of the same queue (the former is the case of request accepted, the latter is about request declined or not answered in time).
  - ***RequestProposal***: This interface is provided to the Taxi Driver Manager component in order to notify the taxi driver about a new generic request. The Request is chosen by the Request Manager from a queue of requests (not started yet without a taxi driver assigned to them) in the Request Collector component.
  - ***NotifyFee***: This interface is provided to the User Manager component: it allows the user to be informed about his ride fees, both after he has done a simple request and after the end of the fee modification algorithm called after the add of a passenger in a shared request. The user can be contacted by SMS or E-Mail service, depending on the choice of the client during the request compilation.
- **ReservationCollector**:
  - ***AddReservation***: This interface is provided to the User Manager and allows to archive reservations and save them into the database.
  - ***DeleteReservation***: This interface is provided to the UserManager and allow the user to delete his own reservations. The reservations are deleted from the database by the Reservation Collector.
- **AccountManager**:

- ○ ***Registration***: This interface is provided to the User Manager component and allows the client to register himself into the system. His user data will be collected into the database by the Account Manager.
- **UserManager**:
  - ○ ***Login***: This interface is provided to the Account Manager component. It allows a client to log into the system. The Account Manager will be responsible of checking the corrispondence between username and password.
- **QueueManager:**
  - ○ ***TaxiSearch***: This interface is provided to the Request Manager Component and allows to make research into the queue of available taxis of a certain zone.
- **RequestCollector**:
  - ○ ***SendRequest***: This interface is provided to the User Manager. It allows the client to make a simple request, which is istantly added both into a queue of request in a class of the Request Collector and in the database, with the taxi driver of the new request tuple initially set to null.
  - ○ ***MoveToRequest***: This interface is provided to the Reservation Collector in order to move the reservations from their database table to the queue of requests in the Request Collector and in the database in the request table with the taxi driver field initially set to null.
  - ○ ***Retrieve Request***: This interface is provided to the Request Manager. The Request Manager can retrieve a new request waiting for a taxi driver from the queue of the Request Collector, after the request retrieved before that one has just been added his taxi driver field in the database (it was previously added in the DBMS with that field set to null)
- **SharedRideManager**
  - ○ ***ComputeRides***: This interface is provided to the Request Manager. When the Request Manager takes from the Request Collector a new request, if it is shared, the Request Manager use the method of the shared ride manager for determining the best compatible shared route.

G. Selected architectural styles and patterns

We've decided to apply the **Client-Server architecture** divided into the following four tier:

1) **Client Tier**
2) **Web Tier**
3) **Business Tier**
4) **EIS Tier**

This is the classic way to implement a system using java EE since this framework supports several functionalities that makes this architectural choice simpler to handle:

- EJB to handle the communication between business tier components
- JSF for the presentation of the pages in the web tier to the client tier
- JPA for the communication between EIS tier and the entity beans of the business tier.

By dividing the software in these subtiers we can focus on the business logic development.

Moreover the Client-Server paradigm grants a strong control on user actions: it will be difficult to access protected data and perform not allowed actions thanks to the fact that the core application part isn't running the user device.
The only way in which the user can attempt to do illegal actions is by sending requests to the server side and it will be unlikely to succeed if the control classes (check of correspondence of username and password for example) will be implemented with a strong care about security.

As it's usually done with a client-server architecture with four tiers, we adopt the **MVC pattern**, implemented in the following way: Session Beans acts as controllers, database entities as models and JSF manages the view and the pages presentation to the web browser. User manager and Taxi driver manager components handle the graphical presentation on the mobile application.

The Model-View-Controller paradigm is useful because it increases modularity and makes a strong distinction between the application model classes and controller classes, so that every software part will have his specific role and will be easier to change or correct some functionalities.

Service Oriented Architecture style is implemented in many aspects of our software. EJBs represent services offered to other components accessible through interfaces. Moreover the software exposes programmatic interfaces and uses external services as:

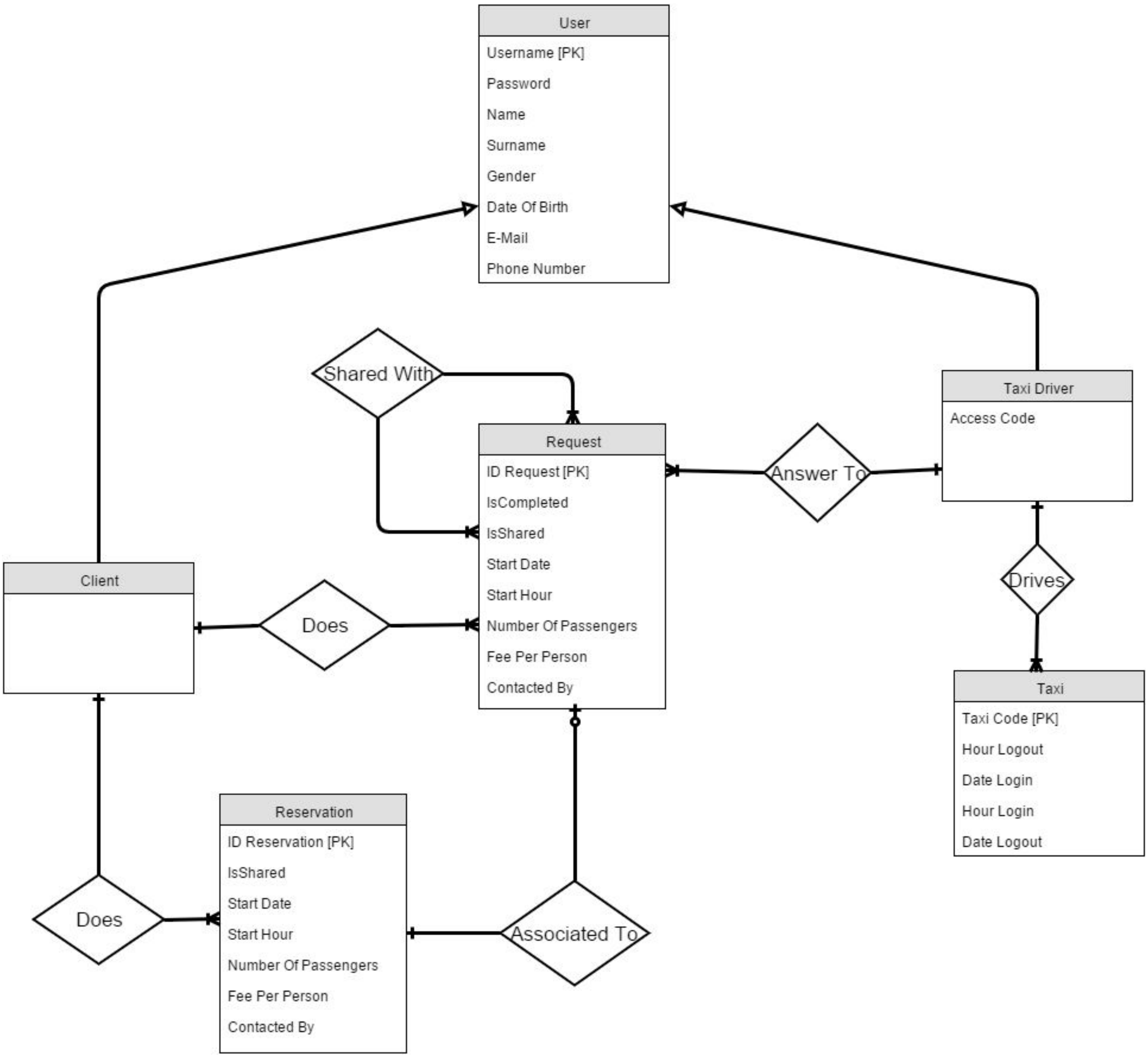
- The Google Maps API for the visualization of the route on the onboard computer of the taxi driver and the calculation of shared ride routes.

- The GPS service for the localization of clients and taxi drivers in real time.
- The SMS and E-mail service to contact clients after the submission of a request.

We have chosen to design the system architecture with **components and connectors** in order to increase maintenability and maximize the decoupling between classes. In this document components are present in most of our diagrams and this design choice is helpful because it adds a new level of abstraction above the concept of classes.

Since EasyTaxi will be implemented in Java, which is a OO language, we have used the **distributed object** style. Thanks to this design decision it will be easier in the future to allocate the various object in different physical tiers in case of necessity.

H. Other design decisions
Conceptual Design of DB:



Logic Design of DB:

CLIENT: (**Username**, Password, Name, Surname, Gender, Date Of Birth, E-Mail, Phone Number)

TAXI DRIVER: (**Username**, Password, Access Code, Name, Surname, Gender, Date Of Birth, E-Mail, Phone Number)

REQUEST: (**ID Request**, IsShared, Shared With*****, StartPoint, Start Date, Start Hour, EndPoint*****, Number Of Passenger, Fee Per Person*, Contacted By, Client*, Taxi Driver*, IsCompleted, IDReservation)

RESERVATION: (**ID Reservation**, IsShared, StartPoint, Start Date, Start Hour, EndPoint, Number Of Passenger, Fee Per Person, Contacted By, Client)

TAXI: (**Taxi Code**, **Login Date**, **Login Hour**, Logout Date, Logout Hour, Driver)

Notes About The Diagrams:

- To represent the zone it's sufficient to keep an enumeration which isn't saved in the database but contained in a class of the Queue Manager Component. A zone is characterized by four coordinates, expressed in latitude and longitude, which are the geographical boundary defining it. To compute whether a certain coordinate given by GPS representing the actual taxi driver position is inside a zone, it's sufficient to check that the latitute and longitude belong both to the interval described by the four coordinates. If there doesn't exists such an interval, it means that the avalaible driver is outside the city and therefore his state will be set to busy by the Queue Manager.
- The Contacted By field can contain only the strings 'SMS' or 'E-Mail'.
- The informations about current availability of taxis and the current zone where they are driving are not contained in the DB as they would frequently updated and they don't need to be stored for future analysis. Those informations are contained in data structures accessed by the Queue Manager.
- The Client in the Request table is optional if the request is sent by a visitor. Since a reservation can't be done by a visitor, this field is not nullable in the Reservation table
- The simple request without sharing option can be submitted by a user without the indication of the ride destination, so the DestinationPoint and the Fee Per Person could be empty in the Request table.
- Taxi Driver field is optional because when a request is added to the database right after it is added to the request collector queue, this field is initially null and it will be updated after a taxi driver is found.
- IsCompleted field is a true boolean if it indicates rides happened in the past. This field is not present in the Reservation table, but it's easy to retrieve past reservations by a join with the Request table.
- Both the informations about past taxi logins and about past rides could be deleted periodically or saved into another place for future statistical analyses purposes, because the presence of many tuples could slow down the computation of some functions of the software based on the data in the Request Table and into the Taxi table.
- The field SharedWith contains another ID Request, that is the one who shares some part of the route with the tuple considered. By retrieving the startPoint and the EndPoint of the individual requests involved, it's possible to have informations about the whole shared route.

3. ALGORITHM DESIGN

FEE CALCULATION

This algorithm compute the fee per person of a request. The request can be simple or shared and the algorithm make a distinction whether the ride is actually shared with someone or a compatible route hasn't been found.
The algorithm shows how the fee is computed and how this information is updated in the database without considering the notification to the user.

```
BEGIN
IdRequest = input;                  //input of the algorithm

SID    = null;                      //ID of the request that starts the ride, initially without a client who shares the route

CSID   = null;                      //ID of the request whose start point field correspond to the beginning point of the
                                    //shared route, i.e the part of the route where both the clients involved with their
                                    //respective passengers are all together in the car

EID    = null;                      //ID of the request which ends after the ending of the shared route

CEID   = null;                      //ID of the request whose end point field correspond to the ending point
                                    //of the shared route

STARTP = null;                      //starting point of the ride for the case of a simple request

ENDP   = null;                      //ending point of the ride for the case of a simple request

NOP1   = null;                      //number of passenger of the request whose ID is saved into SID variable

NOP2   = null;                      //number of passenger of the request whose ID is saved into CSID variable.
                                    //The sum of NOP1 and NOP2 indicates the number of passengers in the car
                                    //during the shared route.
km1    = null;
km2    = null;
cost   = variable;                  //fixed cost for chilometer, it is a constant value

fixedCost[0...24]= variable;        //base cost of the ride, only based on the hour of start of the ride,fixed for
                                    //every hour

IF type! = "shared"  OR (type == "shared" AND cond == null)   THEN  //fee calculation for simple request and
        NOP1 = IdRequest.NumOfPassenger;                             //for shared request that hasn't already
        STARTP = IdRequest.startPoint;                                     //found a sharer
        ENDP = IdRequest.endPoint;
        km1 = chilometersCalculation(STARTP,ENDP);
        IdRequest.fee = (km1 * cost + fixedCost(ActualTime)) / NOP1;
ELSE
        IF IdRequest.time < IdRequest.cond.time             THEN    //identification of the user that does the
                SID = IdRequest;                                     //first part of the ride alone
                CSID = IdRequest.SharedWith;
        ELSE
                CSID = IdRequest;
                SID = IdRequest.SharedWith  ;            END-IF

        km1 = chilometersCalculation(CSID.startPoint , IdRequest.endPoint);
        km2 = chilometersCalculation(CSID.SharedWith.startPoint , IdRequest.SharedWith.endPoint);
```

```
        IF km1 > km2        THEN                                  //identification of the ID of the requests that
                EID = IdRequest;                                  //terminates at the end of the shared route
                CEID = IdRequest.SharedWith;                      //and of  the one who does the last part of
        ELSE                                                      //the route alone
                CEID = IdRequest;
                EID = IdRequest.SharedWith;  END-IF

        NOP1 = SID.numberOfPerson
        NOP2 = CSID.numberOfPerson

                                                  //calculation of the fee per person of every request of the ride

        CSID.fee = chilometersCalculation(CSID.startPoint,CEID.endPoint) * cost/ (NOP1 + NOP2) + fixedCost(ActualTime);
        SID.fee = chilometersCalculation(SID.startPoint,CSID.startPoint) * cost/ NOP1) + CSID.fee;

        IF SID == CEID      THEN
                CSID.fee = CSID.fee + chilometersCalculation(CEID.endPoint,EID.endPoint) * cost / NOP2;
        ELSE
                SID.fee = SID.fee + chilometersCalculation(CEID.endPoint,EID.endPoint) * cost / NOP1;     END-IF
END-IF
        END
```

The algorithm takes in input the ID of the request for which is needed the calculation of the fee.
Actual time is a function that return the current system's actual time.
ChilometersCalculation(startPoint, endPoint) is a function that simulates the real method name in the google maps API. It computes the best route between startPoint and endPoint and return the distance between the two point in chilometers, which is saved into the variables km1 and km2.
All the other variables (NOP,SID, etc..) are initially null and their value is retrieved by queries on the database.

TAXI SEARCHING ALGORITHM

This algorithm finds the queue with at least one available taxi of the nearest zone to the zone without available taxis containing the starting point of a pending request.

```
Begin FUNCTION ( IOFQ ){            //IOFQ means IndexOfCurrentQueue;
```

```
	find = false;			//boolean variable;if it is true, a taxi has been found.

	Y = null;				//the queue containing at least one available taxi.

	AD = null;				//Set of adjacent zones of the current zone such that x
is

						//the current zone queue

	i = 0;				//index

	x = IOFQ;				//the current zone queue

	IF Queue(x) isEmpty Then
		AD = FindAdjacentZone(x);

		FOR each i in AD DO
			IF Queue(i) !isEmpty && !find
				find = true;
				y = i;	END-IF
		IF !find
			FOR each i in AD DO
				y = FUNCTION(i);
		ELSE
			return y;		END-IF
	ELSE
		y = x;
		return y;
END
}
```

The algorithm takes in input the index of the queue of the zone where the ride is required and return the index of the most closed queue zone with a free taxi.

DESCRIPTION OF THE ALGORITHM THAT FINDS A COMPATIBLE ROUTE TO A PENDING SHARED REQUEST

The Shared Ride Manager Component receives the Id of a shared request just created by a client through the Request Manager.

The Shared Ride Manager queries the database and finds the start point and the end point correspondant to the ID of the request, and saves them in two variables StartPendReq, EndPendReq.
The component scan all the tuples in the request table that fulfill all the following conditions:

- IsCompleted field set to false
- IsShared field set to true
- StartHour value at most one minute before the current system time
- The sum of the number of passengers of the pending request considered and the candidate shared request is not bigger than 4

that minimizes the following formula:

- chilometersCalculation(StartPendReq, StartPoint) +chilometersCalculation (EndPendReq, EndPoint)

where StartPoint and EndPoint are the coordinates of the candidate shared requests considered at the moment.

This is the sum of the distance between two coordinates, so it's a linear distance between two points on the Earth. We assume that our city doesn't contain places really near to each other in linear distance which are instead almost not reachable through taxi routes, otherwise these last heuristic must be corrected.

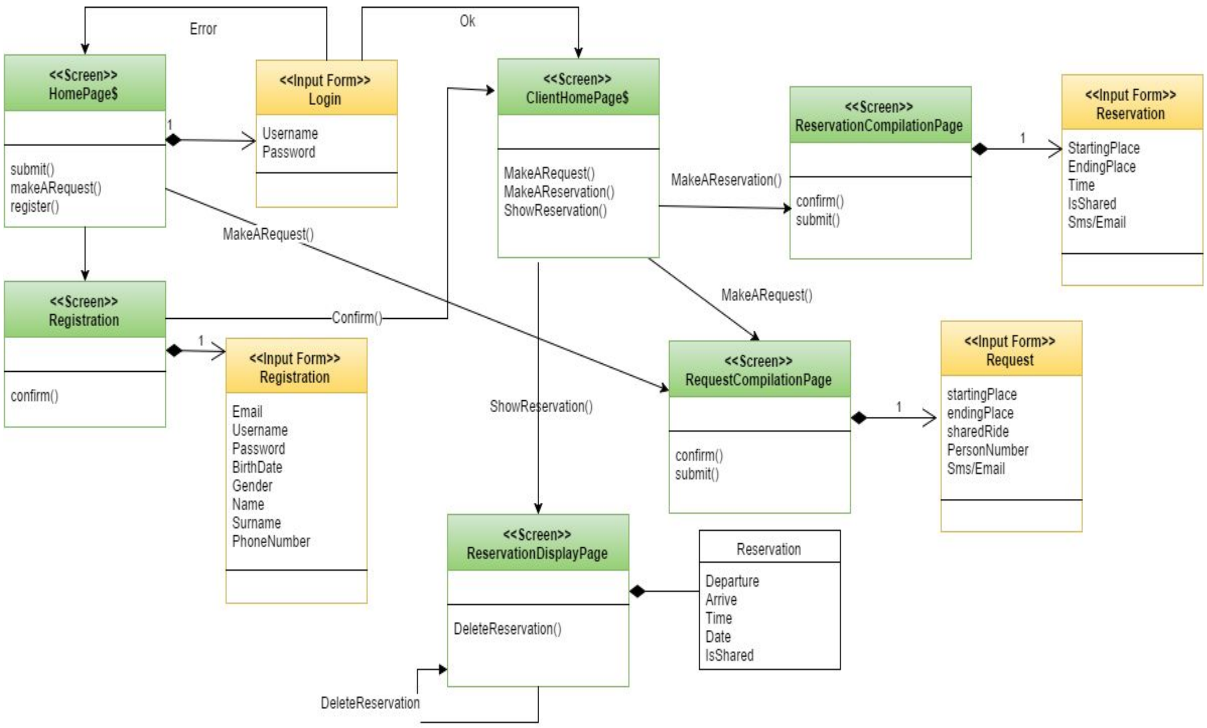
Let's call the coordinates of the request that satisfies and minimizes that formula as StartPointCandidateReq and EndPointCandidateReq.
Then the algorithm computes

- the maximum between linear distances (StartPendReq,EndPointCandidateReq) and (StartPointCandidateReq,EndPendReq)
- the linear distances (StartPendReq,EndPendReq) and (StartPointCandidateReq, EndPointCandidateReq), which are the distances of the single requests if they aren't shared.

If one of these last two distances is 20% longer than the first of these last three linear distances, the rides are not compatible, otherwise the heuristic has found that the candidate shared request is compatible and the system can save the Id of the candidate request on the Request table in the field 'Shared With' and start the procedure for sharing a ride.

4. USER INTERFACE DESIGN

We have already provided the project mockup in the RASD document.
Here with the UX model it's provided how all screens are connected and their interactions.



5. REQUIREMENTS TRACEABILITY

All functional requirements expressed in the RASD are respected in the design proposed by this document. Here below is described which components are involved in order to achieve a certain functional requirement.
All the fulfilled requirement in the following list are ordered in the same way of the RASD requirements in order to easily compare this section with the 3.2 section of our RASD.

REGISTRATION

- The UserManager component will provide a specific interface providing the user the possibility to fill the registration module. These data will be then sent to the AccountManager that will check their consistency save them into the database and actuate the effective registration.

USER LOGIN

- The UserManager component will provide the Login interface and interacting with the AccountManager component will grant the login functionality.

GENERIC REQUEST MANAGEMENT(valid both for requests and reservations)

- The RequestManager will send request confirmations using sms or email notification according to the user choice retrievable from the 'Contacted by' field ad can be seen in the ER description part.
- The RequestManager will deal with all inconsistent generic requests compilation signalizing the UserManager the error.
- The RequestManager interacting with the UserManager will show the fees and the waiting time for the arrival of the taxi to the client.

SHARED REQUEST / SHARED RESERVATION MANAGEMENT

- The UserManager component will provide a form in which the user can fill a request/reservation form.
- The RequestManager component interacting with the SharedRideManager will compute if there exists a compatible shared ride and in that case will calculate the shared ride cost, and notify it to the taxi driver (through the TaxiDriverManager) and to
  the clients involved (through the Request Manager).

RESERVATION MANAGEMENT

- The UserManager component will provide to clients an input form for the compilation of a reservation or a shared reservation.
- The UserManager interacting with the ReservationCollector will save the new reservations in the database and will let the clients to view all their booked requests not yet happened. There will be the further possibility to delete a reservation if this action is performed ten minutes before the starting hour.

QUEUE MANAGEMENT

- The QueueManager component will deal with the queue of available taxis and interacting with the TaxiDriverManager will let the TaxiDriver to change his status in the system and change his queue zone to another one according to his GPS position.

GPS LOCALIZATION

- The TaxiDriverManager component will allow taxi drivers to visualize his position in the city's map by using the GPS localization and, if it's the case, the path he has to follow. In order to achieve this it will use some external methods provided by the Google Maps API.
- The User Manager will permit the client to use GPS localization to express his position during a generic request compilation.

TAXI DRIVER FUNCTIONALITIES

- The TaxiDriver Manager interacting with the Queue Manager will let a taxi driver to change his state from "busy" to "available" and conversely.
- The TaxiDriverManager interacting with the RequestManager will let the taxi driver to accept or refuse a generic request.

TAXI DRIVER LOGIN

- The TaxiDriverManager will provide a login form to the taxi driver and, interacting with the AccountManager for checking the data consistency, it will grant the login functionality.

6. REFERENCES

Draw.io for the syntax of the ER model

Hours of work:

Alessandro Dell'Orto: 22 h
Andrea Brunato: 20 h
Lorenzo Costantini: 20 h

RASD Document Corrections:

- We haven't managed to satisfy the functional requirement that finds the best compatible route. Our algorithm if it manages to find a compatible shared ride has no guarantee of optimality because we use an heuristic based on linear distances through coordinates
- We haven't said in the RASD that we intended to use the Google Maps API.