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 Revision History

1.1.1 Changes in the DD second version

- Order of presentation: sequence diagram ordering is now more meaningful; considerations about zones in section 2.H has been moved to the algorithm design section; a diagram previously in section 2.D has been moved in section 2.B and the one present in the first version has been eliminated.
- The algorithm that finds a compatible shared route has been corrected in his last part, and the taxi queue management is now described in detail.
- Some corrections have been made to the interfaces of the Component Diagram in section 2.C and 2.F during the realisation of the integration testing plan assignment. However the number and type of components isn't changed.

1.1.2 RASD 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, it has no guarantee of optimality because we use an heuristic.
- We haven't said in the RASD that we intended to use the Google Maps API.
- A future possible implementation of the RASD (the taxi searching algorithm) is now described into the Algorithm Design section and ready to be developed.

1.2. 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.

1.3. 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.

1.4. Acronyms

PK: in the Entity Relationship model it means primary key
UX: User eXperience
UML: Unified Modeling Language
DBMS: Database Management System
DB: Database
EIS: Enterprise Information System
JSF: Java Server Faces
JPA: Java Persistence API
ER: Entity Relationship
SOA: Service Oriented Architecture
EJB: Enterprise Java Beans
RASD: Requirement Analysis And Specification Document
DD: Design Document

1.5. Reference Documents

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

1.6. Document Structure

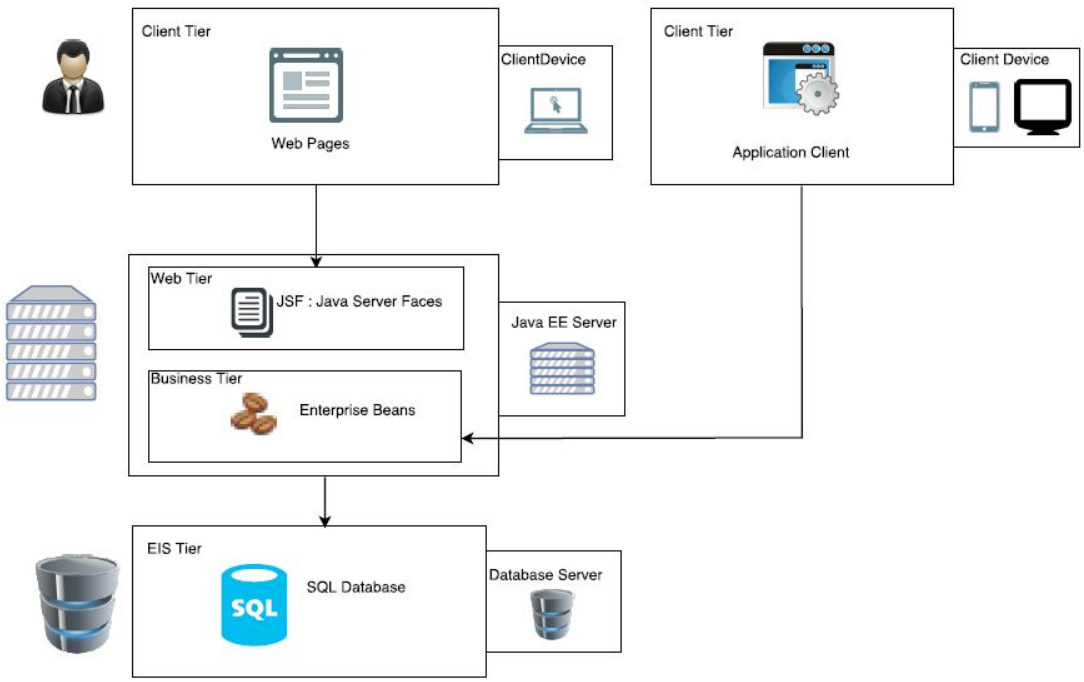
1) **Introduction**: This section contains general informations about our document, the purpose of the realization of the DD and his goals, plus the list of definitions and acronyms used in the following sections of the document.
2) **Architectural Design**: This part describes the most important design 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.

2. Architectural Design

A. Overview

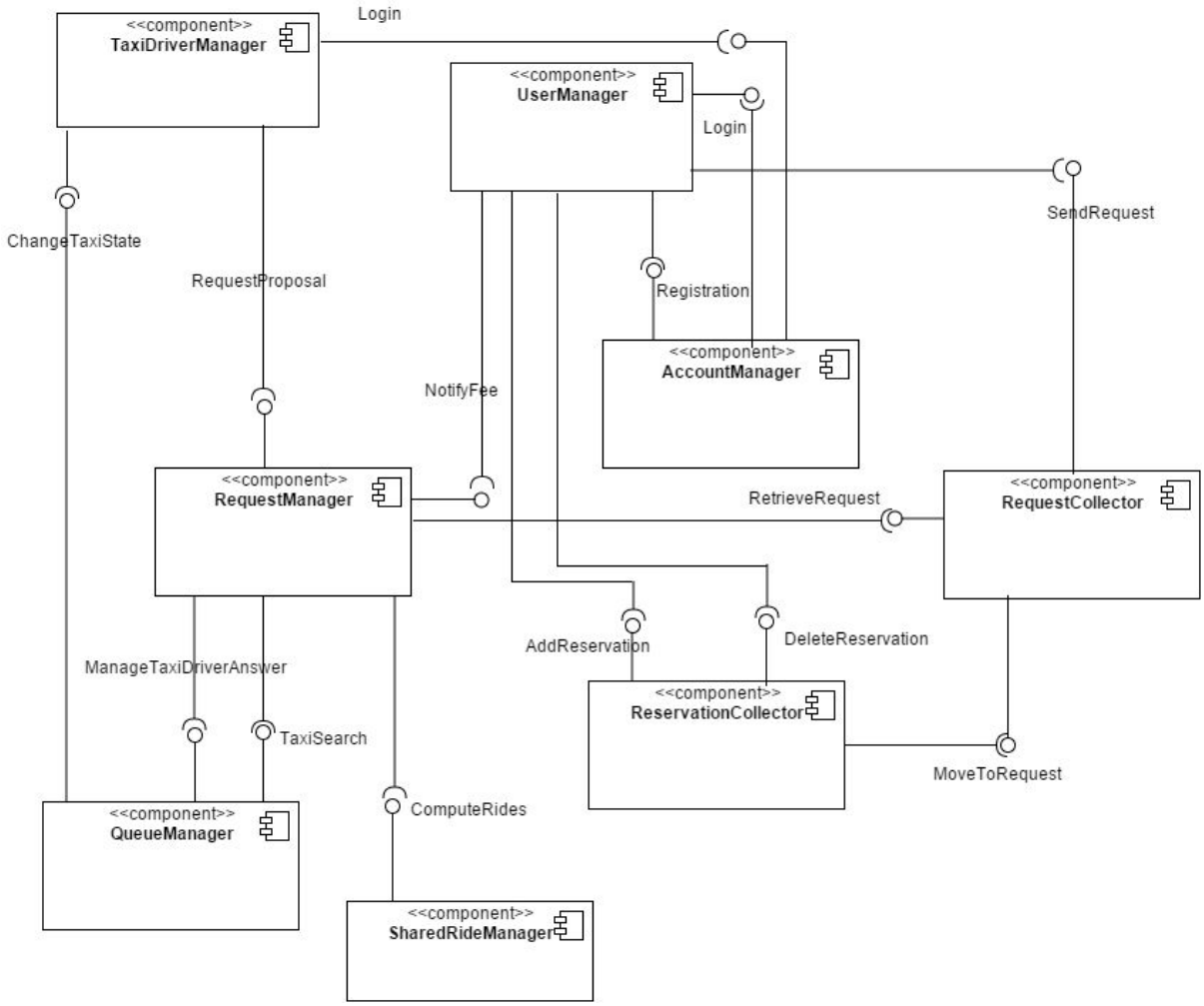
The section B shows the high level components of our system. The components of the system will be more deeply analyzed in the section C regardless of tiers through a component diagram, which also specifies the interfaces between them. The meaning of all interfaces can be read in section F.
Section D 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 describes 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

This component diagram presents the all the components of the application without considering tiers. They will be mapped in their correspondent tiers in the deployment view, and their functionalities are described on the following section “Components Description”.



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 Email 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 data structures and continuously keeps updating them whenever a new taxi driver logs into the system or change his status or changes his zone while available. It manages a queue for every city zone and a set that contains the logged in taxis in the “busy” state.
Every active taxi sends automatically the information about his position every 30 seconds, and the component checks if the position given by the Taxi Driver Manager (thanks to the GPS) is still inside the previous zone or not 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 taxis interacting with the system this component will be implemented as a singleton bean.

ReservationCollector: This component deals with the reservations. Its main functionalities are the creation/deletion of new reservation instances thanks to the interaction with a Reservation Entity Bean which will represent the Reservation table of the database. It will also move a reservation that is going to start in 10 minutes into the Request Collector queue.

Request Collector: This component deals with the requests. Its main functionalities consists into the interaction with the Request Entity Bean for the creation/ deletion of new tuples of the database request table, 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 uses the Client Entity Bean, the Taxi Driver Entity Bean and the Taxi Entity Bean for performing its operations described before.

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 depends on the Request Entity Bean to search for compatible rides and invokes Google Maps API methods in order to find a compatible route path to the one given by the Request Manager.

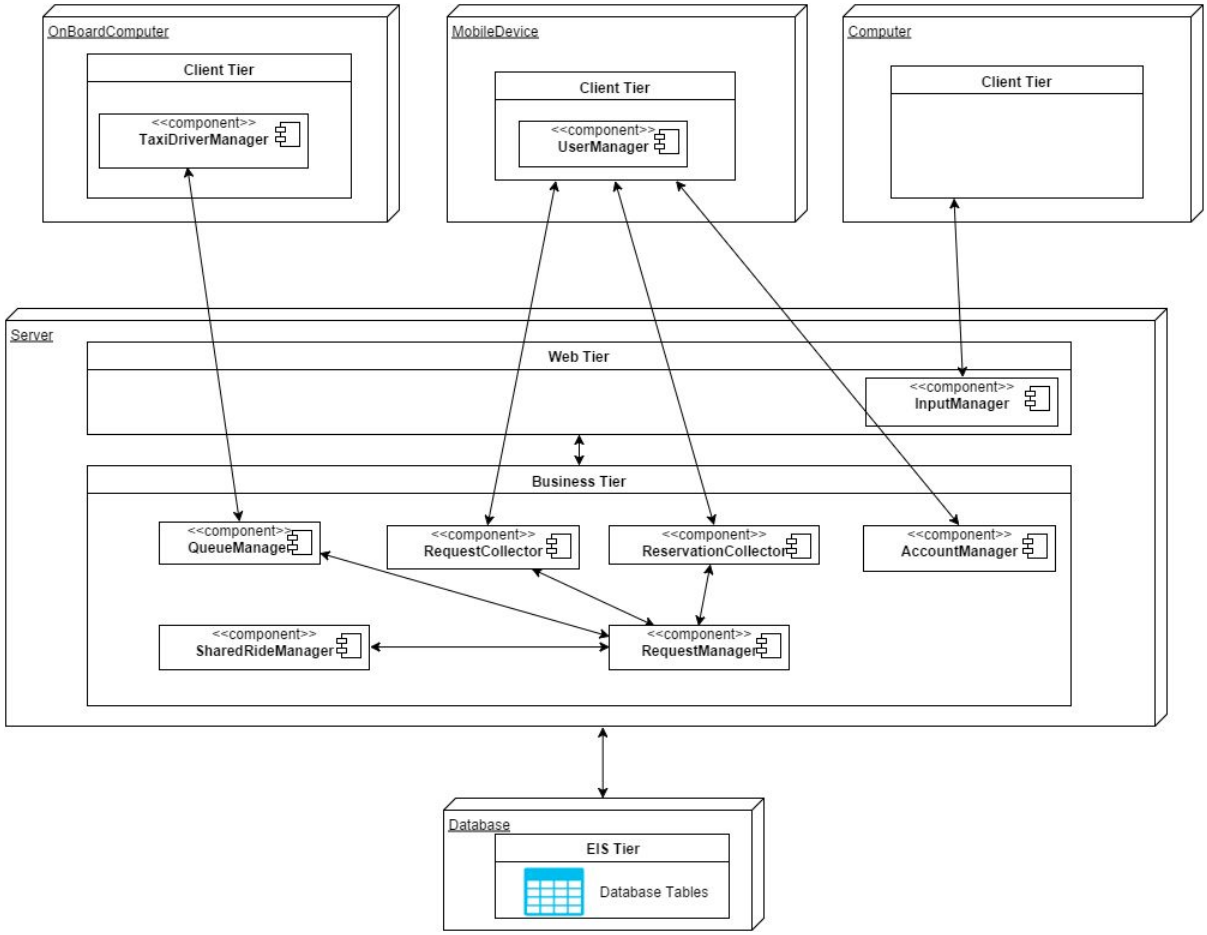
User Manager: This component is completely devoted to deal with the user interaction with the system. It offers a GUI in which the user can select all the operations implemented by the system,retrieves the user position through GPS and can send data through the internet.

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

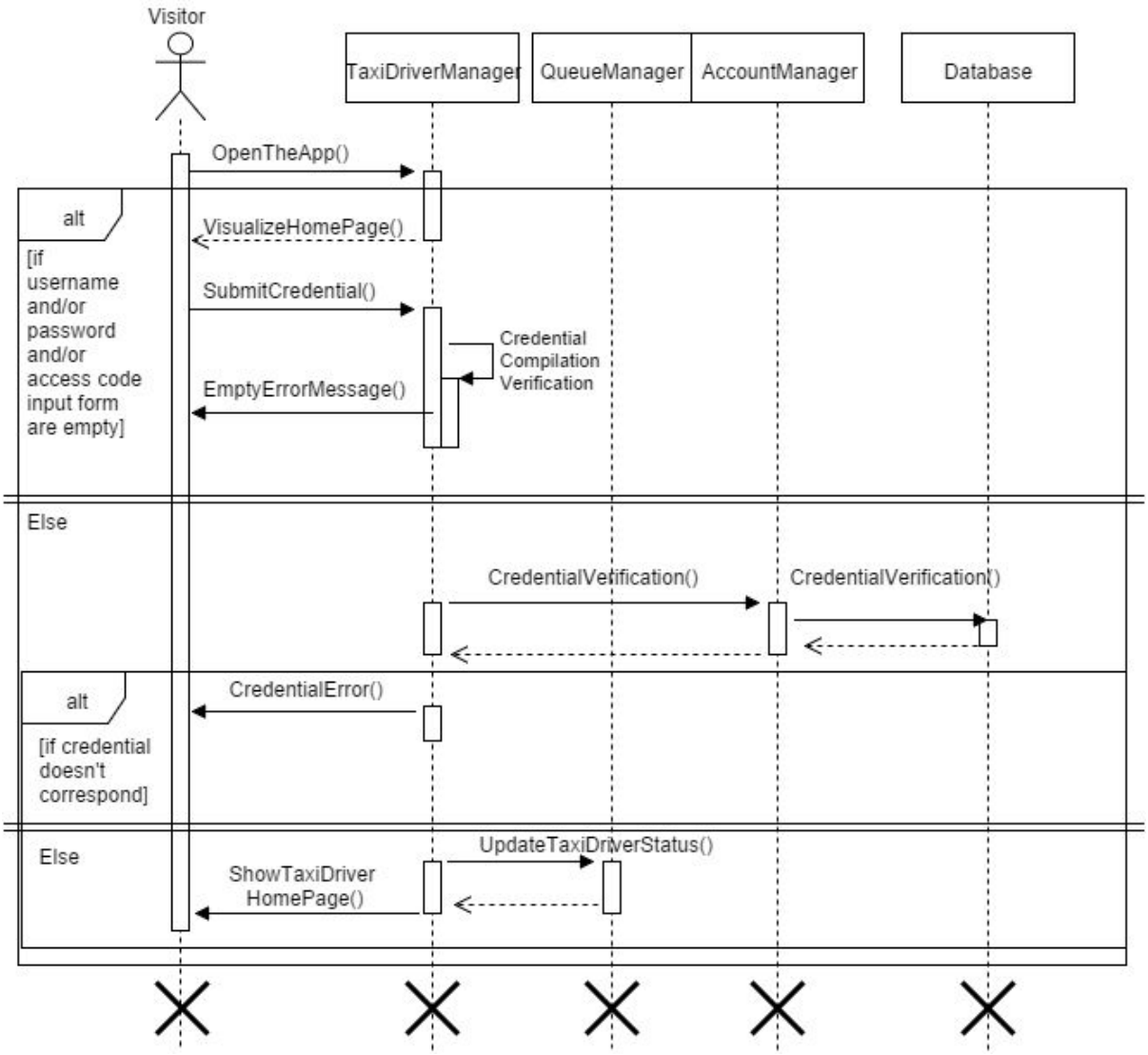
This deployment diagram mixes the previous ones of section B and C giving a unique and quite detailed description of our system



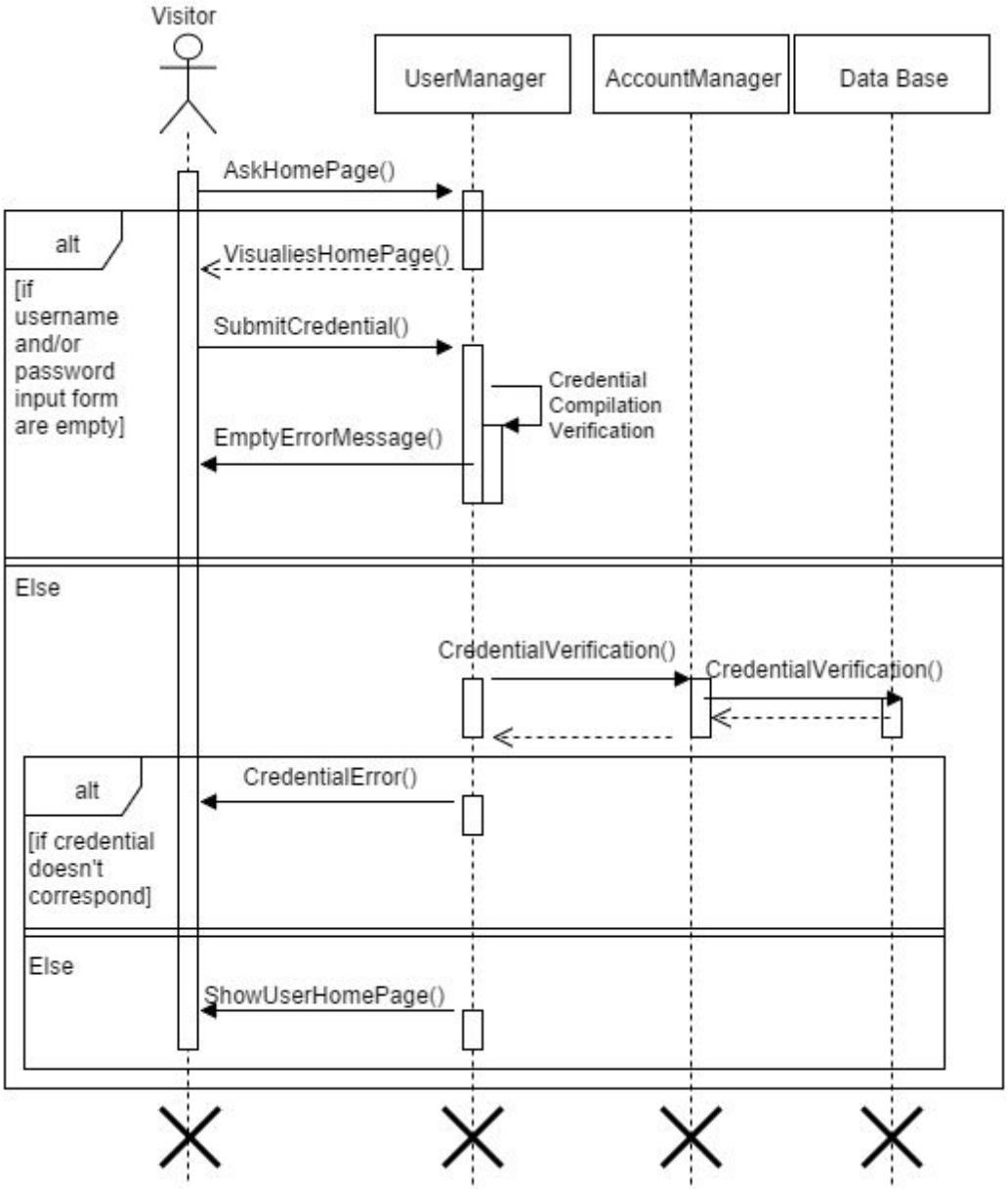
E. Runtime view

Here all the functionalities described in the sequence diagrams and activity diagrams of the RASD document have been analysed taking into account the components previously described. The interaction between components for every functionality is better explained by the following sequence diagrams.

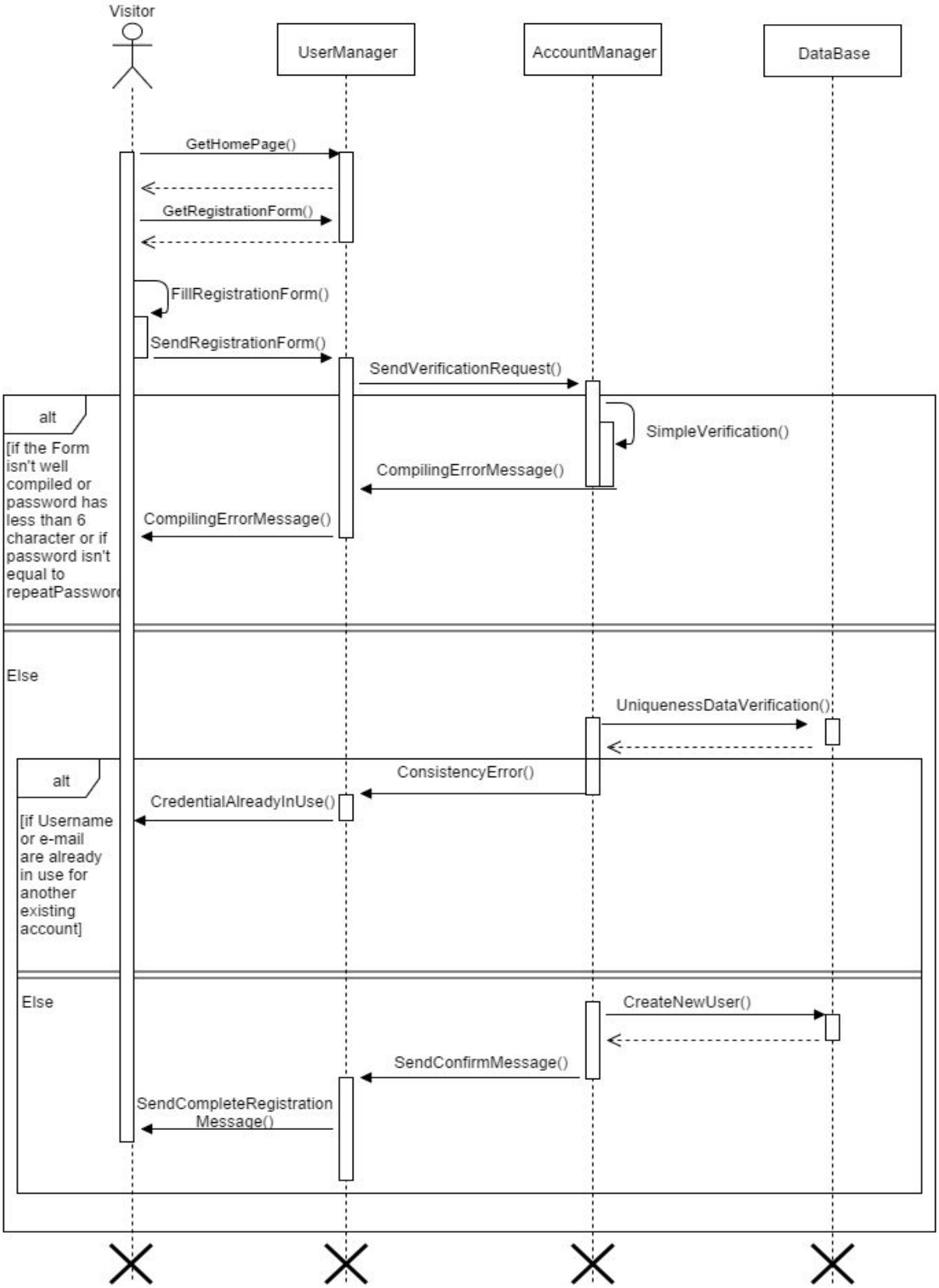
1.Taxi Driver Login



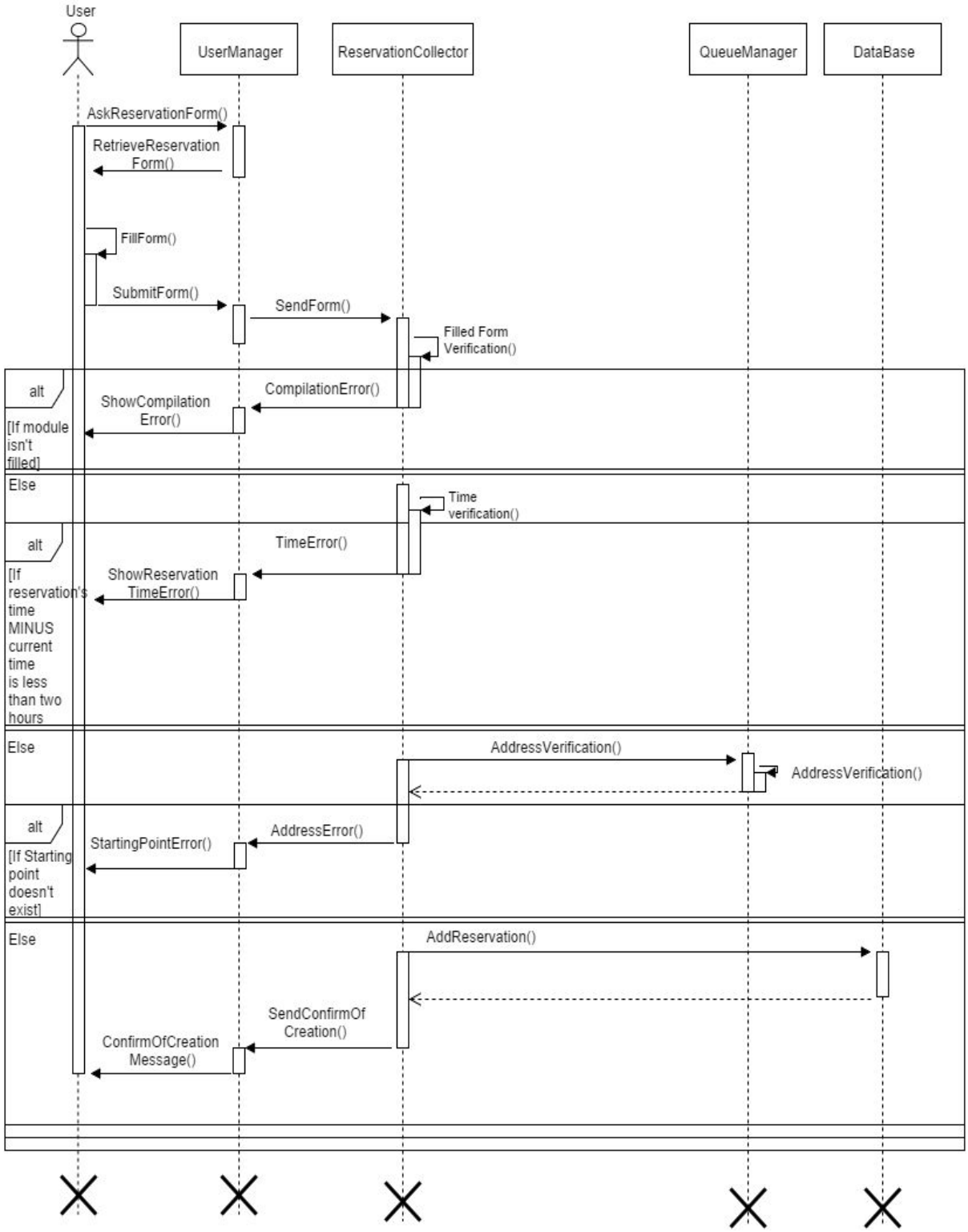
2.Client Login



3. Client Registration

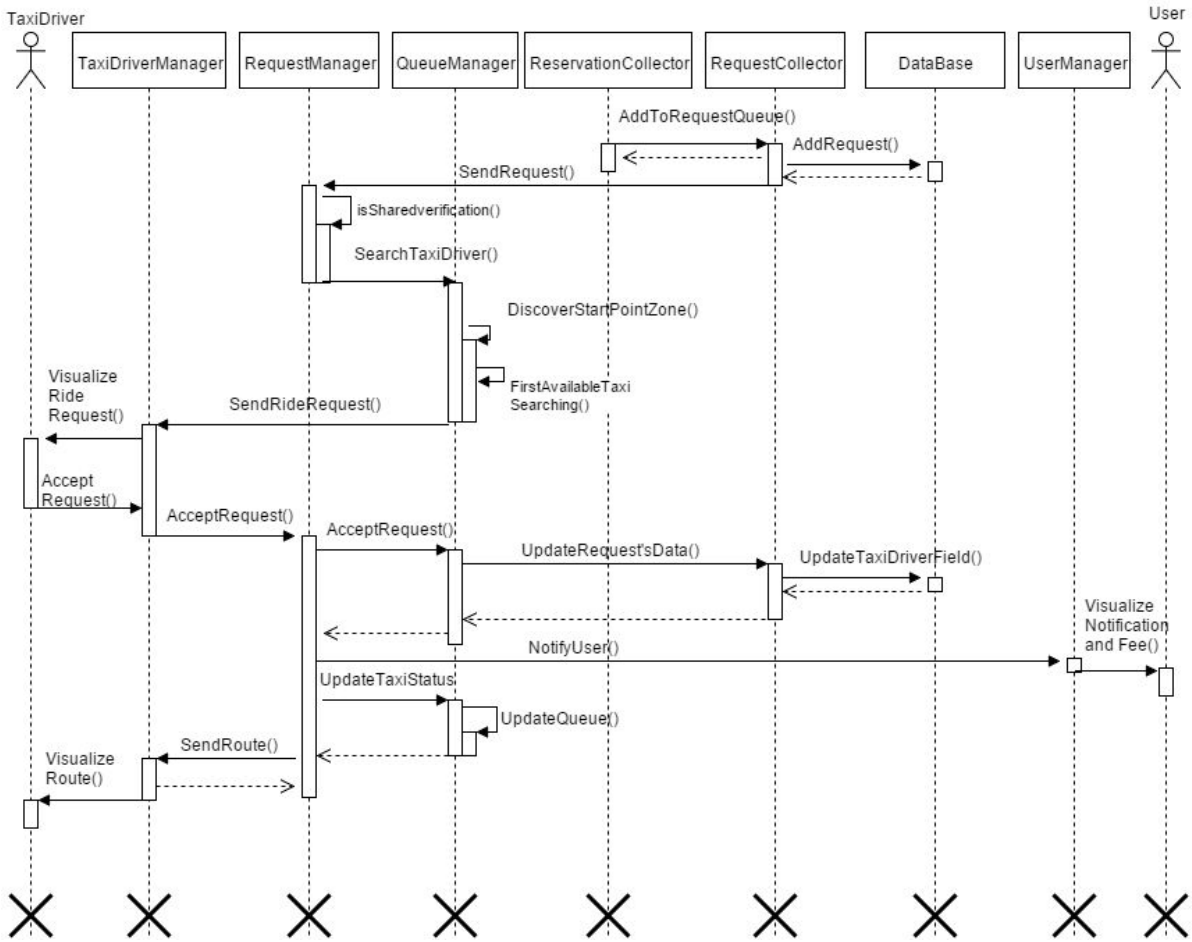


4.Create Reservation

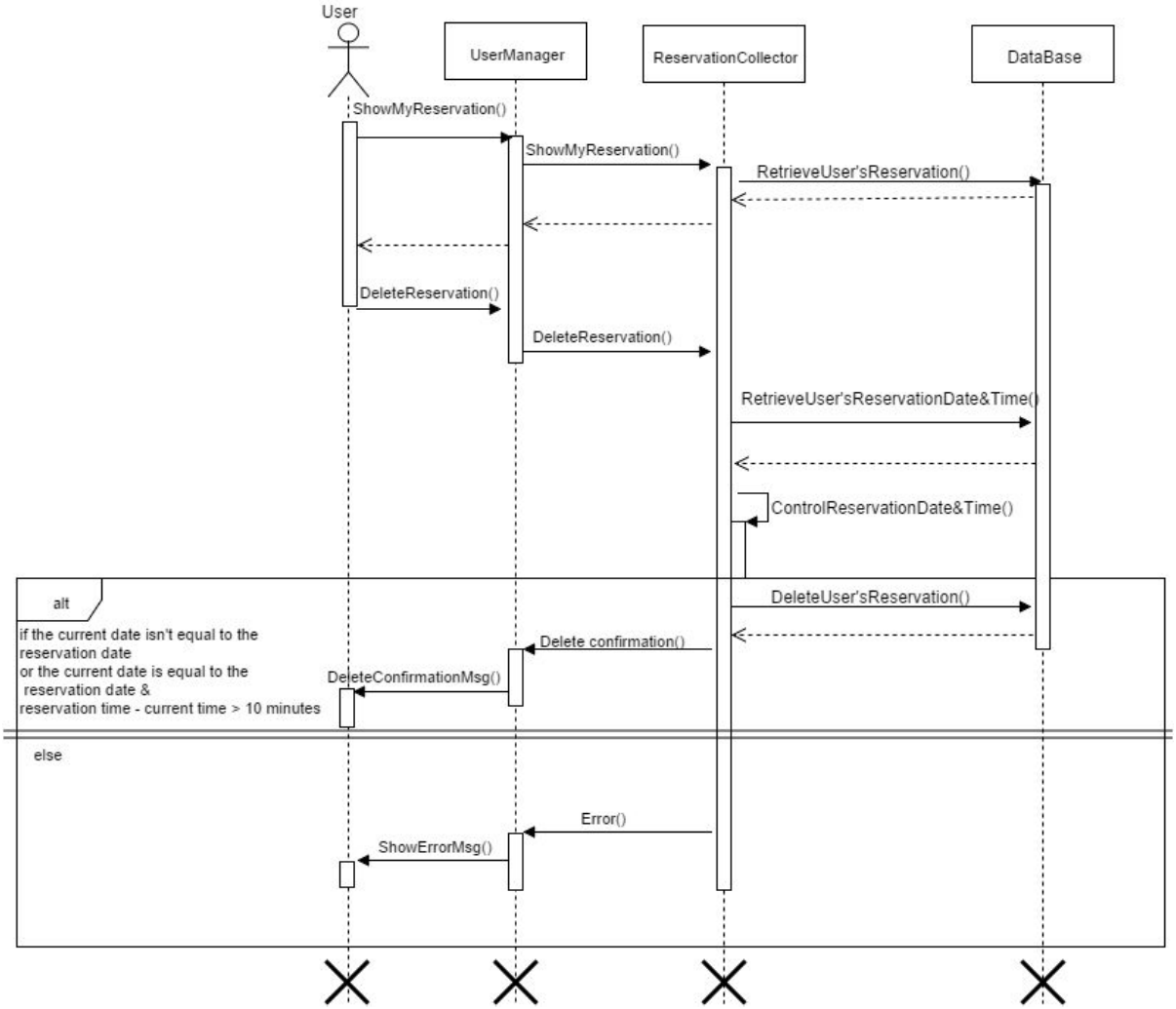


5.Start of Simple Reservation

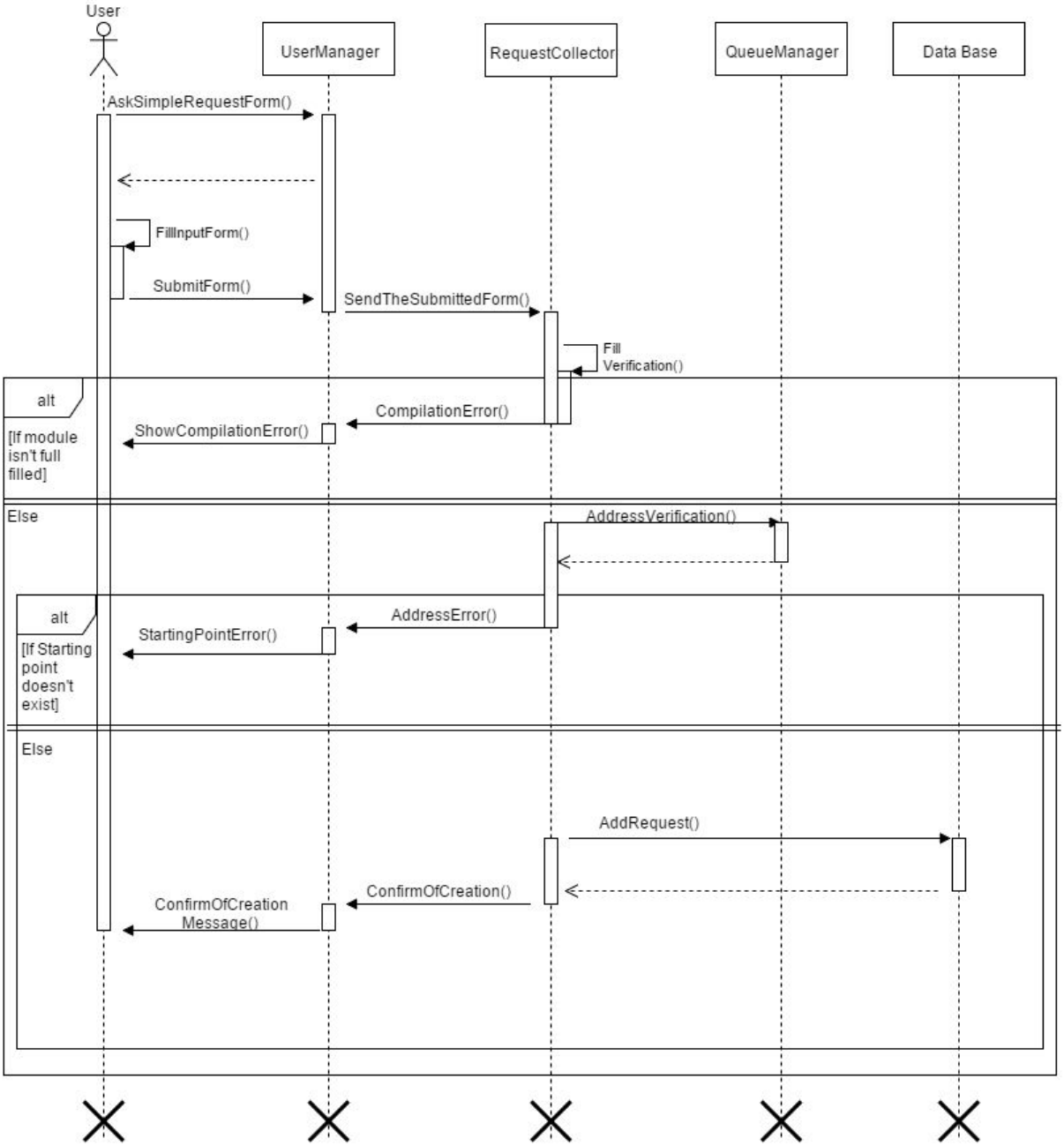
The simple reservation is the one without shared option, according to the definitions given in the RASD which are all still valid for this document.



6.Delete Reservation

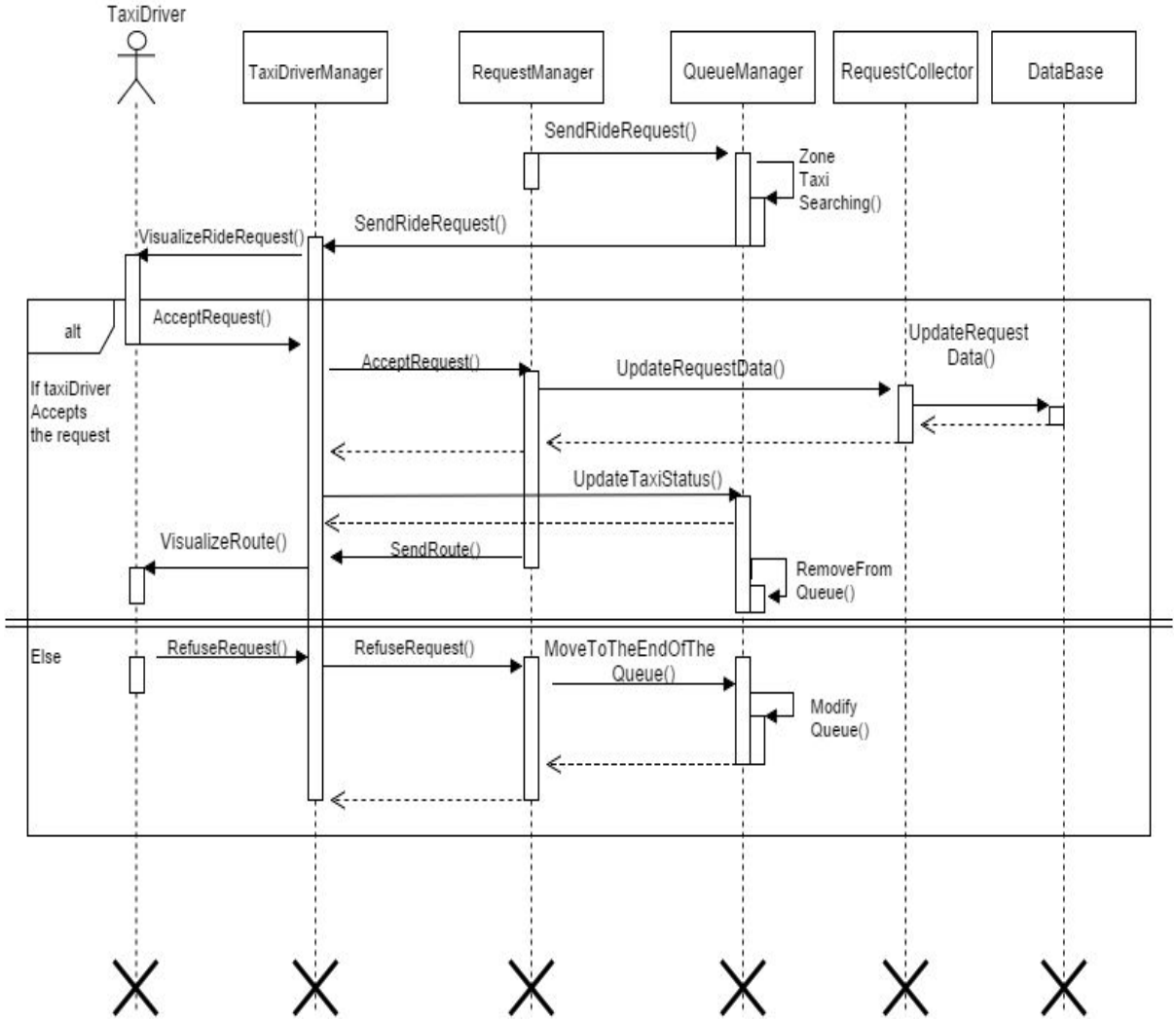


7.Create Simple Request

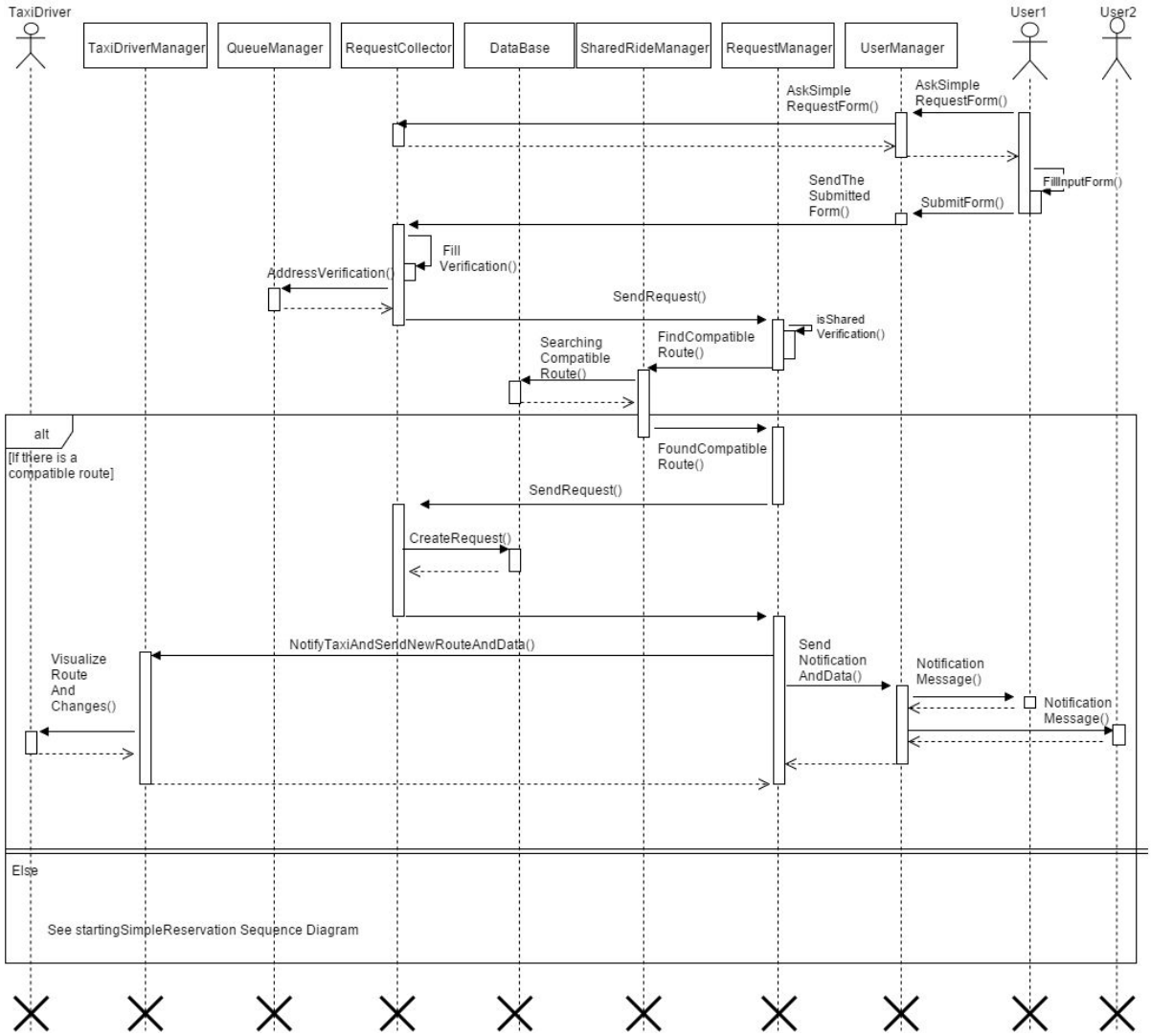


8.Answer to Generic Request

This sequence diagram is valid for both requests and reservations.



9.Creation of Shared Request



F. Component interfaces

Each component exposes the following interfaces:

- **RequestManager**:
    - ***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.
      Moreover 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.
    - ***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 User Manager and allows 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.
    - ***Login (Taxi Driver)***: This interface is provided to Taxi Driver 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.
    - ***Login (User)***: This interface is provided to the User Manager component. It allows a client to log into the system. The Account Manager will be responsible of checking the correspondence 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.
    - ***ManageTaxiDriverAnswer***: This interface is provided to the Request 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 available 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).
    - ***ChangeTaxiState***: This interface is provided to the Taxi Driver 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 available, he is added to the correspondent queue, while if he set himself to busy he is removed from that queue.

- **RequestCollector:**

    - ***SendRequest***: This interface is provided to the User Manager. It allows the client to make a simple request, which is instantly 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 the model 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 Email service to contact clients after the submission of a request.

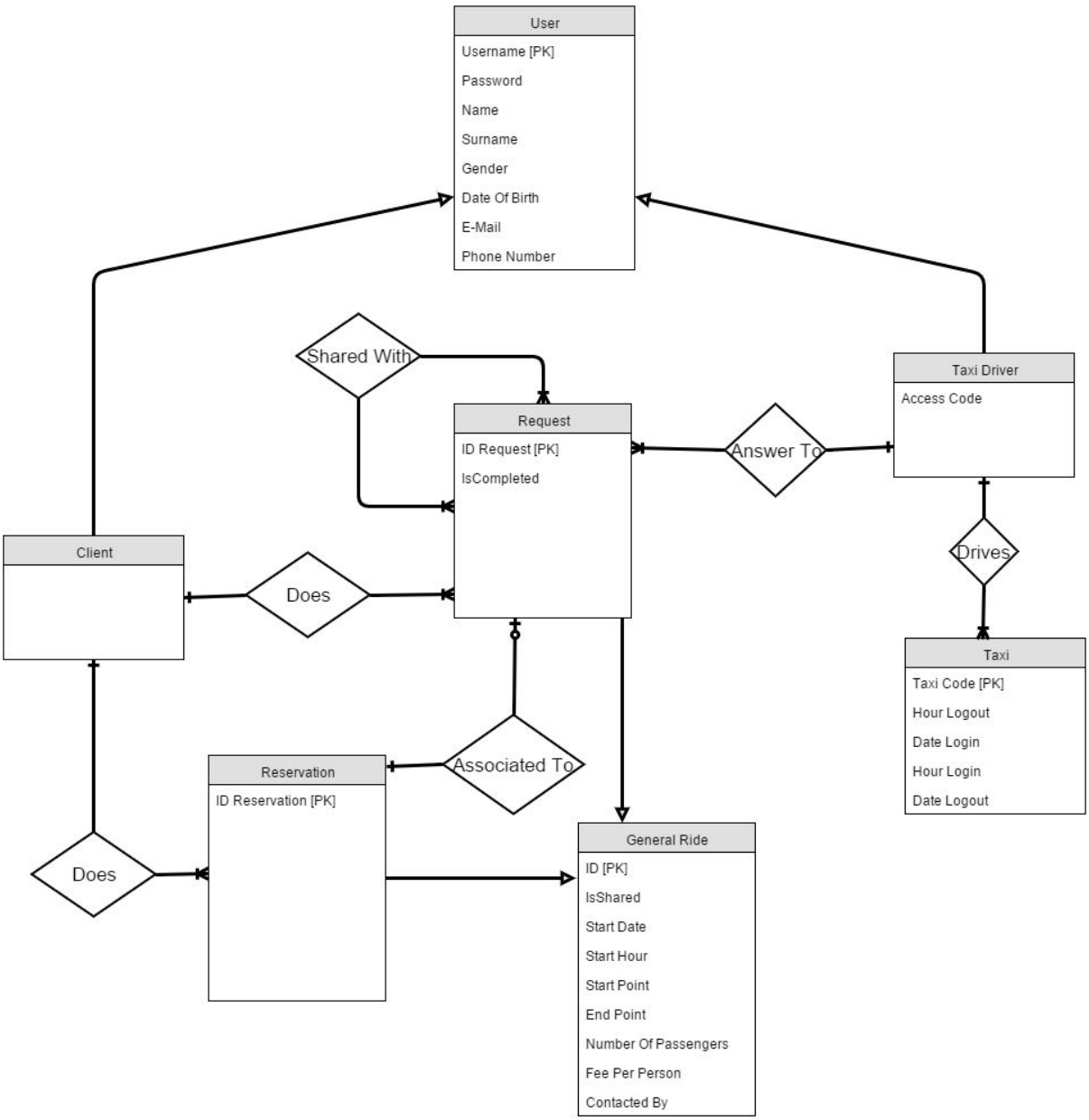
We have chosen to design the system architecture with **components and connectors** in order to increase maintainability and maximize the decoupling between classes. In this

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 an 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:

- The Contacted By field can contain only the strings ‘SMS’ or ‘E-Mail’.
- The informations about current availability of taxis, their position 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.

The algorithm takes in input the ID of the request for which is needed the calculation of the fee. The algorithm is dependant to the choices made in the logical schema of the database in section 2.H, and IDRequest.SharedWith means that the field Shared With of the request in the DB with that ID is accessed.
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 kilometers, 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.

```
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 kilometer, 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 SharedWith == 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 = kilometersCalculation(STARTP,ENDP);
        IdRequest.fee = (km1 * cost + fixedCost(ActualTime)) / NOP1;
ELSE
        IF IdRequest.time < IdRequest.SharedWith.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 = kilometersCalculation(CSID.startPoint , IdRequest.endPoint);
        km2 = kilometersCalculation(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 for every request of the ride

        CSID.fee = kilometersCalculation(CSID.startPoint,CEID.endPoint) * cost/ (NOP1 + NOP2) + fixedCost(ActualTime);
        SID.fee = kilometersCalculation(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 last part of the algorithm distinguish the shared requests of the following type:

- User 1 starts the ride - User 2 joins the ride - User 1 ends the ride - User 2 ends the ride.
- User 1 starts the ride - User 2 joins the ride - User 2 ends the ride - User 1 ends the ride.

With user 1 we intend all the passengers associated to the user 1 request (the same holds for user 2 obviously).

TAXI SEARCHING ALGORITHM

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

```
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 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 closest queue zone with a free taxi.

CITY ZONES REPRESENTATION AND TAXI STATE CHANGE ALGORITHM

- 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, because it is an immutable data (saving for sporadic redefinitions of zones) with a low number of instances. A zone is characterized by an identifier number, four coordinates expressed in latitude and longitude, which are the geographical boundary defining it, and the list of adjacent zones. 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 latitude and longitude belong both to the interval described by the four coordinates.
- The taxi is represented into the application as an identifier number associated to the last coordinates registered. Whenever the taxi identifier is found in a queue or in a set, the coordinate value is updated.
- GPS informations are sent every 30 seconds by both busy and available taxis and they are also sent extraordinarily when a taxi driver set himself from busy to available or viceversa.
- The algorithm implements the State Machine Diagram of the RASD.

<u>30 seconds from the last position updates are passed, no zone matches with position</u>:
If after scanning all the coordinate intervals of the zones there doesn't exists one which satisfies the condition on both latitude and longitude, it means that the available driver is now gone outside the city and therefore his state will be set to busy by the Queue Manager according to the State Machine Diagram of the RASD.

The system will start to search the taxi ID first into the set of busy and logged in taxis because the most probable option is that even 30 seconds before it was still outside the city, and in case it isn't found there, the search will be performed into active taxis queues without following a particular order (starting from the queue representing the zone with the lowest identification number is an option).

If the taxi instance is found in a queue, it's deleted from that data structure and added into the set of inactive taxis with an updated coordinate value. If the taxi instance is found in the set, only the coordinate value will be updated.

<u>30 seconds from the last position updates are passed, a zone matches with new position</u>:
The system searches the taxi instance into the zone in which the taxi is driving currently, because it's more probable to find it there than assuming that it has changed position in the last 30 seconds.

If the taxi instance isn't found there, the system will search into the queues adjacent to zone considered at the beginning.

If the taxi isn't found again, the system will search in the set of busy taxis assuming that the taxi has just reentered into the city border.

If the taxi isn't found yet, the system will search through every queue randomly.

Assuming that the taxi has remained in the same zone of the last measurement, the system will only update the taxi coordinates.

In all the other cases the taxi instance is deleted from the previous data structure and added at the bottom of the queue of the matching zone with the new coordinate value.

GPS is sent after a state change from busy to available:
the occurrence of this event can be detected by a special message in union with the new coordinate position.
The system first searches if there exists a zone which satisfies the coordinate conditions: if there doesn't exists such a zone it means that the taxi driver has tried to put himself available despite being outside the city. In this case the state change operation is invalidated and only the new coordinates will be memorized in the set of busy taxis.

Otherwise, the zone found is memorized and the system will only have to search into the set of busy taxis because the taxi will surely be there. The instance associated to it will be deleted from the set and moved at the bottom of the queue which satisfies the conditions on coordinates.

GPS is sent after a state change from available to busy:
as before, the occurrence of this event can be detected by a special message in union with the new coordinate position.
When this happens, the system will search randomly to every queue in order to find the taxi instance. When it's found, it is deleted and added into the set of busy taxis with the new coordinate value.

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 corresponding 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 behind the current system time. This is important because after some time that the ride has started, the value in the Start Point field will be really different to the actual position of the taxi. The latter is infeasible to retrieve for every single ride from the interaction of the database with the all the taxi queues.
- StartHour value at most four minutes ahead the current system time. It's not fair to let a client wait more than this time for the start of his ride.
- The sum of the number of passengers of the pending request considered and the candidate shared request is not bigger than 4, the number of seats in the car

that minimizes the following formula:

- (StartPendReq - StartPoint) + (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 StartCandidateReq and EndCandidateReq.

- First, the algorithm uses the google Maps Api for the calculation of the best route that passes through StartPendReq, StartCandidateReq, EndPendReq and EndCandidateReq (=SharedRouteDistance. This route will be the one visualized by the taxi driver if the shared route computed will satisfy the validation condition (see below).

Then the Google Maps API is used again for:

- the calculation of kilometers between the two start points = StartDistance
- the calculation of kilometers between the two end points = EndDistance

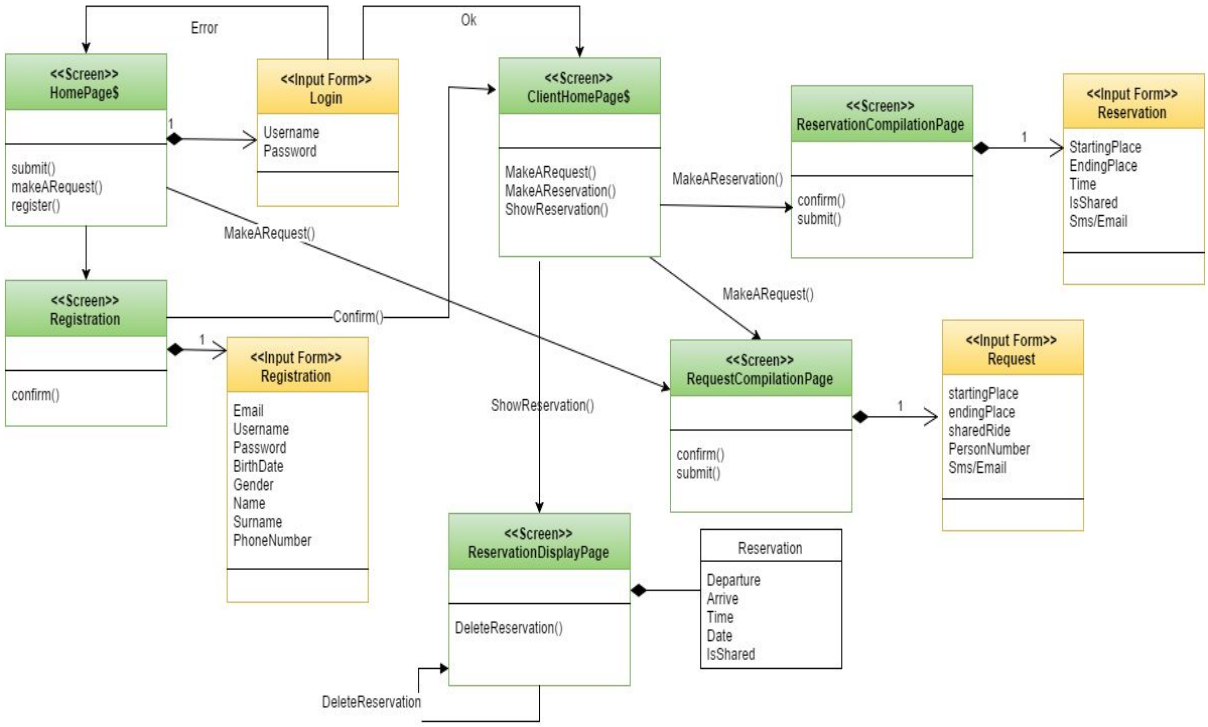
Validation Condition: StartDistance+EndDistance <= 0.2 * SharedRouteDistance.

So if the sum of distance in kilometers calculated with Google Maps (the linear distance heuristic is used only to find a candidate request) between the start points and end points is less than the 20% of the kilometers to be done in case of shared route, the clients are guaranteed to share most of the path and to save money.

If the rides are not compatible, the shared route calculated is discarded and the requests won't be shared, 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'.

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. Hours Of Work

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