My Taxi Service

Design Document

Authors:

Andrea DONATI	{andrea4.donati@mail.polimi.it}
Gabriele CARASSALE	{gabriele.carassale@mail.polimi.it}
Manuel DELEO	{manuel.deleo@mail.polimi.it}

Prof: Elisabetta Di Nitto

Milan, December 4, 2015

Contents

1 Introduction

1.1 Purpose

The goal of this Design Document is to describe the architecture of myTaxiService system, its components, the logical and conceptual design of data, the scheme of its main algorithms and the choice of tradeoffs. It will also include the description of the UI flow through a UI Diagram. This document will be used by developers in the further stages of the project, such as testing and implementation.

1.2 Scope

This document is intended to be a detailed design and architectural description of MyTaxiService, for which we have already provided a Requirements Analysis and Specification Document. So, this document provides additional and more detailed information than RASD, and describes our software from a different point of view. For this reason, the Design Document and the RASD document developed earlier have to be both considered in order to have a general view of our project in all its aspects.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **User - Passenger:** person who wants to make use of the system. A user should be able of taking advantage of the functionalities listed in the goals section.
- **Taxi driver:** person who owns a taxi and a license. A taxi driver should be able to use the functionalities listed in the goals section.
- **Taxi ride:** ride which goes from an origin to a destination, requested by a specific user and managed by a taxi driver.
- **Taxi zone:** portion of the city which is 2 km^2 large. Each of these zones is associated to a queue of taxis.
- **Taxi queue:** ordered list in which every available taxi in the associated zone is stored.
- **Reserved ride:** ride reserved in advance by a user.
- **Shared ride:** ride created by a user who wants to share the ride and its cost with others.

1.3.2 Acronyms

- **API:** Application Programming Interface;
- **ETA:** Estimated Time of Arrival;
- **OS:** Operating system;
- **MVC:** Model-View-Controller.
- **JEE:** Java Enterprise Edition.

1.3.3 Abbreviations

- **[Cn]:** n-th ER or Relational constraint

1.4 Reference Documents

- Requirements Analysis and Specification Document of myTaxiService.pdf
- Assignments 1 and 2 (RASD and DD).pdf
- Structure of the design document.pdf

2 Architectural Design

2.1 Overview

We followed a top-down approach in order to design the system and define the parts of it. We identified the main tiers and, then, we splitted them into components which offer specific functionalities. MyTaxiService system will be designed following a 3-tier architectural style, based on Java Enterprise Edition.

- Client Application
  - User interface
  - Request, reservation and sharing

The client tier will take care of retrieving the users' needs, such as requests and reservations of taxi rides. Through the user interface, users will navigate into the dynamic web pages generated by Java Server Faces framework and both send their requests and see the system's response.

- Business Logic
  - Notification Manager
    - Taxi notification
    - User notification
  - Queue Manager
    - Taxi allocation manager
    - Queues management
  - Request Manager
    - Requests management
    - Google Maps API
    - Payment service
  - Account Manager
    - User check
    - Taxi driver check
    - Sessions management

The business logic tier will take care of performing calculations and logical decisions. It will also manage the interaction between the client and the persistence tier, accessing the database to store or retrieve data.

- Persistence
  - Users
  - Taxi drivers
  - Administrators
  - Zones and Queues
  - Rides

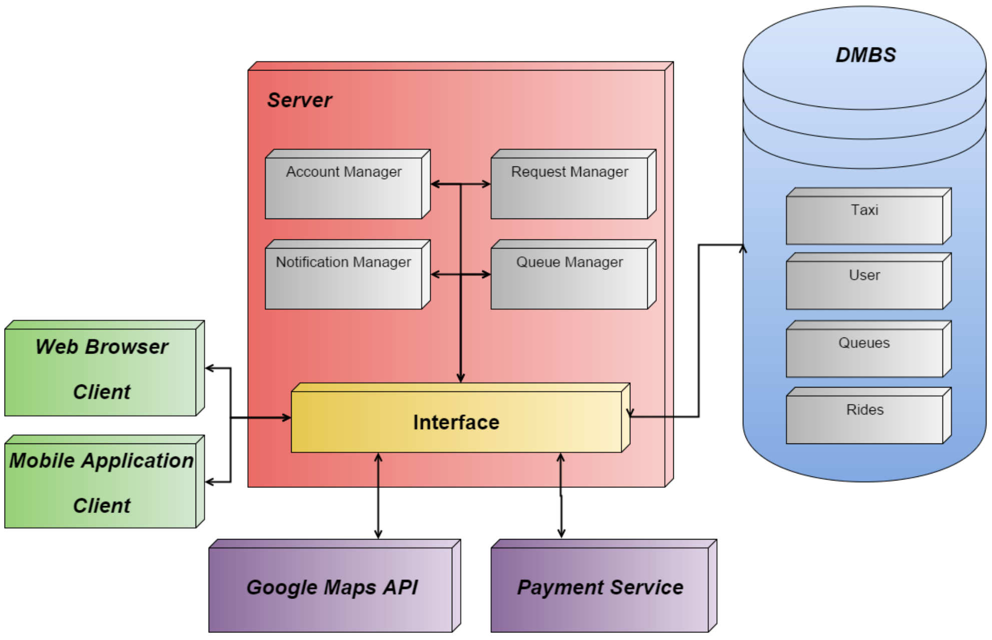
The persistence tier will take care of storing information and of answering the business logic tier's data requests.

2.2 High Level Components and Their Interaction

In this general view of the system we identified the three main system components:

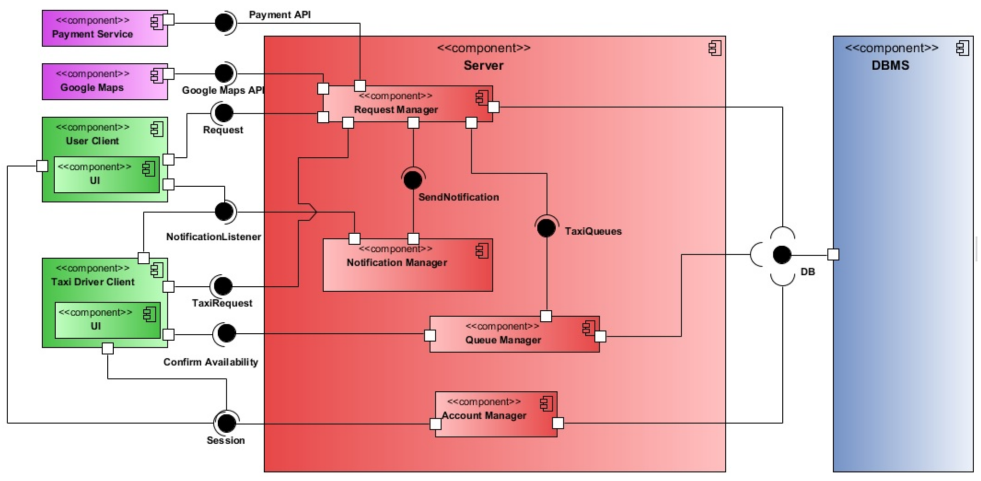
- clients, which can be accessed from a web browser or the mobile application;
- the server, in which requests are handled, coming from the various interfaces;
- the database, where data is stored.

In addition, the diagram contains the external interfaces exploited by the application, such as Google Maps API and the Payment Service interface. The various sub-components of the system and their corresponding interfaces will be analyzed in the following chapters.



2.3 Component View

Here it's presented a more detailed description of the components of the system and their interaction, showing the interfaces used in the communication.

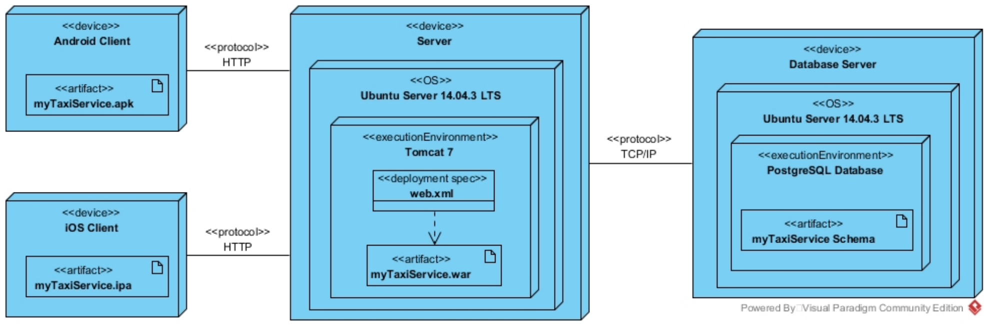

<<component>> Payment Service
Payment API
<<component>> Google Maps
Google Maps API
<<component>> User Client
<<component>> UI
Request
<<component>> Server
<<component>> Request Manager
SendNotification
NotificationListener
<<component>> Notification Manager
TaxiQueues
<<component>> Taxi Driver Client
<<component>> UI
TaxiRequest
Confirm Availability
<<component>> Queue Manager
<<component>> Account Manager
Session
DB
<<component>> DBMS

- **Request Manager:** The Request Manager is the component that manages all the operations related to the Normal, Reserved and Shared Rides. It also handles payments and interaction with Google Maps.
- **Notification Manager:** It manages the delivery of all the notifications to the users and taxi drivers, when it's requested by the Request Manager.
- **Queue Manager:** It handles all the operations between the taxi drivers and the queues. In particular it is responsible for the correspondence between the position of the taxi driver and the correct queue.
- **Account Manager:** It manages all the operations linked to accounts like creation, login and the session kept during the navigation.
- **DBMS:** It is responsible for storing all the data of the system, that can be accessed by other components through the "DB" interface.
- **User Client:** The user client can be both a browser or a mobile application like Android or iOS, it's responsible for showing information to the user and for communicating with the business tier.
- **Taxi Driver Client:** The taxi driver client is the Android or iOS application, and communicates directly with the server.
- **Payment Service:** It's an external service used for making payments with the credit card of the user.
- **Google Maps:** The Google Map Service is accessed through the "Google Maps API" interface and it's used to get the expected times and distances between two given locations.

2.4 Deployment View

In this section it's described, using a simple deployment diagram, how the components of our system are mapped to hardware components.
The mobile applications will be delivered to the users using specific methods depending on the platform, for instance in Android it will be delivered an .apk file while in

iOS it will be released a .ipa file.
The server implementing the business logic will be deployed through a .war file to a Tomcat server environment.
The database will be implemented by a PostgreSQL DBMS containing all tables defined later in this document.

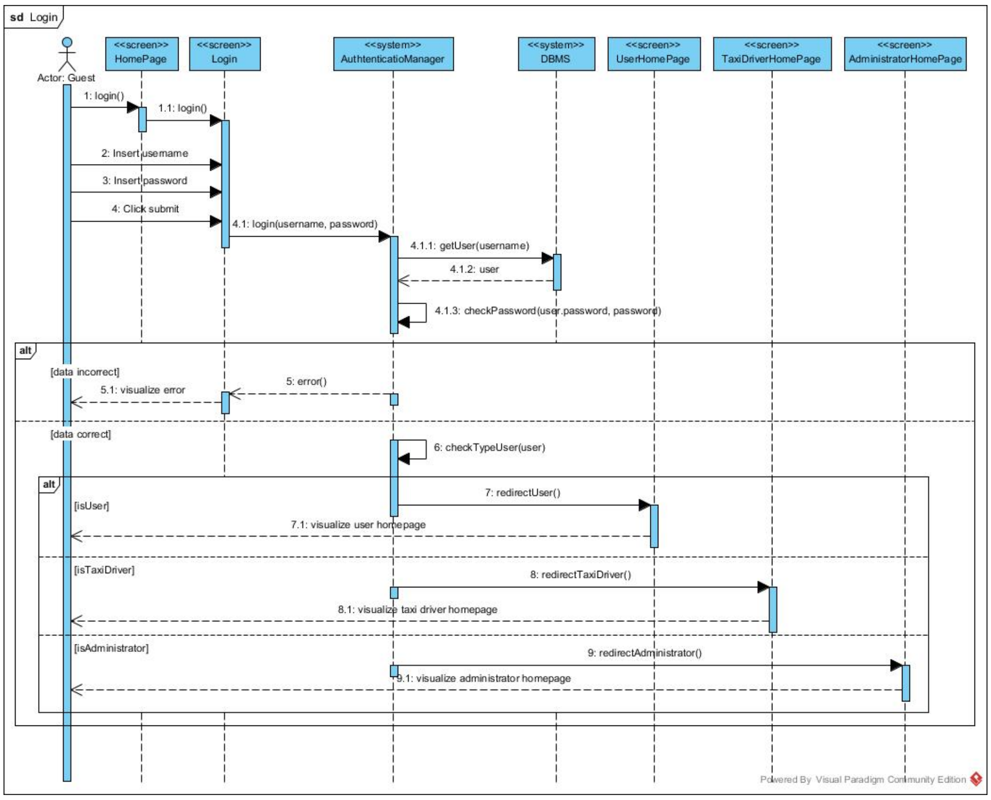


2.5 Runtime View

We'll now introduce the sequence diagrams, based on the use cases presented in the RASD, with a particular focus on the interaction between the components of the system.

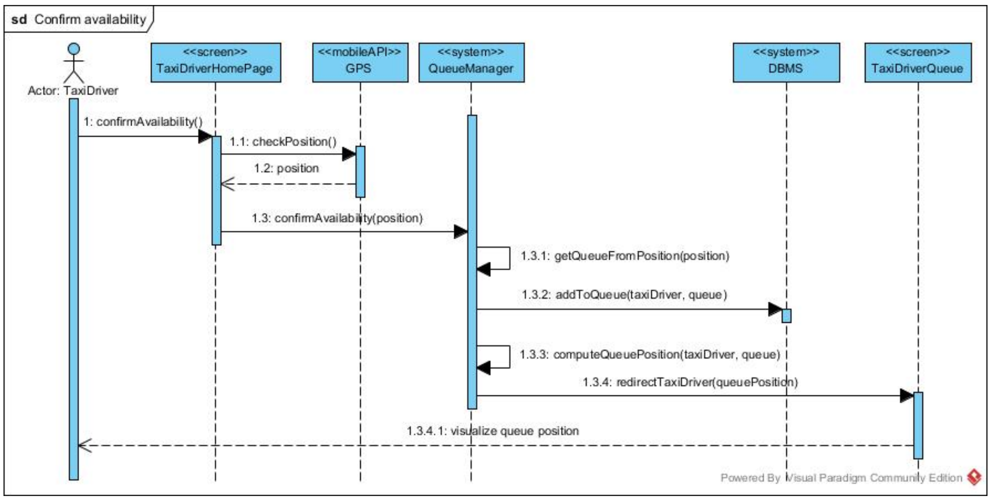
2.5.1 Login

The user inserts the username and the password and sends the data to the "Account-Manager" calling the "login" method. Now the "Account Manager" gets from the "DBMS" the user relative to the username and checks if the password corresponds, if it is incorrect or the username is not present in the database, it returns an error message to the user, otherwise checks the account type and redirects to the correct home page.



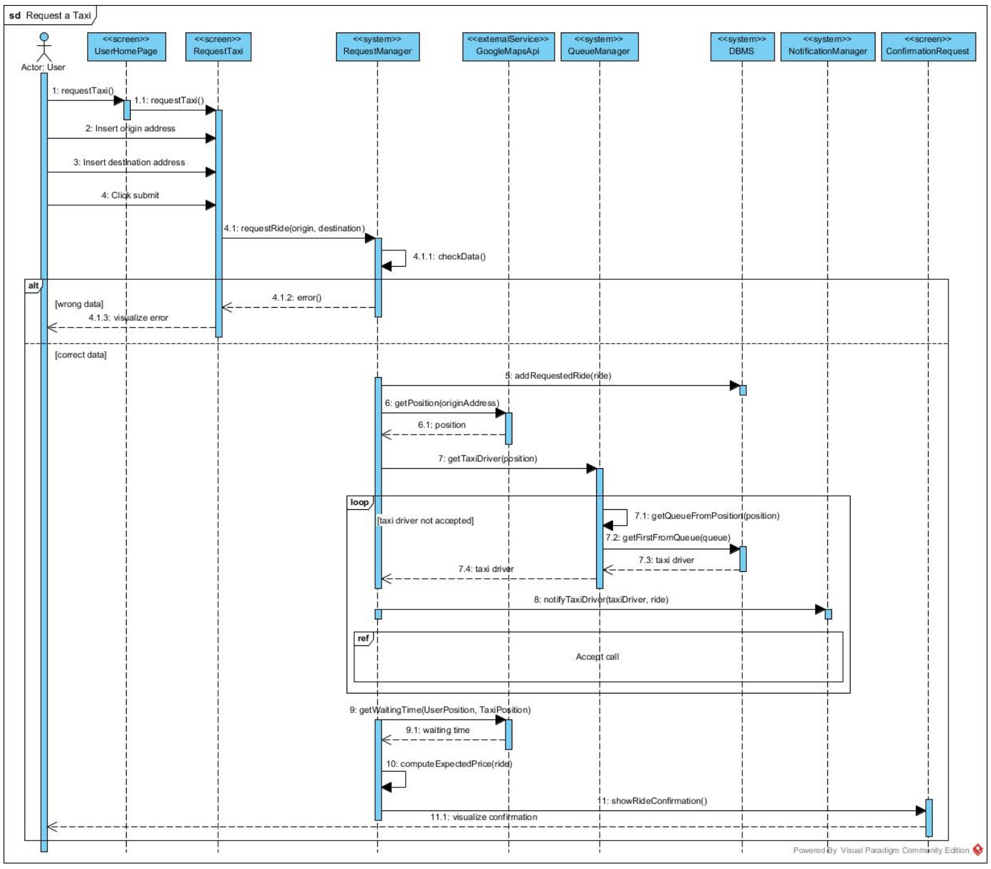
2.5.2 Confirm Availability

The taxi driver confirms his availability from the "TaxiDriverHomePage", that checks the position through the GPS and sends it to "QueueManager" through "confirmAvailability" method. Then the "QueueManager" identifies the city zone relative to the position and sends both to "DBMS" that inserts the couple in the database. The taxi driver is now redirected to the "TaxiDriverQueue" that visualizes his position in the queue.


sd Confirm availability
Actor: TaxiDriver
<<screen>> TaxiDriverHomePage
<<mobileAPI>> GPS
<<system>> QueueManager
<<system>> DBMS
<<screen>> TaxiDriverQueue
1: confirmAvailability()
1.1: checkPosition()
1.2: position
1.3: confirmAvailability(position)
1.3.1: getQueueFromPosition(position)
1.3.2: addToQueue(taxiDriver, queue)
1.3.3: computeQueuePosition(taxiDriver, queue)
1.3.4: redirectTaxiDriver(queuePosition)
1.3.4.1: visualize queue position
Powered By Visual Paradigm Community Edition

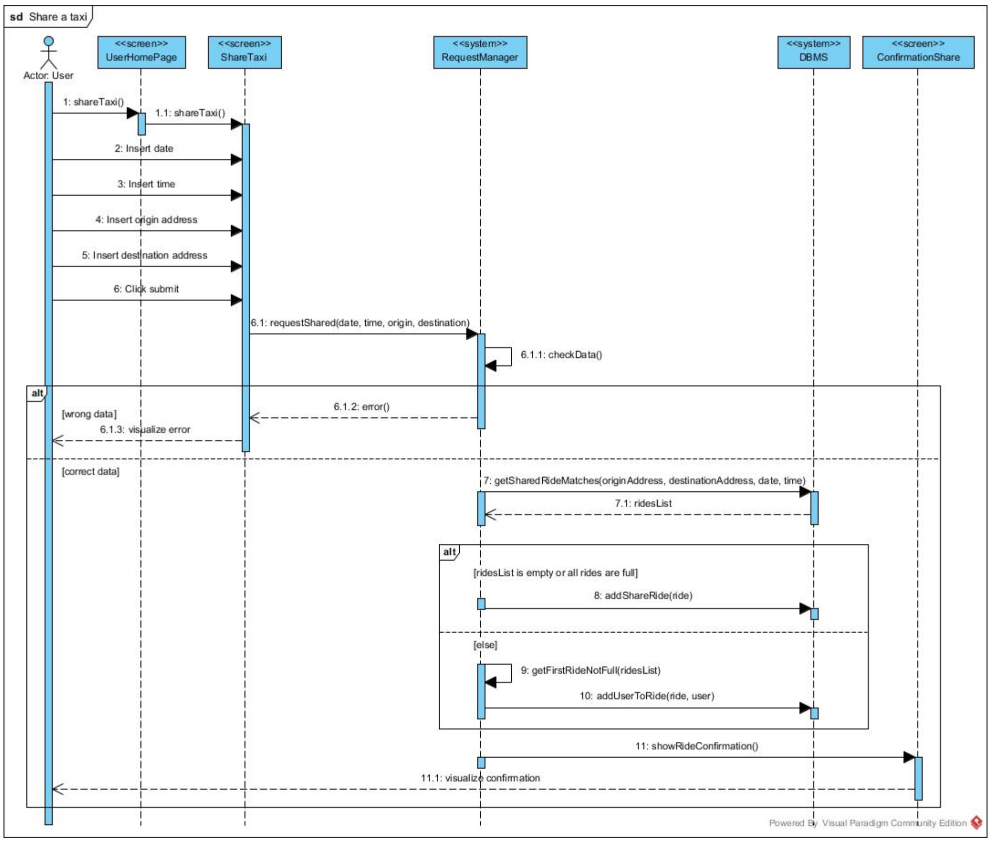
2.5.3 Request a Taxi

The user inserts the origin and the destination address of the ride and sends them to the “RequestManager”, that checks the data and if there is an error it shows it to the user. Otherwise it creates a new ride in the database, it transforms the origin address from string to position through “GoogleMapsAPI” and it searches an available taxi driver through the “QueueManager”. The “QueueManager” gets the first taxi driver from the queue relative to the given position and sends a notification through “NotificationManager” to the taxi driver. If he accepts the call the process continues, otherwise the “Queue Manager” looks for the second taxi driver until he accepts. In the end the “RequestManager” calculates the waiting time, the expected price and redirects the user to the “ConfirmationRequest”.


sd Request a Taxi
Actor: User
<<screen>> UserHomePage
<<screen>> RequestTaxi
<<system>> RequestManager
<<externalService>> GoogleMapsApi
<<system>> QueueManager
<<system>> DBMS
<<system>> NotificationManager
<<screen>> ConfirmationRequest
1: requestTaxi()
1.1: requestTaxi()
2: Insert origin address
3: Insert destination address
4: Click submit
4.1: requestRide(origin, destination)
4.1.1: checkData()
alt
[wrong data]
4.1.2: error()
4.1.3: visualize error
[correct data]
5: addRequestedRide(ride)
6: getPosition(originAddress)
6.1: position
7: getTaxiDriver(position)
loop
[taxi driver not accepted]
7.1: getQueueFromPosition(position)
7.2: getFirstFromQueue(queue)
7.3: taxi driver
7.4: taxi driver
8: notifyTaxiDriver(taxiDriver, ride)
ref
Accept call
9: getWaitingTime(UserPosition, TaxiPosition)
9.1: waiting time
10: computeExpectedPrice(ride)
11: showRideConfirmation()
11.1: visualize confirmation
Powered By Visual Paradigm Community Edition

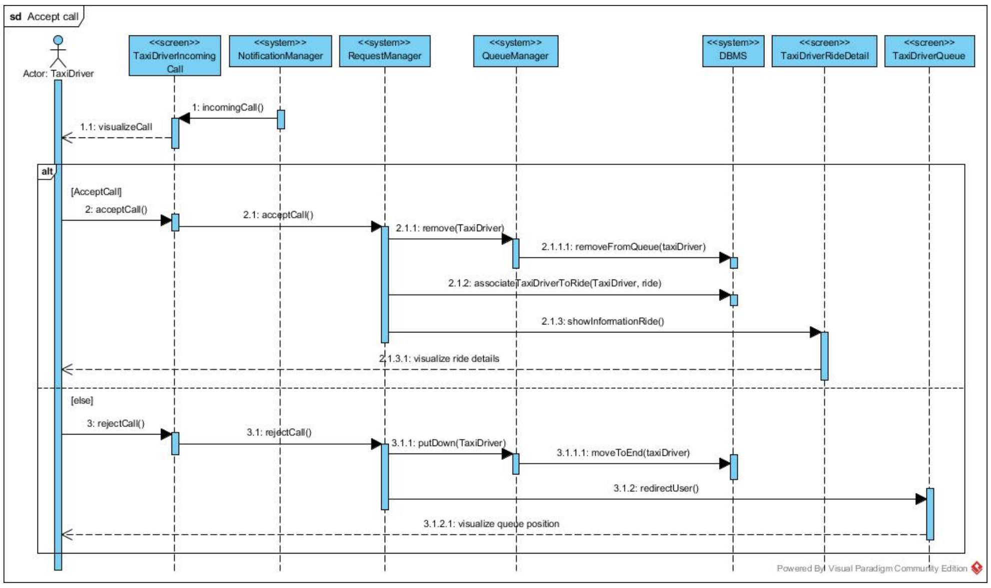
2.5.4 Share a Taxi

The user inserts the date, the time, the origin and the destination address of the ride and sends them to the "RequestManager", that checks the data and if there is an error it shows it to the user. Otherwise it gets a list of similar shared ride from the database. If the list is empty or all rides are full, it inserts a new shared ride in the databse, otherwise it adds the user to the ride. In the end it redirects the user to the confirmation page.


sd Share a taxi
Actor: User
<<screen>> UserHomePage
<<screen>> ShareTaxi
<<system>> RequestManager
<<system>> DBMS
<<screen>> ConfirmationShare
1: shareTaxi()
1.1: shareTaxi()
2: Insert date
3: Insert time
4: Insert origin address
5: Insert destination address
6: Click submit
6.1: requestShared(date, time, origin, destination)
6.1.1: checkData()
alt
[wrong data]
6.1.2: error()
6.1.3: visualize error
[correct data]
7: getSharedRideMatches(originAddress, destinationAddress, date, time)
7.1: ridesList
alt
[ridesList is empty or all rides are full]
8: addShareRide(ride)
[else]
9: getFirstRideNotFull(ridesList)
10: addUserToRide(ride, user)
11: showRideConfirmation()
11.1: visualize confirmation
Powered By Visual Paradigm Community Edition

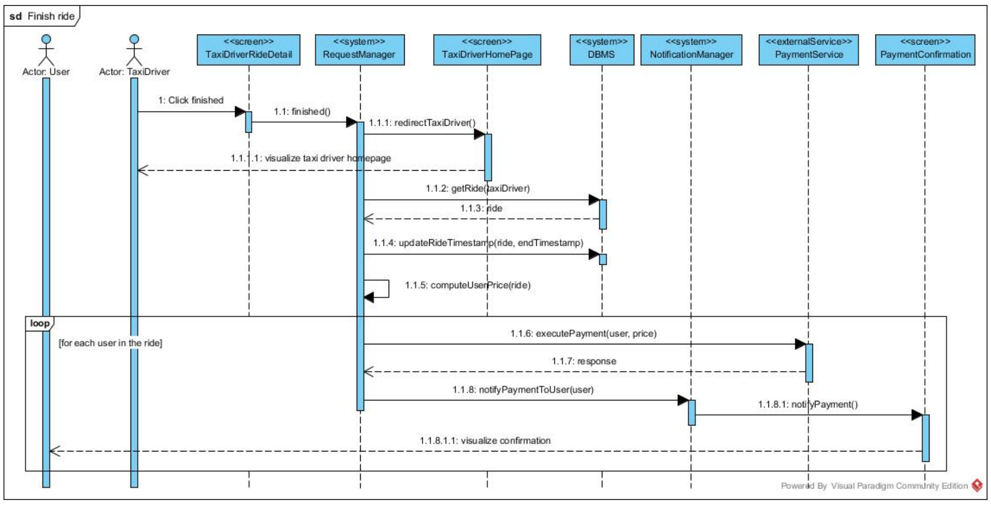
2.5.5 Accept Call

The taxi driver receives a notification from the “NotificationManager”. If the taxi driver accepts the call, the “RequestManager” says to the “QueueManager” to remove it from the correspondent queue. The taxi driver is now associated to the ride and is redirected to the screen with information about the ride. Otherwise the “Request-Manager” says to the “QueueManager” to put the taxi driver to the end of the queue and redirects the taxi driver to the “TaxiDriverQueue” screen.


sd Accept call
Actor: TaxiDriver
<<screen>> TaxiDriverIncomingCall
<<system>> NotificationManager
<<system>> RequestManager
<<system>> QueueManager
<<system>> DBMS
<<screen>> TaxiDriverRideDetail
<<screen>> TaxiDriverQueue
1: incomingCall()
1.1: visualizeCall
alt
[AcceptCall]
2: acceptCall()
2.1: acceptCall()
2.1.1: remove(TaxiDriver)
2.1.1.1: removeFromQueue(taxiDriver)
2.1.2: associateTaxiDriverToRide(TaxiDriver, ride)
2.1.3: showInformationRide()
2.1.3.1: visualize ride details
[else]
3: rejectCall()
3.1: rejectCall()
3.1.1: putDown(TaxiDriver)
3.1.1.1: moveToEnd(taxiDriver)
3.1.2: redirectUser()
3.1.2.1: visualize queue position
Powered By Visual Paradigm Community Edition

2.5.6 Finish Ride

The taxi driver clicks on the “finished” button and says to the “RequestManager” that the ride is over. The “RequestManager” redirects the taxi driver to his home page, updates the timestamp relative to the end of the ride and computes the price for all the passengers and sends them a notification of the payment.


sd Finish ride
Actor: User
Actor: TaxiDriver
<<screen>> TaxiDriverRideDetail
<<system>> RequestManager
<<screen>> TaxiDriverHomePage
<<system>> DBMS
<<system>> NotificationManager
<<externalService>> PaymentService
<<screen>> PaymentConfirmation
1: Click finished
1.1: finished()
1.1.1: redirectTaxiDriver()
1.1.1.1: visualize taxi driver homepage
1.1.2: getRide(taxiDriver)
1.1.3: ride
1.1.4: updateRideTimestamp(ride, endTimestamp)
1.1.5: computeUserPrice(ride)
loop
[for each user in the ride]
1.1.6: executePayment(user, price)
1.1.7: response
1.1.8: notifyPaymentToUser(user)
1.1.8.1: notifyPayment()
1.1.8.1.1: visualize confirmation
Powered By Visual Paradigm Community Edition

2.6 Component Interfaces

- DB
  - +addToQueue(TaxiDriver taxiDriver, Queue queue)
  - +removeFromQueue(TaxiDriver taxiDrive)
  - +getFirstFromQueue(Queue queue : TaxiDriver taxiDriver)

  - +moveToEnd(TaxiDriver taxiDriver)
  - +getRide(TaxiDriver taxiDriver : Ride ride)
  - +addReservedRide(Ride ride)
  - +addRequestedRide(Ride ride)
  - +addSharedRide(Ride ride)
  - +addUserToRide(Ride ride, User user)
  - +associateTaxiDriverToRide(TaxiDriver taxiDriver, Ride ride)
  - +updateRideTimestamp(Ride ride, Timestamp endTimestamp)
  - +getSharedRideMatches(String originAddress, String destinationAddress, Date date, String time : List¡Ride¿ rideList)
  - +removeRide(Ride ride)
  - +getUser(String username: User user)
  - +addUser(User user)
  - +updateUser(User user)
  - +removeUser(User user)
  - +getTaxiDriver(TaxiDriver taxiDriver)
  - +addTaxiDriver(TaxiDriver taxiDriver)
  - +removeTaxiDriver(TaxiDriver taxiDriver)
  - +updateTaxiDriver(TaxiDriver taxiDriver)
  - +getAdministrator(Administrator administrator)
  - +addAdministrator(Administrator administrator)

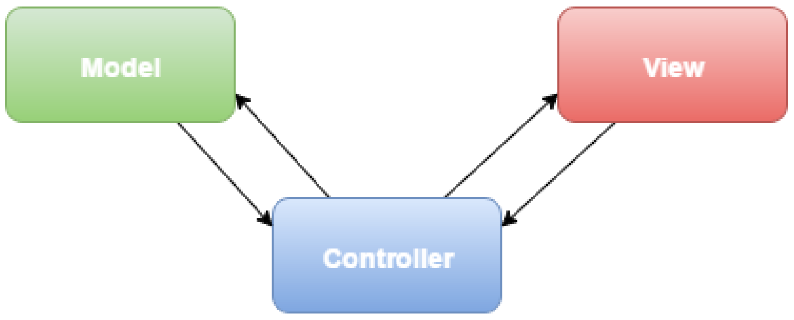
- TaxiQueues
  - +remove(TaxiDriver taxiDriver)
  - +putDown(TaxiDriver taxiDriver)
  - +getTaxiDriver(String position : TaxiDriver taxiDriver)
- Confirm Availability
  - +confirmAvailability(String position)
- NotificationListener
  - +incomingCall(Ride ride)
  - +notifyPayment(Double price)
- SendNotification
  - +notifyPaymentToUser(User user)
  - +notifyTaxiDriver(TaxiDriver taxiDriver, Ride ride)
- TaxiRequest
  - +acceptCall(Ride ride)
  - +rejectCall(Ride ride)
  - +finised(Ride ride)

- Request
  - +requestRide(String originAddress, String destinationAddress : String response)
  - +requestShared(Date date, String time, String originAddress, String destinationAddress : String response)
- Session
  - +login(String username, String password)
  - +createUserAccount(String username, String password, String name, String surname, String email, String creditCardNumber, String cardExpirationDate, String cardSecurityNumber)
  - +createTaxiDriverAccount(String username, String password, String name, String surname, String email, Image drivingLicensePhoto, String drivingLicenseID, Image, taxiLicensePhoto, String taxiLicenseID, String plateID, Image profilePicture)
- Payment API
  - +executePayment(User user, Double price : String response)
- Google Maps API
  - +getPosition(String address : String position)
  - +getWaitingTime(String position1, String position2 : Double waitingTime)

2.7 Architectural Styles

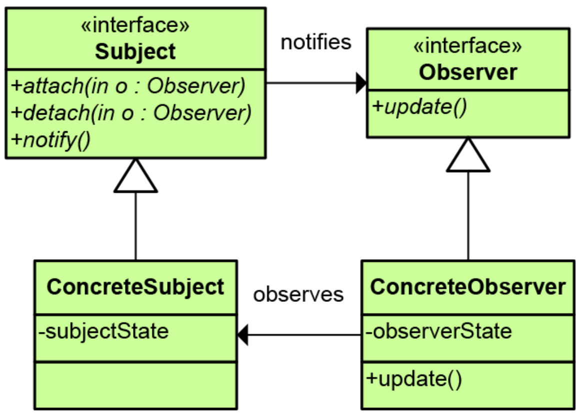
MyTaxiService system will be designed following a client-server 3 tier architectural style. In particular, it will consist in a thin client, which will strongly depend on the server. In fact, it will only consist of a User Interface which will offer to the client the possibility to send requests to the application server and to visualize in a user-friendly way the corresponding responses.

We also followed the MVC approach to separate the data model, the view of corresponding data and the controller which sends commands to the model to update its state and to the view to change its presentation of data. The model will be represented by the persistence tier, which will respond to the controller's commands and will be displayed in the view. The view will be represented by the client's user interface and it will display changes in the presentation of data according to the changes in the model. The controller will be represented by the components of the business logic tier, such as the Request Manager, the Queue Manager, the Account Manager and the Notification Manager.

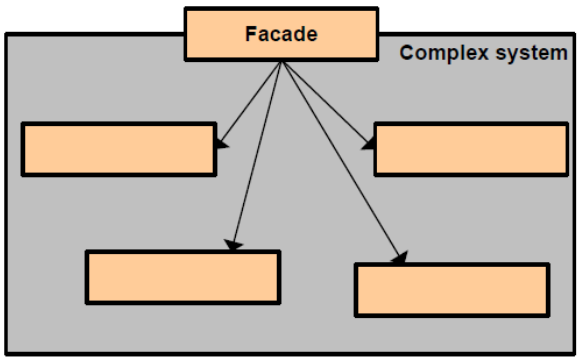


2.8 Architectural Patterns

We used the Observer pattern to design the Taxi Driver and User notification listener.



We also used The Facade pattern to handle the communication between different components. In fact, every communication passes through a defined interface in order to reduce coupling of the components and to increase the level of abstraction and reusability. In this way, future updates of a component will not affect other components if the interface doesn't change.



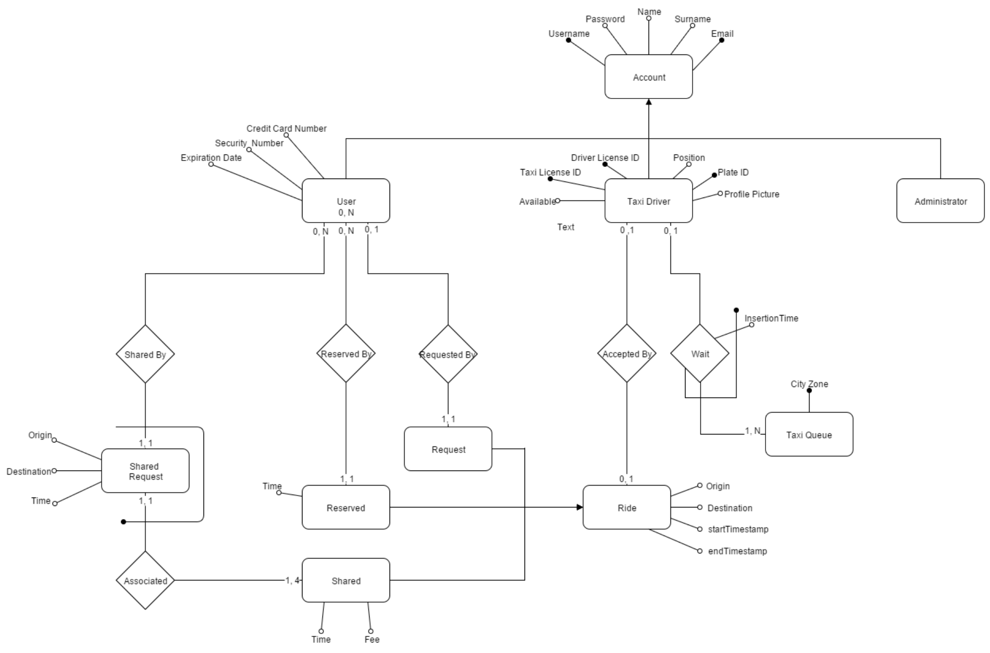
3 Persistent Data Design

In this section it will be designed the data layer of the system, starting from the class diagram defined in the Requirements Analysis and Specification Document and producing the definitions of the tables that will be implemented directly in the selected relational Database Management System.

3.1 Conceptual Design

3.1.1 ER Schema

The following Entity-Relationship schema is directly derived from the class diagram defined in the RASD.



The list of taxis in the queue is represented by the "Wait" relationship. The list is then sorted according to the "InsertionTime" attribute, that represents the exact time of the insertion of the taxi driver in the "Wait" relationship. In this way, the oldest taxi driver in a queue will have the minimum value of the "InsertionTime" attribute, while the last inserted one will have the greatest timestamp.

3.1.2 Data Glossary

In the following data glossary are defined all the entities and relationships with correspondent attributes and identifiers, and for each element is given a brief description.

ENTITY	DESCRIPTION	ATTRIBUTES	IDENTIFIERS
Account	Abstract entity that represents a general account in the system	• Username • Password • Name • Surname • Email	• Username • Email
User	Account owned by a user	• Credit Card Number • Security Number • Expiration Date	
Taxi Driver	Account owned by a taxi driver	• Taxi License ID • Driver License ID • Plate ID • Profile Picture • Position • Available	• Taxi License ID • Driver License ID • Plate ID
Administrator	Account owned by an administrator		
Taxi Queue	Taxi queue associated with a city zone, where available taxis are inserted	• City Zone	• City Zone

Ride	Abstract entity that represents a general ride requested by a user	• Origin • Destination	
Request	Normal ride requested by a user		
Reserved	Reservation of a ride	• Time	
Shared	Ride shared among one or more users	• Time • Fee	
Shared Request	Request for a shared ride	• Origin • Destination • Time	

RELATIOHNSHIP	**DESCRIPTION**	**ROLES**	**ATTRIBUTES**
Wait	Available taxi drivers waiting in a taxi queue	• Taxi Driver • Taxi Queue	• Insertion Time
Accepted By	Acceptance of a ride by a taxi driver	• Taxi Driver • Ride	
Requested By	Request of a normal ride by a user	• User • Request	
Reserved By	Request of a reserved ride by a user	• User • Reserved	
Shared By	Request of a shared ride by a user	• User • Shared Request	
Associated	Association of the shared ride requests and the actual shared ride	• Shared Request • Shared	

ATTRIBUTE	ENTITY/RELATIONS HIP	DOMAIN	DESCRIPTION
Username	Account	String	Username chosen by the user, unique in the system
Password	Account	String	Password chosen by the user
Name	Account	String	Name of the user
Surname	Account	String	Surname of the user
Email	Account	String	Email of the user, unique in the system
Credit Card Number	User	Integer	Credit Card Number of a registered user, used for payments
Security Number	User	Integer	Security Number of the Credit Card of the user
Expiration Date	User	Date	Expiration Date of the Credit Card of the user
Taxi License ID	Taxi Driver	Integer	The Taxi License ID of the taxi driver. This must be unique in the system
Driver License ID	Taxi Driver	Integer	The Driver License ID of the taxi driver. This must be unique in the system

Plate ID	Taxi Driver	String	The Plate ID of the car of the taxi driver. This must be unique in the system
Profile Picture	Taxi Driver	String	Picture of the taxi driver
Position	Taxi Driver	String	Current position of the taxi driver in the city
Available	Taxi Driver	Boolean	True if taxi driver is currently available, false otherwise
City Zone	Taxi Queue	Integer	The city zone to which the taxi queue is associated
Origin	Ride	String	Origin address of the ride
Destination	Ride	String	Destination Address of the ride
Start Timestamp	Ride	Timestamp	Time of the actual start of the ride
End Timestamp	Ride	Timestamp	Time of the actual end of the ride
Time	Reserved	Timestamp	The time of the reservation. It represents when the ride should start
Time	Shared	Timestamp	The time of the shared ride. It represents when the ride should start

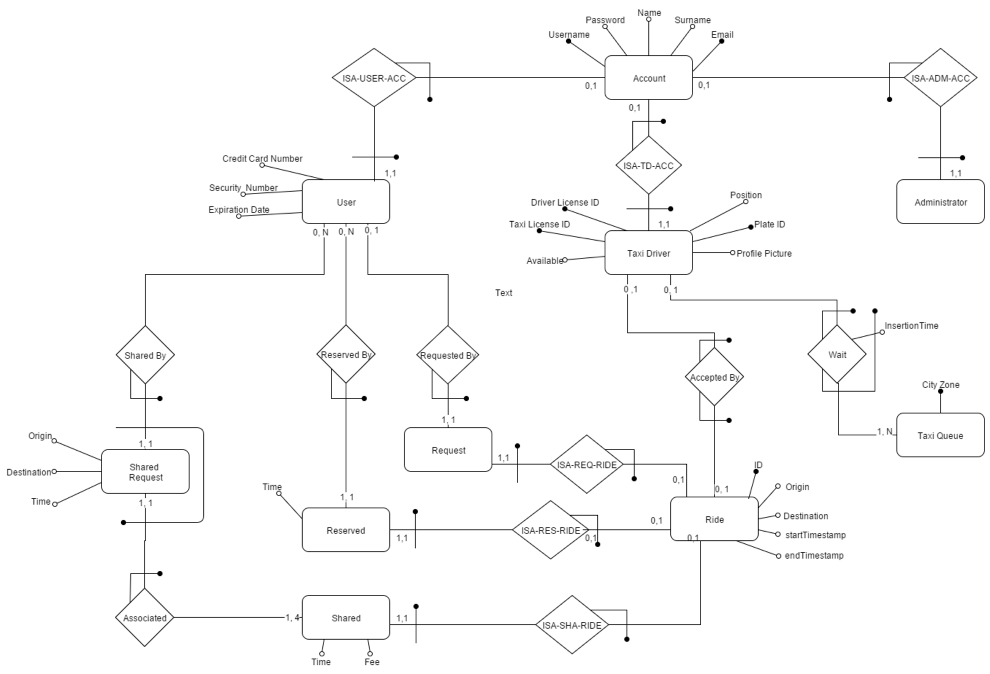
Fee	Shared	Double	The fee for the user in a shared ride. It's a part of the total cost
Origin	Shared Request	String	The origin address of the requested shared ride
Destination	Shared Request	String	The destination address of the requested shared ride
Time	Shared Request	Timestamp	The time of the requested shared ride. It represents when the ride should start
InsertionTime	Wait	Timestamp	The time when the taxi is inserted in the queue

3.1.3 Constraints

- **[C1]** A "Taxi Driver" cannot participate in both "Wait" and "Accepted By" relationships at the same time
- **[C2]** A "User" cannot reserve or share two different "Reserved" and/or "Shared Request" with the "Time" attribute values differing for 30 minutes or less
- **[C3]** A "User" cannot request a "Request" if there is a "Reserved" or "Shared" starting in less than 30 minutes or if there is a "Ride" not yet concluded (EndTimestamp == NULL)
- **[C4]** A "User" cannot reserve or share a ride if the difference between the "Time" attribute value and the current timestamp of the system is less than 2 hours

3.2 Logical Design

3.2.1 Restructured ER Schema



In this phase the ER schema has been restructured as a preparation to be translated in relational model. In fact all the constructs not directly translatable to the relational model have been eliminated. For instance, the two generalizations of "Account" and "Ride" have been replaced with relationships, adding also two constraints [C5] [C6] in order to maintain the disjoint and complete characteristics.

3.2.2 Constraints

- **[C1]** A "Taxi Driver" cannot participate in both "Wait" and "Accepted By" relationships at the same time
- **[C2]** A "User" cannot reserve or share two different "Reserved" and/or "Shared Request" with the "Time" attribute values differing for 30 minutes or less
- **[C3]** A "User" cannot request a "Request" if there is a "Reserved" or "Shared" starting in less than 30 minutes or if there is a "Ride" not yet concluded (End-Timestamp == NULL)
- **[C4]** A "User" cannot reserve or share a ride if the difference between the "Time" attribute value and the current timestamp of the system is less than 2 hours
- **[C5]** Every instance of "Account" participates in exactly one relationship among "ISA-USR-ACC", "ISA-TD-ACC" and "ISA-ADM-ACC"
- **[C6]** Every instance of "Ride" participates in exactly one relationship among "ISA-REQ-RIDE", "ISA-RES-RIDE" and "ISA-SHA-RIDE"

3.2.3 Translation in Relational Model

Now the restructured ER schema is translated to relational model producing the tables that will actually be in the physical database. For each table is given the primary key, and eventually all key, foreign key and inclusion constraints. In addition, the previous constraints expressed in the ER schema are adapted to the newly generated relational schema.

- Account(Username, Password, Name, Surname, Email)
  - key: Email
- User(AccountUsername, CreditCardNumber, SecurityNumber, ExpirationDate)
  - foreign key: User[AccountUsername] $\subseteq$ Account[Username]
- TaxiDriver(AccountUsername, DriverLicenseID, TaxiLicenseID, PlateID, ProfilePicture, Position, Available)
  - foreign key: TaxiDriver[AccountUsername] $\subseteq$ Account[Username]
  - key: DriverLicenseID
  - key: TaxiLicenseID
  - key: PlateID
- Administrator(AccountUsername)
  - foreign key: Administrator[AccountUsername] $\subseteq$ Account[Username]
- TaxiQueue(CityZone)
  - inclusion: TaxiQueue[CityZone] $\subseteq$ Wait[TaxiQueue]
- Ride(ID, Origin, Destination, startTimestamp*, endTimestamp*)
- Request(RideID)
  - foreign key: Request[RideID] $\subseteq$ Ride[ID]
  - foreign key: Request[RideID] $\subseteq$ RequestedBy[Request]
- Reserved(RideID, Time)
  - foreign key: Reserved[RideID] $\subseteq$ Ride[ID]
  - foreign key: Reserved[RideID] $\subseteq$ ReservedBy[Reserved]
- Shared(RideID, Time, Fee)
  - foreign key: Shared[RideID] $\subseteq$ Ride[ID]
  - inclusion: Shared[RideID] $\subseteq$ SharedRequest[Shared]
- SharedRequest(User, Shared, Origin, Destination, Time)
  - foreign key: SharedRequest[User] $\subseteq$ User[AccountUsername]
  - foreign key: SharedRequest[Shared] $\subseteq$ Shared[RideID]
- Wait(TaxiDriver, TaxiQueue, InsertionTime)
  - foreign key: Wait[TaxiDriver] $\subseteq$ TaxiDriver[AccountUsername]

  - foreign key: Wait[TaxiQueue] ⊆ TaxiQueue[CityZone]
  - key: TaxiQueue, InsertionTime

- AcceptedBy(<u>TaxiDriver</u>, RideID)
  - foreign key: AcceptedBy[TaxiDriver] ⊆ TaxiDriver[AccountUsername]
  - foreign key: AcceptedBy[RideID] ⊆ Ride[ID]
  - key: RideID

- RequestedBy(<u>Request</u>, User)
  - foreign key: RequestedBy[Request] ⊆ Request[RideID]
  - foreign key: RequestedBy[User] ⊆ User[AccountUsername]
  - key: User

- ReservedBy(<u>Reserved</u>, User)
  - foreign key: ReservedBy[Reserved] ⊆ Reserved[RideID]
  - foreign key: ReservedBy[User] ⊆ User[AccountUsername]

3.2.4 Relational Constraints

- **[C1]** Wait[TaxiDriver] ⋂ AcceptedBy[TaxiDriver] = Ø
- **[C2]** A "User" cannot reserve or share two different "Reserved" and/or "Shared Request" with the "Time" attribute values differing for 30 minutes or less
- **[C3]** A "User" cannot request a "Request" if there is a "Reserved" or "Shared" starting in less than 30 minutes or if there is a "Ride" not yet concluded (End-Timestamp == NULL)
- **[C4]** A "User" cannot reserve or share a ride if the difference between the "Time" attribute value and the current timestamp of the system is less than 2 hours
- **[C5.1]** User[AccountUsername] ⋂ TaxiDriver[AccountUsername] ⋂ Administrator[AccountUsername] = Ø
- **[C5.2]** Account[Username] = User[AccountUsername] ∪ TaxiDriver[AccountUsername] ∪ Administrator[AccountUsername]
- **[C6.1]** Request[ID] ⋂ Reserved[ID] ⋂ Shared[ID] = Ø
- **[C6.2]** Ride[ID] = Request[ID] ∪ Reserved[ID] ∪ Shared[ID]
- **[C7]** NOT EXIST SELECT * FROM SharedRequest GROUP BY Shared HAVING count(*) ¿ 4

4 Algorithm Design

This is a small list of the most significant algorithm used in the project.

4.1 Expected Price Calculation in Normal and Reserved Rides

This algorithm is used by the “Request manager” to calculate the expected price of the ride at the moment of the request. It used the Google Maps API for calculate the distance and an expected duration of the ride based on the origin and destination. It return a double variable containing the price.

```
double computeExpectedPrice(Ride ride)
   totalDistance ← getDistanceFromGMaps(ride.origin, ride.destination)
   expectedDuration ← getDurationFromGMaps(ride.origin, ride.destination)
   factor ← 2 // constant factor
   distanceFactor ← 0.5
   durationFactor ← 0.5
   price ← (totalDistance * distanceFactor + expectedDuration*durationFactor)
       * factor
   return price
```

4.2 Actual Price Calculation

It is invoked by “computeUserPrice” function, it calculate the total actual price of the ride.The total distance is calculate by a call to Google Maps API with origin and destination, the total duration is calculate by the difference between start and end timestamp. It uses a tariff of 50 cents per minute and another 50 cents per kilometer. It return a double variable containing the price.

```
double computeActualPrice(Ride ride)
   factor ← 2
   distanceFactor ← 0.5
   durationFactor ← 0.5
   totalDistance ← getDistanceFromGMaps(ride.origin, ride.destination)
   totalDuration ← computeDuration(ride.startTimestamp, ride.endTimestamp)
   price ← (totalDistance*distanceFactor +
       totalDuration*durationFactor)*factor
   return price
```

4.3 Fee Division in Shared Rides

It is invoked by “Request manager” for calculate the actual price of the ride. Simply it divide the total actual price of the ride, calling “computeActualPrice”, by the number of passengers of the ride. It return a double variable containing the price.

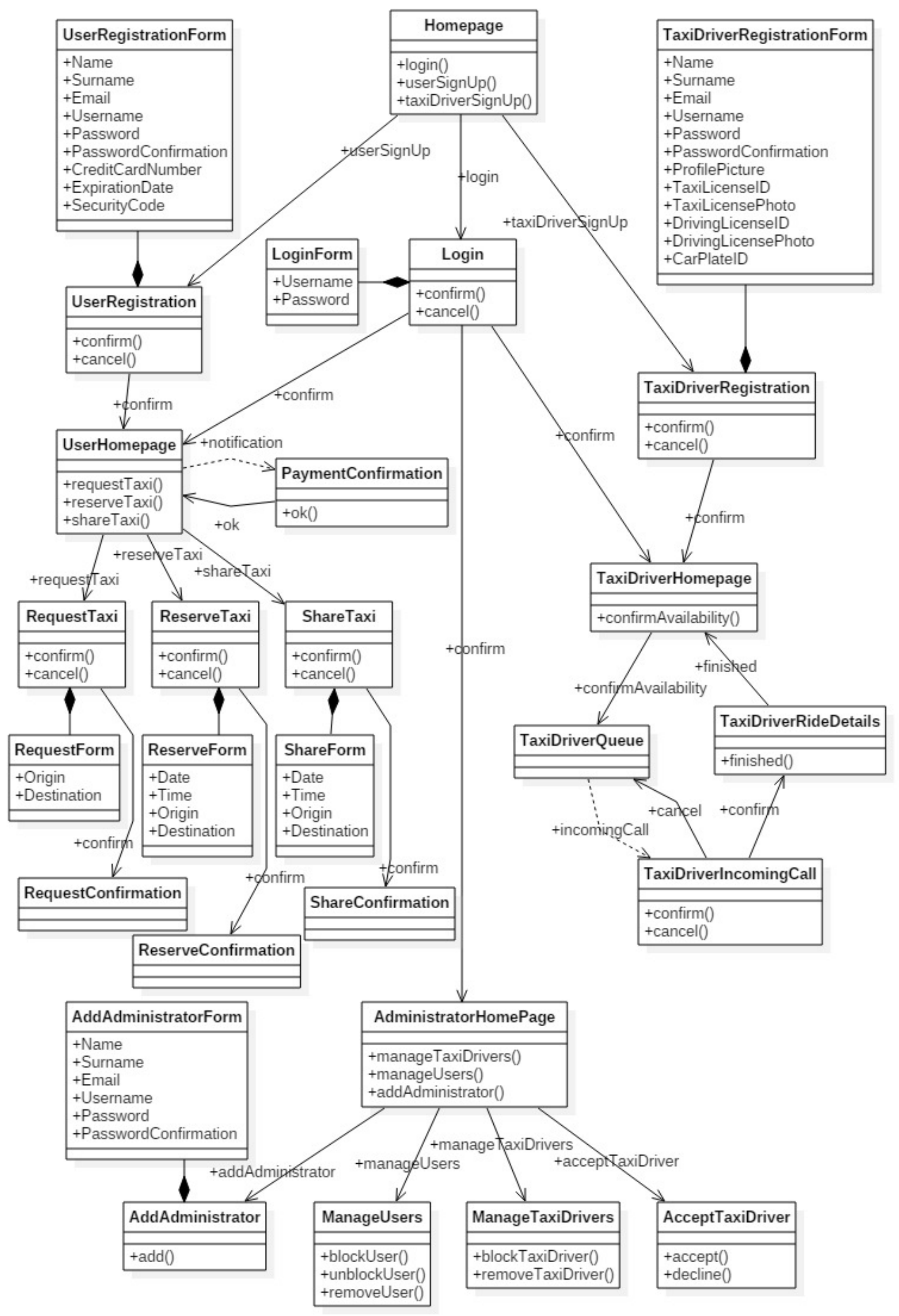
```
double computeActualPrice(Ride ride)
   factor ← 2
   distanceFactor ← 0.5
   durationFactor ← 0.5
   totalDistance ← getDistanceFromGMaps(ride.origin, ride.destination)
   totalDuration ← computeDuration(ride.startTimestamp, ride.endTimestamp)
   price ← (totalDistance*distanceFactor +
       totalDuration*durationFactor)*factor
```

```
    return price
```

5 User Interface Design

5.1 User Experience Diagram

This UX diagram shows the interaction between the different types of users and the system in terms of the possible sequences of visited pages. It describes all the possible navigations based on the actions in each page.



Mock-ups of the User Interface have been already presented in the RASD document.

6 Requirements Traceability

Requirements presented in the RASD will be managed by the components of the system and by the interfaces they offer.

- Requirements from [R1] to [R6] will be satisfied by the Account Manager component and its interfaces' methods.
- Requirements from [R7] to [R8] will be satisfied by the Request Manager component and its interfaces' methods.
- Requirements from [R9] to [R10] will be satisfied by the Notification Manager component and its interfaces' methods.
- Requirements from [R11] to [R16] will be satisfied by the Request Manager component and its interfaces' methods.
- Requirement [R17] will be satisfied by the Notification Manager component and its interfaces' methods.
- Requirements from [R18] to [R20] will be satisfied by the Queue Manager component and its interfaces' methods.

7 Annotations

7.1 Tools

- Visual Paradigm for Component, Deployment and Sequence diagrams
- Star UML for UX diagram
- Draw.io for ER Schema
- Overleaf for latex document

7.2 Times spent working on this document

Group Member	**Total Hours**
Andrea Donati	26
Gabriele Carassale	24
Manuel Deleo	25