POLITECNICO DI MILANO

Computer Science and Engineering

MyTaxiService

Design Document

Authors: Greta Ghiotti

Raffaele Malvermi

Mirco Mutti

Version 1.1

Date 13/01/2016

Reference Professor: Mirandola Raffaela

TABLE OF CONTENTS

Version1.1:

- **Addition to requirements traceability table (at pages 65, 66) of the corresponding rows for signUp TaxiDriver and Shared Fees.**

1) INTRODUCTION

1.1 Purpose

The following Design Document aims to point out, in a formal way, the structure of the System described in the RASD and the main architectural and design decisions that have to guide the development of the software.

For this reasons, the main recipients we have considered writing this DD are: the committee of MyTaxiService and the developers that will have to implement the application. We want to show to the committee how we intend to produce a System that satisfies the goals pointed out in the RASD document; we aim also to provide to the developers all the necessary information for the following implementation phase.

1.2 Scope

We have tried to create the Design Document in order to have an abstraction level sufficient to make clear the System design (avoiding ambiguity as much as possible) without introducing unnecessary constraints to the following implementation. Then we didn't mention in the document details as the dbms to-be-adopted and we have tried to make our design independent to the programming language that will be used.

1.3 Definitions, Acronyms, Abbreviations

We report the definition of certain expressions which we have used in the document:

- Final sharing route: this expression refer to the route computed at the receiving of the departure declaration from the MyTaxiDriver that is serving a sharing request; it is exactly the route that the Taxi Driver will follow during the sharing ride;
- Asynchronous component: we have decided to call "Asynchronous" the module of the Communication component (both Server and Client side) that manage interactions with asynchronous behavior;
- Synchronous component: we have decided to call "Synchronous" the module of the Communication component (both Server and Client side) that manage the interaction with synchronous behavior;

(This is only a supplement to the glossary reported in the 1.3 section of the RASD)

1.4 Reference Documents

We report the list of documents we have considered as landmarks for our DD and also the documents we have referred-to:

- MyTaxiService Requirements Analysis and Specification Document
- "IEEE Standard for Information Technology—Systems Design— Software Design Descriptions", IEEE Computer Society
- "Systems and software engineering — Architecture description", IEEE Computer Society
- "Design I-II" and "Software architectures and styles" sets of slide form Software Engineering 2 course
- www.uml-diagrams.org

1.5 Document Structure

In the chapter below, 2 - Architectural design, that is the most important in the document, we present a description of the architectural and design decisions from different level of abstraction.

In the following chapter (Algorithm design) we focus on the System algorithms; it's not our purpose to cover the entire set of algorithms that will be implemented in the application, but only to consider a set of critical and very significant algorithms that we have decided to describe in a detailed way.

Then, in the chapter 4 - User interface design we present an update of the corresponding RASD document section (3.2.1 User Interface).

In the following Requirements traceability chapter (5) we report a focus on the relation between each functional requirement included in the RASD and the set of System components that are involved in its satisfaction.

Then we end the DD including a chapter for the notable references (6 - References).

2) ARCHITECTURAL DESIGN

2.1 Overview

In the sections below we present the architectural design of our application.

The main landmark that we have followed to build our architecture is the “divide-et-impera” principle and, starting from this consideration, we have reduced our problem in many sub-problems: not “how to develop our application” anymore, but “how to develop the components that constitute our application and their interactions”. We didn’t forget to consider how our types of problems was been solved in the past and so we have taken into account the main architectural styles and patterns. In the component individuation process we have tried to make as high as possible the cohesion of each module and to reduce at the same time the coupling between different modules. Detailing the expected input and output of each function provided by modules we have also tried to make more testable our to-be-developed system.

First, focusing on the higher level description of our software system (2.2 High level components and their interaction), we present three unformal graphical illustration of the system architecture, increasing step by step the level of details.

In the following chapter (2.3 Component view) we describe in a more formal and extended way the system structure presented in the previous section using an UML component diagram. We also add a detailed description of each component included in the diagram.

Then, in the section 2.4 Deployment view, we present an UML deployment diagram together with a description of the main deployment choices and artifacts.

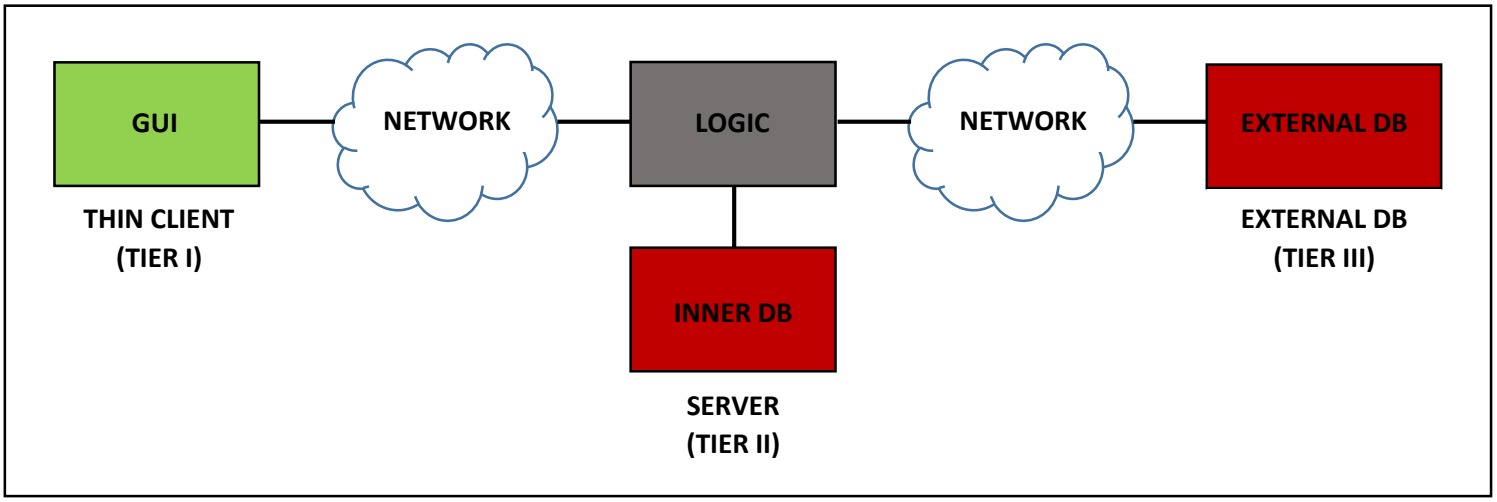
In the following (2.5 Runtime view) we explain the runtime behaviors of our application including a set of UML sequence diagrams; the purpose of this section is to extend the description of Use Cases presented in the Rasd document, previously released, from a more design-specific point of view: the matter it’s not the interactions between users and system anymore. In this section we focus on the interactions between the different application subsystems, components and modules.

In the chapter 2.6, Component interfaces, we present a very detailed description of each interface, provided by each component, listing the set of operations included in an interface. For each operation we report the expected input and output and a very brief functional description.

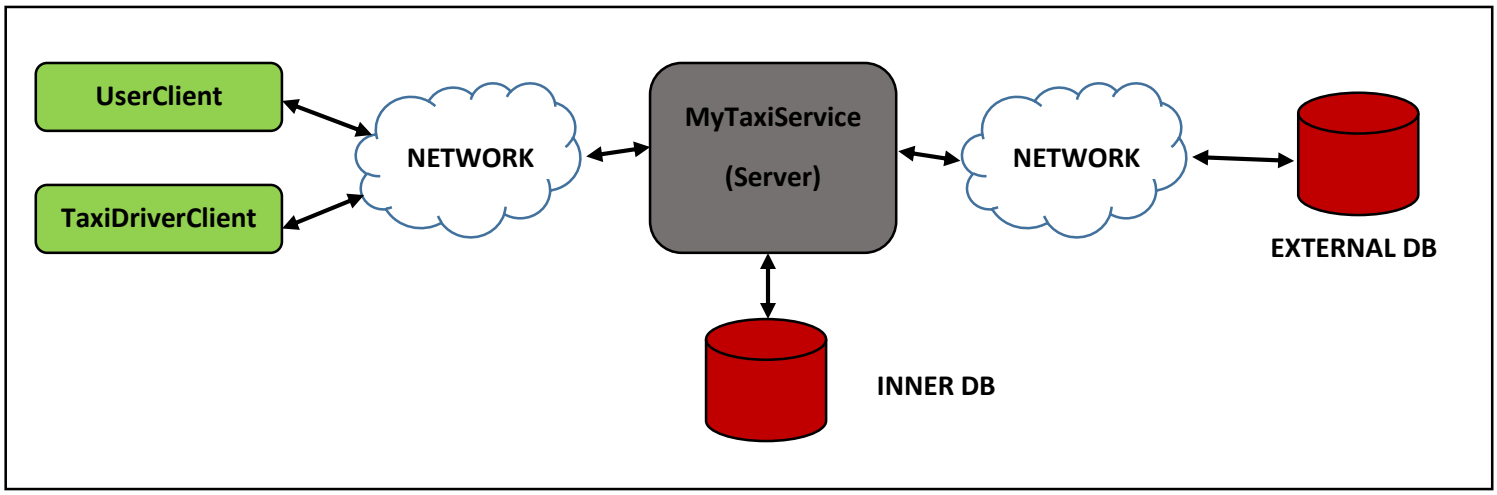
In the last two sections we finally include an explicit explanation of the architectural and design choices presented in the previous formalizations (in particular in the chapter 2.7

Architectural styles and patterns). The section 2.8, Other design decisions, reports a focus on the structure of the most critical and complex module: the Dispatcher component.

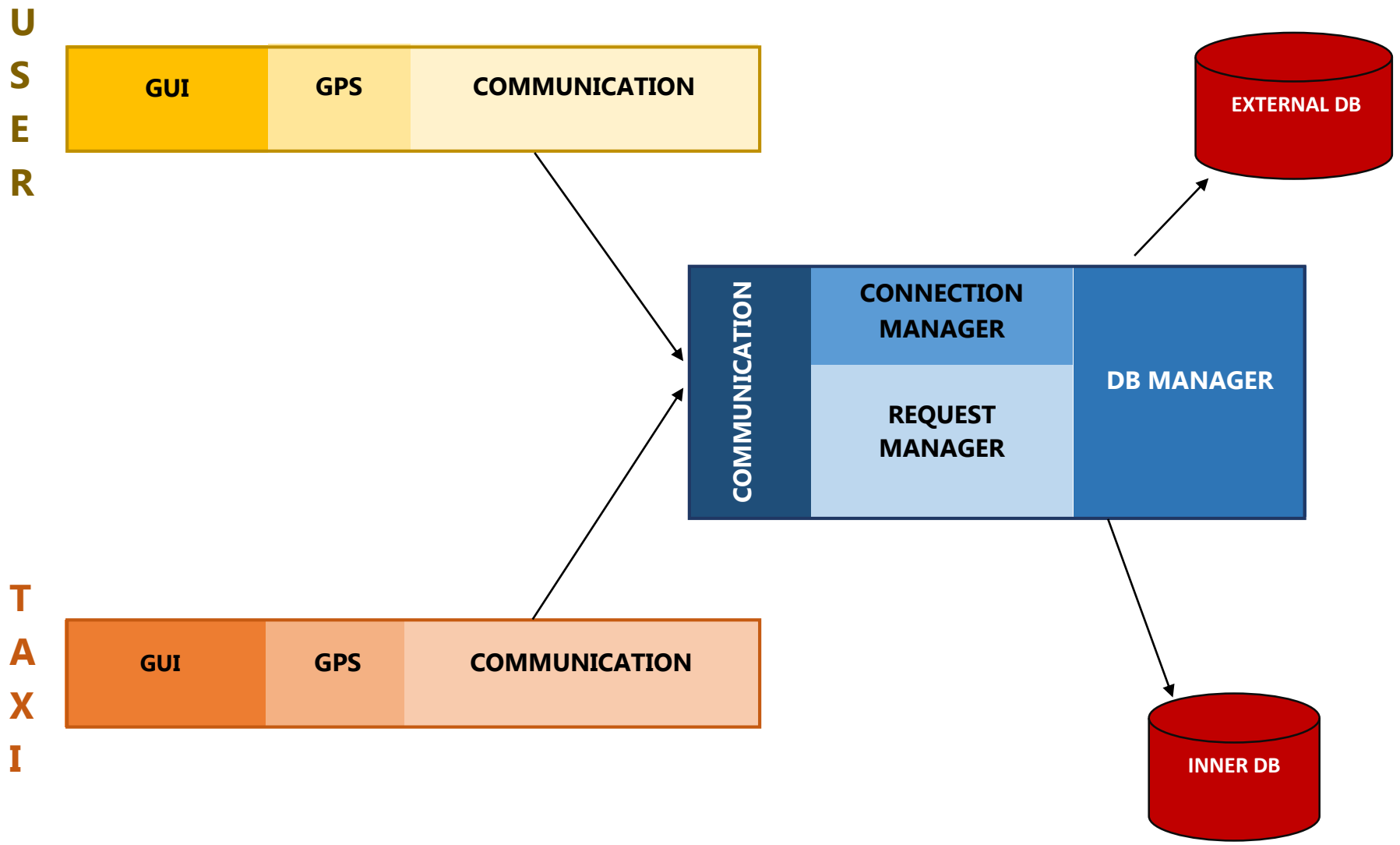
2.2 High level components and their interaction



In the first diagram we report the highest level description of our system in according to the 3-tiered architecture model.



The second diagram describes the transposition of the model previously presented in our system implementation.

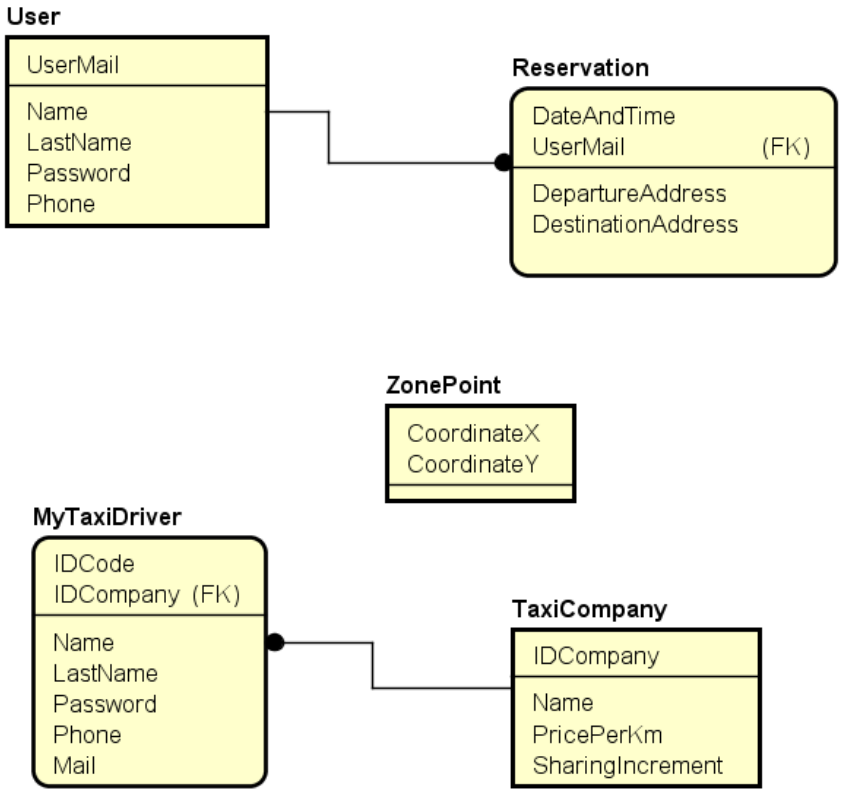


The last image would show the result of the “divide et impera” principle: the architecture is obtained by the interaction of some sub-systems, which in turn are shown by a composition of modules.

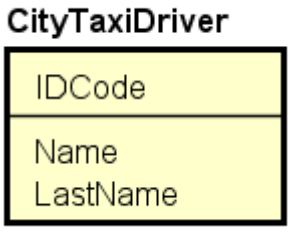
DATABASE

In our system we refer to two database one internal to the server and the other external to the system.

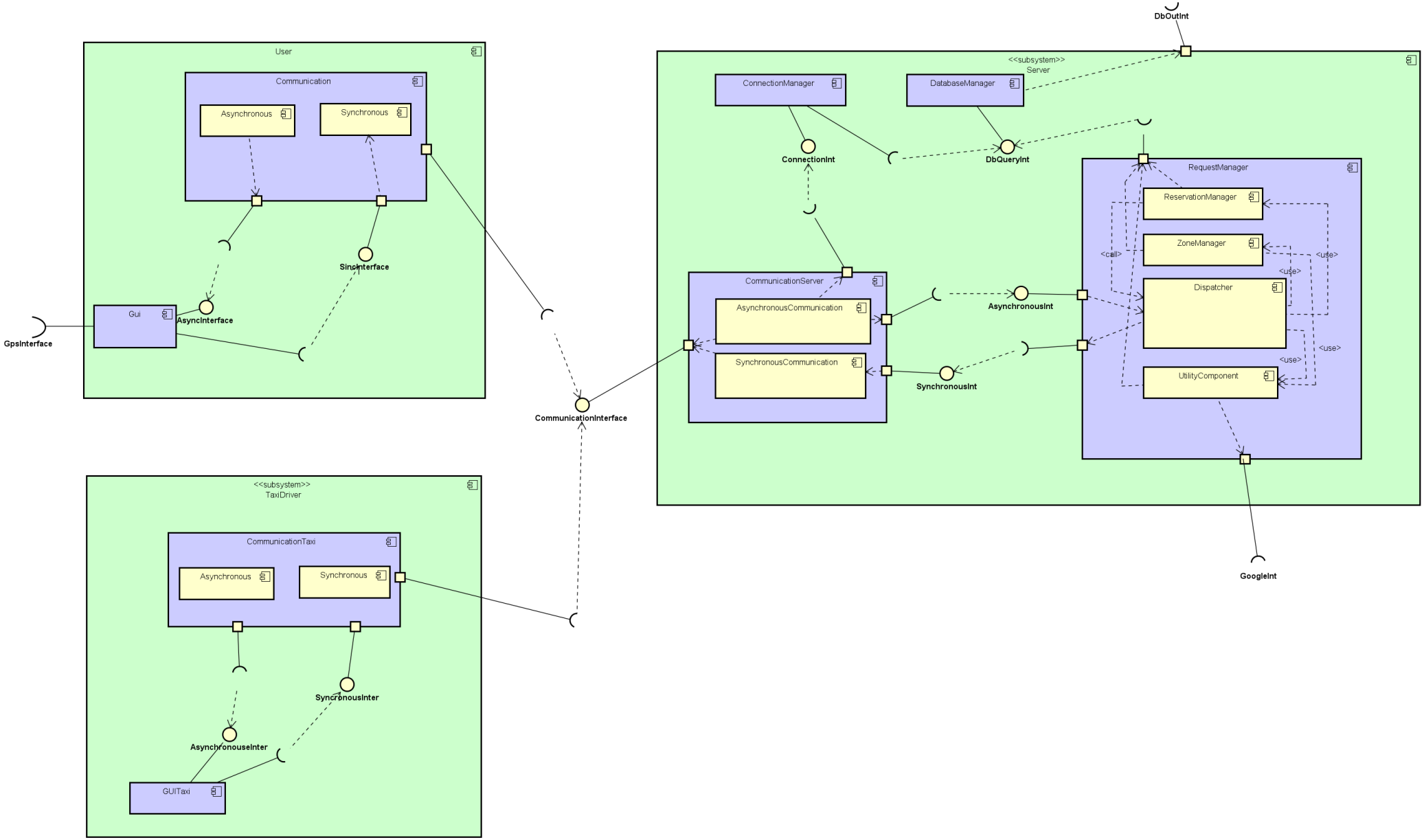
Now we present the ER model of our inner DataBase:



In the external DataBase we refer to the City Taxi driver Entity, naturally it is composed by other entities.



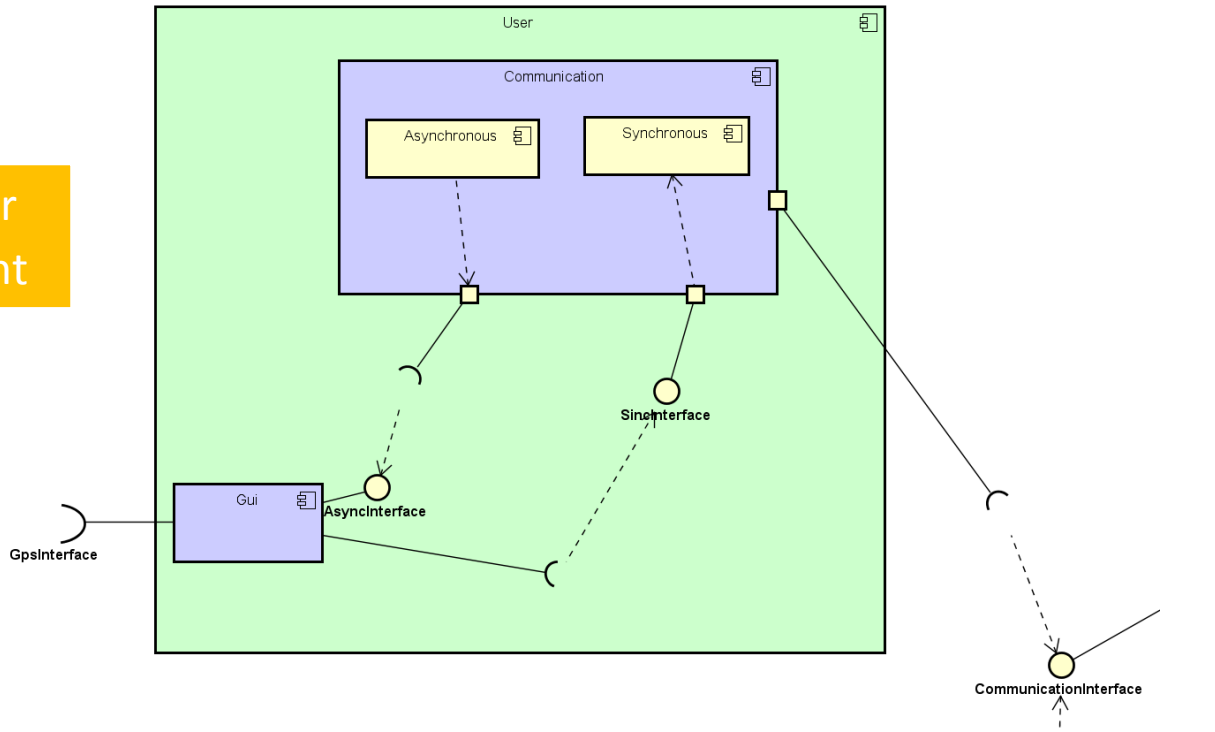
2.3 Component view



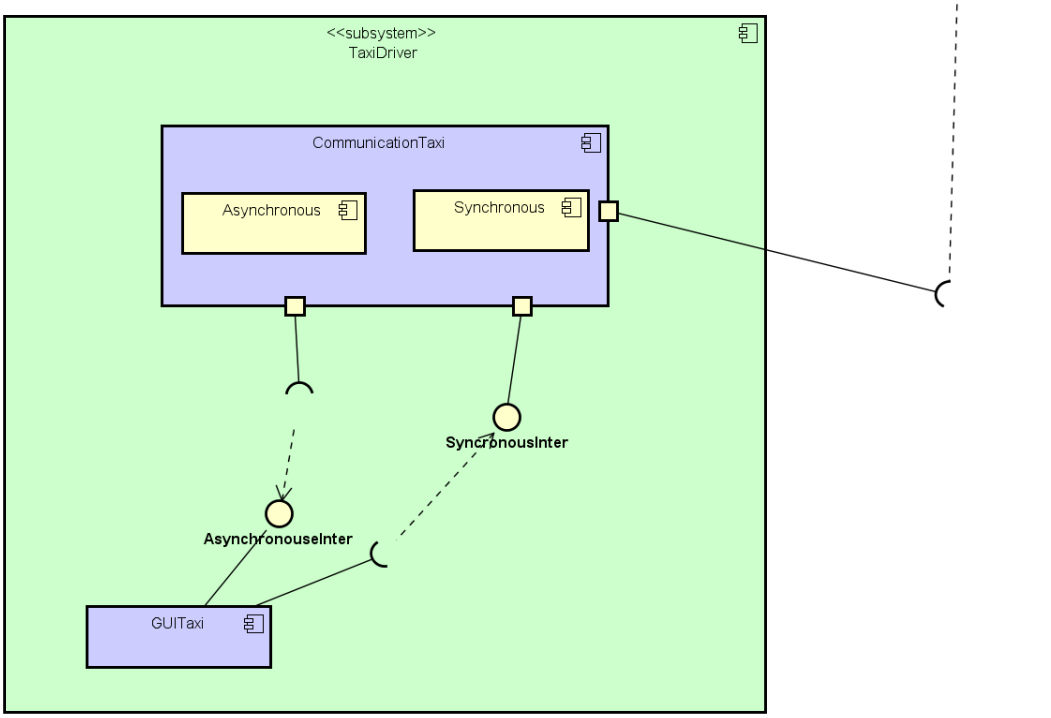
USER CLIENT

We describe the component of user client but the same consideration can be apply to taxi Driver client

User Client

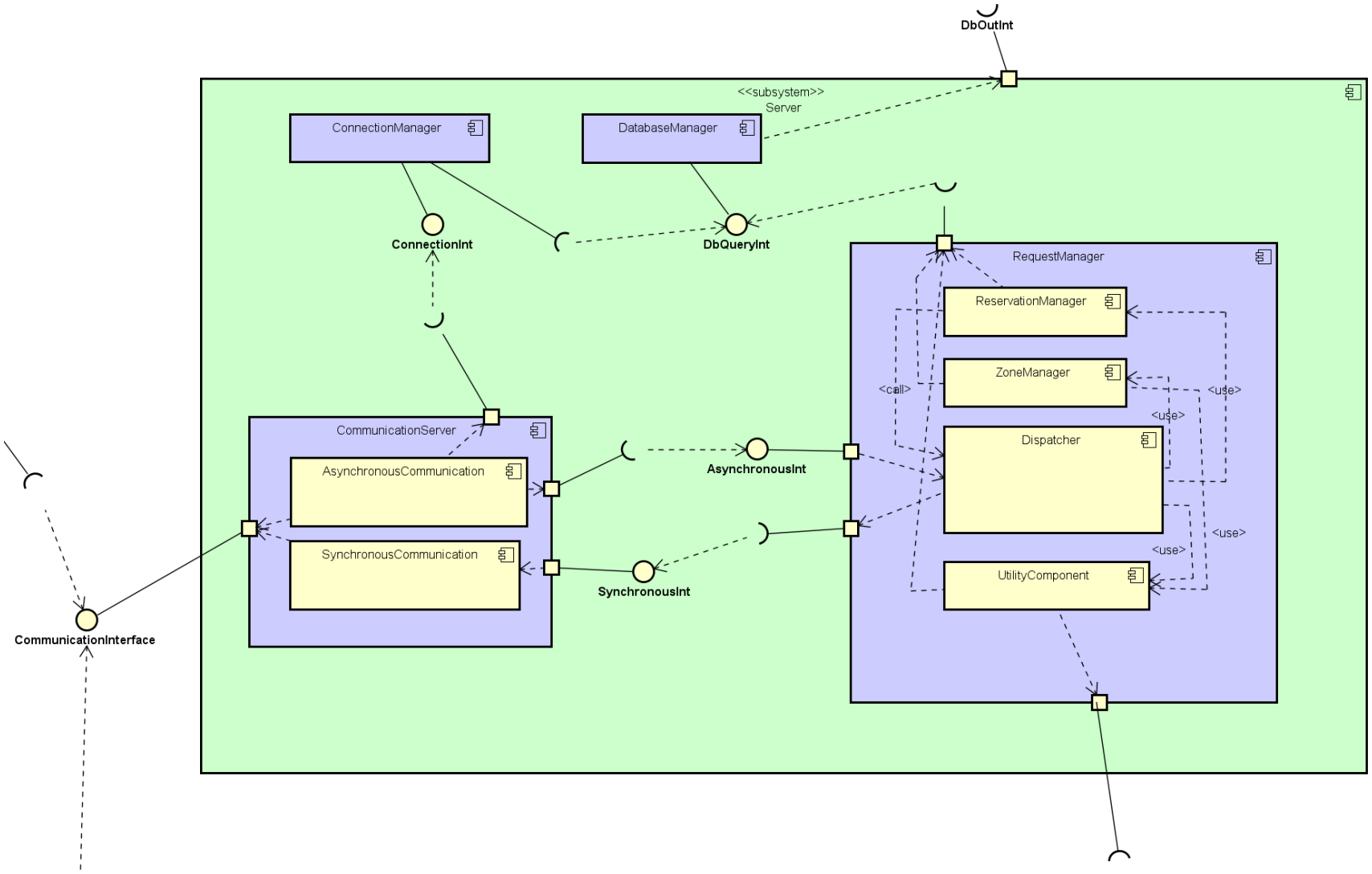


Taxi Driver Client



Component	Description	Interfaces
Gui	• Component that provides the User Interface service • It can be called by communication component, the asynchronous one, to show notification message. • It calls the synchronous component of communication to notify an action done by user/ taxi driver to the system • It required the gpsInterface to get the Gps Coordinate from the device	• AsynchronousInterface (provided) • SynchronousInterface (required) • GpsInterface (required)
Communication	• Module that manages the communication between client and server . • The communication can be asynchronous or synchronous so we have divided this element in two different components Asynchronous Communication and the other Synchronous Communication • This component required the CommunicationInterface provided by the communication component of the server to send and receive message from\by the server	• CommunicationInterface (required)
Asynchronous	• This module manages the asynchronous communication between Gui and CommunicationServer • An example is in the taxi Driver application when the system notifies a reservation to the TaxiDriver and it doesn't expects a answer.	• AsynchronousInterface (required)
Synchronous	• This module manages the synchronous communication from Gui to CommunicationServer • An example is when the user does a log in and she\he expects an answer of correct or wrong access from the system. So the Gui required logIn method to SynchronousInterface of communication that contacts the server to verify the correct authentication.	• SynchronousInterface (provided)

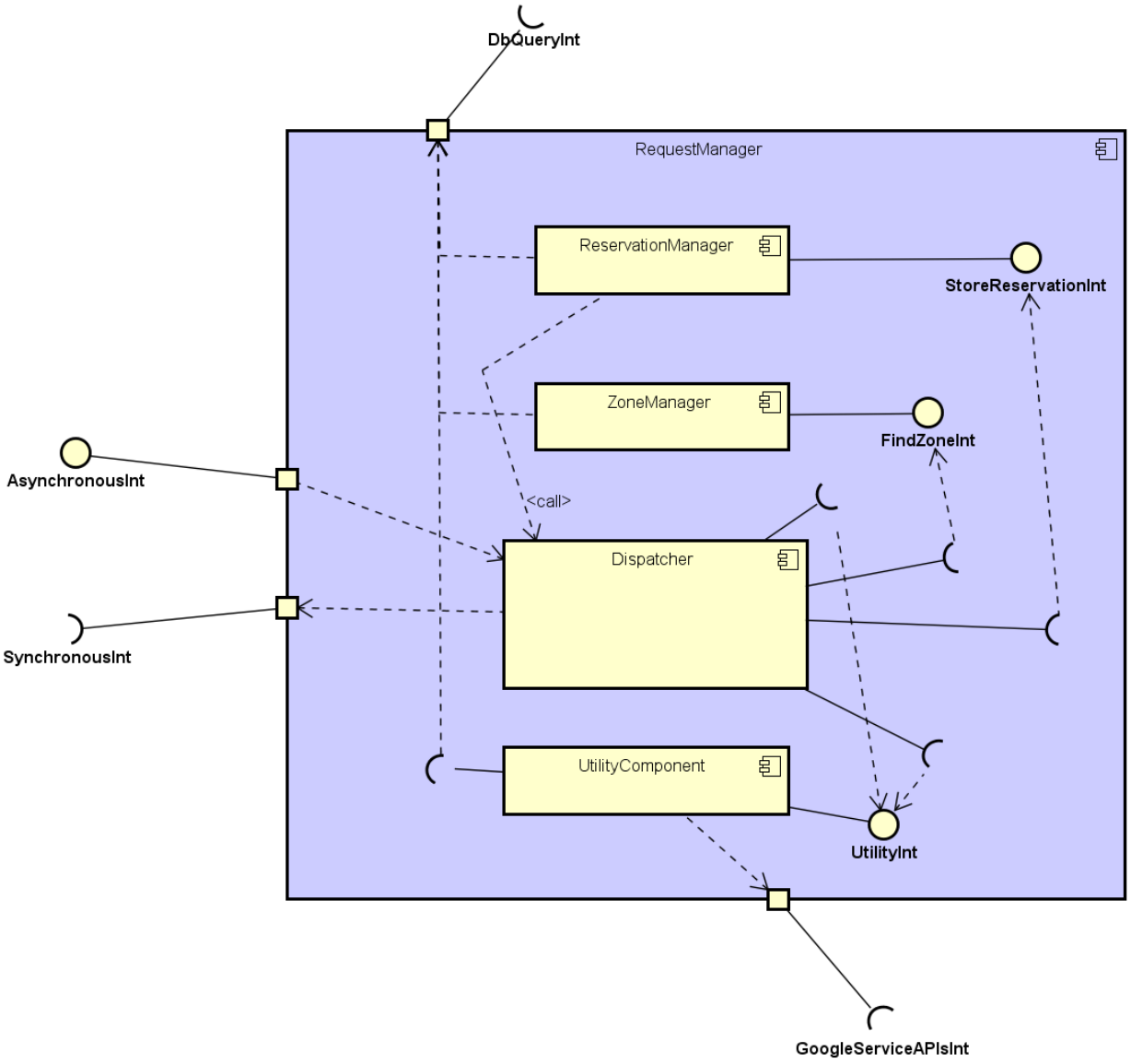
SERVER


DbOutInt
<<subsystem>>
Server
ConnectionManager
DatabaseManager
ConnectionInt
DbQueryInt
RequestManager
ReservationManager
ZoneManager
<call>
<use>
<use>
Dispatcher
<use>
<use>
UtilityComponent
CommunicationServer
AsynchronousCommunication
SynchronousCommunication
AsynchronousInt
SynchronousInt
CommunicationInterface

Component	Description	Interfaces
RequestManager	This is the most complex component in the Server, it has to manage the entire lifecycle of a received request (simple, sharing, reservation but also availability and departure declarations). It is composed by: • Dispatcher • UtilityComponent • ReservationManager • ZoneManager	• DbQueryInt (required); • AsynchronousInt (provided); • SynchronousInt (required);
DatabaseManager	This component manages all the interactions between others Server components and the data model. In particular, it allows the ConnectionManager and the RequestManager to have the result of database queries. DatabaseManager also hide the presence of an external database to the Server: if necessary, it calls the execution of a distributed query.	• DbQueryInt (provided); • DistributedQueryInt (required);
ConnectionManager	This module: • Manages the sign up attempt either by Guest and TaxiDrivers. It has to communicate to the DatabaseManager the presence of new Users and MyTaxiDrivers, in the case of TaxiDrivers SignUp it also has to ask to the DatabaseManager the existence of a	• ConnectionInt (provided); • DbQueryInt (required);

	corresponding TaxiDriver in the Urban Taxi Service. • Manages the log in attempt either by Users and TaxiDrivers. It calls the DatabaseManager to verify the existence of corresponding (same username and password) signed up User or TaxiDriver.	
CommunicationServer	This component manages, in the Server side, the communication between Clients and Server. It is composed by an Asynchronous module and a Synchronous module that are dedicated to the two different types of interactions.	• CommunicationInterface (provided); • SynchronousInt (provided); • AsynchronousInt (required); • ConnectionInt (required);
Asynchronous	This module forward the received requests to the Connection Manager (in case of log in, sign up request) or the Dispatcher (others possible requests).	• AsynchronousInt (required); • ConnectionInt (required);
Synchronous	This module allows the Dispatcher to send messages to the Clients (either User or TaxiDriver).	• SynchronousInt (provided);

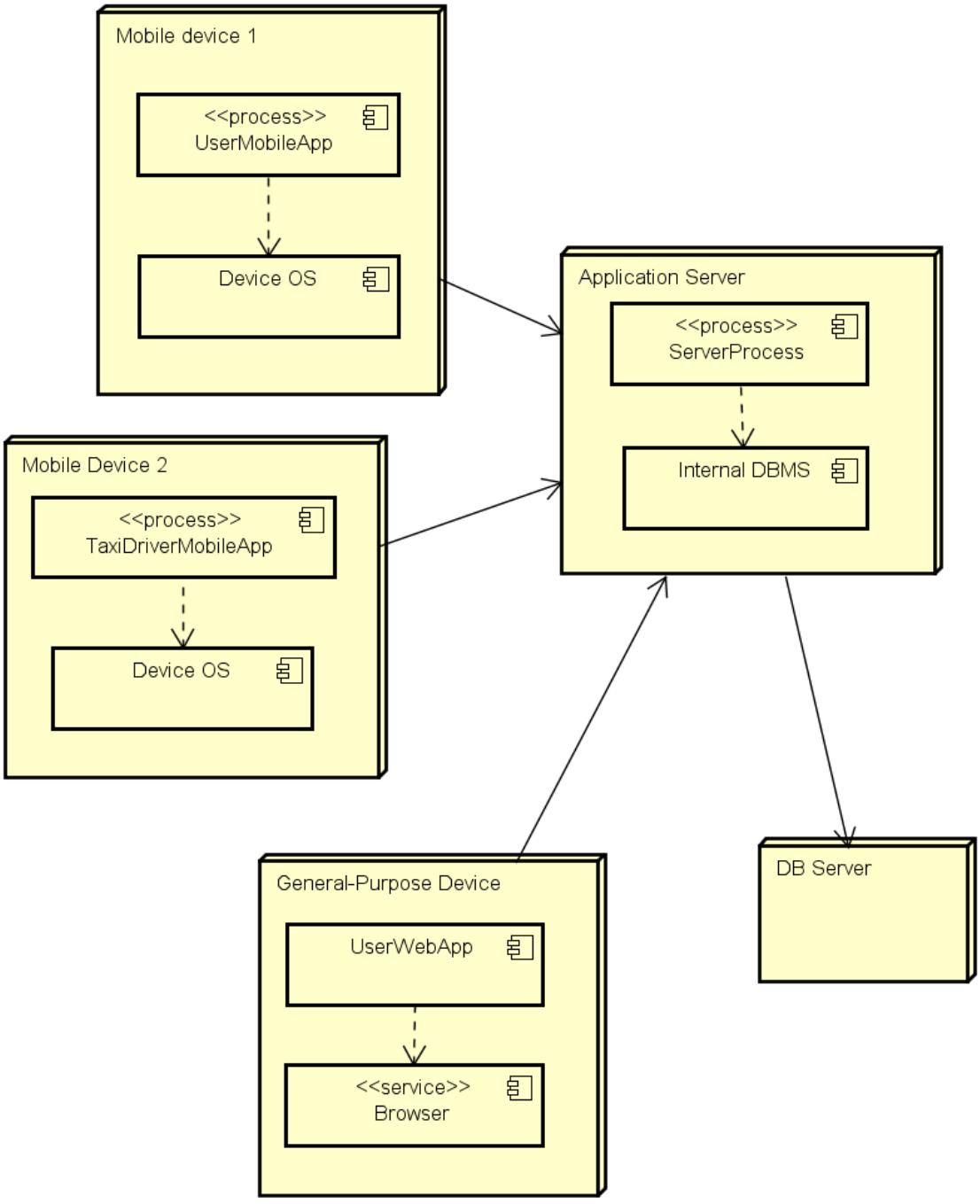
REQUEST MANAGER



Component	Description	Interfaces
Utility Component	This module: • Computes the route between two points, given their coordinates; this route is built to be covered by a car; • Returns the coordinates given an address name; • Computes the travel time given a departure point and a destination one; • Verifies if two routes follow the same direction (see the "same direction" property in the RASD Document); • Computes shared routes; • Computes all the fees given a shared route;	• UtilityInt (provided); • GoogleServiceAPIsInt (required); • DbQueryInt (required);
Zone Manager	This module may contain a data structure to store the limits of city zones in terms of coordinates. Given a pair of coordinates, it determines the zone to which the point belongs. Whenever the server is powered, it computes the city zones dynamically.	• FindZoneInt (provided); • DBQueryInt (required);
Reservation Manager	This module: • Stores a copy of all active reservations in the DB (in order to guarantee the service persistency); • Has a data structure that stores the reservations to check their departure time; • When 10 minutes are missing to the departure time it translates a reservation into a request and sends it to the dispatcher;	• StoreReservationInt (provided); • DBQueryInt (required); • DispatcherInt (required);

	• Add reservations after verifying that it's done at least 2 hours before the gathering time • Return the list of reservations done by a specific user • Serve a deletion request by delete a certain reservation in the data structure and the inner DB	
Dispatcher	This module: • Receives every kind of requests from the ComunicationServer component;(as delete a reservation , reservation, search a taxi ,taxi driver available…) • Dispatches requests by type and sending zone; • Manages queues and their policies; • Searches an available taxi, for each request, in the right taxi queue, according to the policies (see RASD Document); • Sends notifications to requesting clients; Requests are managed using a queue system, as done for the taxi. In the taxi queue system there are two kind of queues per zone: one for the non-shared taxi and another for the shared ones.	• AsynchronousInt (provided); • DispatcherReservationint (provided) • UtilityInt (required); • FindZoneInt (required); • StoreReservationInt (required); • SynchronousInt (required)

2.4 Deployment view



As Represented in the diagram above, the deployment view shows the allocation of the logical view elements to physical processing nodes, and the physical network configuration between nodes.

Executables

At the end of the deployment phase, the system should be made up of 4 executable files:

- UserMobileApp, UserWebApp and TaxiMobileApp as implementations of the client software;
- ServerProcess as implementation of the server logic;

The idea is that Mobile Apps could be set up in every kind of mobile device which has a wireless access by downloading them from the relative App Store.

For this reason, these apps have to satisfy the interoperability property, and work on every commercial environment properly (with the term "commercial" we mean, as instance, IOS, ANDROID and WINDOWS PHONE).

For what concerns the GPS constraint, our apps should call this service from the device OS (irrespective of the kind of this one).

Concerning the web one, typical users use a client web browser to access web apps and interact with them.

Our deployed product works in the same way: using a web browser, users could request the web pages composing its graphical interface and use the system functionality.

So, ours should be compatible to all the main ones, such as Google Chrome, Internet Explorer, Firefox, Opera and Safari.

The server executable file should be set up in a specific node, which will contain the inner DB, too.

Protocols

In this way, it will be possible to guarantee its maintainability and availability (it needs continuous monitoring and supervision, provided by some qualified employees).

Apps and server interact by sending messages: in this case, clients always begin the communication in the system and server has to reply to them (the reason depends on chosen architectural styles and is described in chapter 4.G: "Selected Architectural Styles and Patterns").

In this kind of communication, the system should use standard protocols such as https/Internet, in order to make all the system components reusable.

The critical aspect of the communication concerns messages between clients and server: these contain sensitive information and personal data in order to make all the users traceable from the server (the reason depends on the Dispatcher component design and is described in chapter 4.G: "Selected Architectural Styles and Patterns").

Therefore, a proper message coding is needed, by using cryptographic protocols together with basic communication protocols. This should preserve the integrity and the transparency.

Server has also to call the external DB to make some queries: in this case, messages between the two nodes should use SQL/XML languages and relative protocols, because all the queries are called only by the DB manager component and sent to the external one.

It's obvious that server must have the access permissions to use the data stored in the external DB (with respect to the DB policies).

APIs

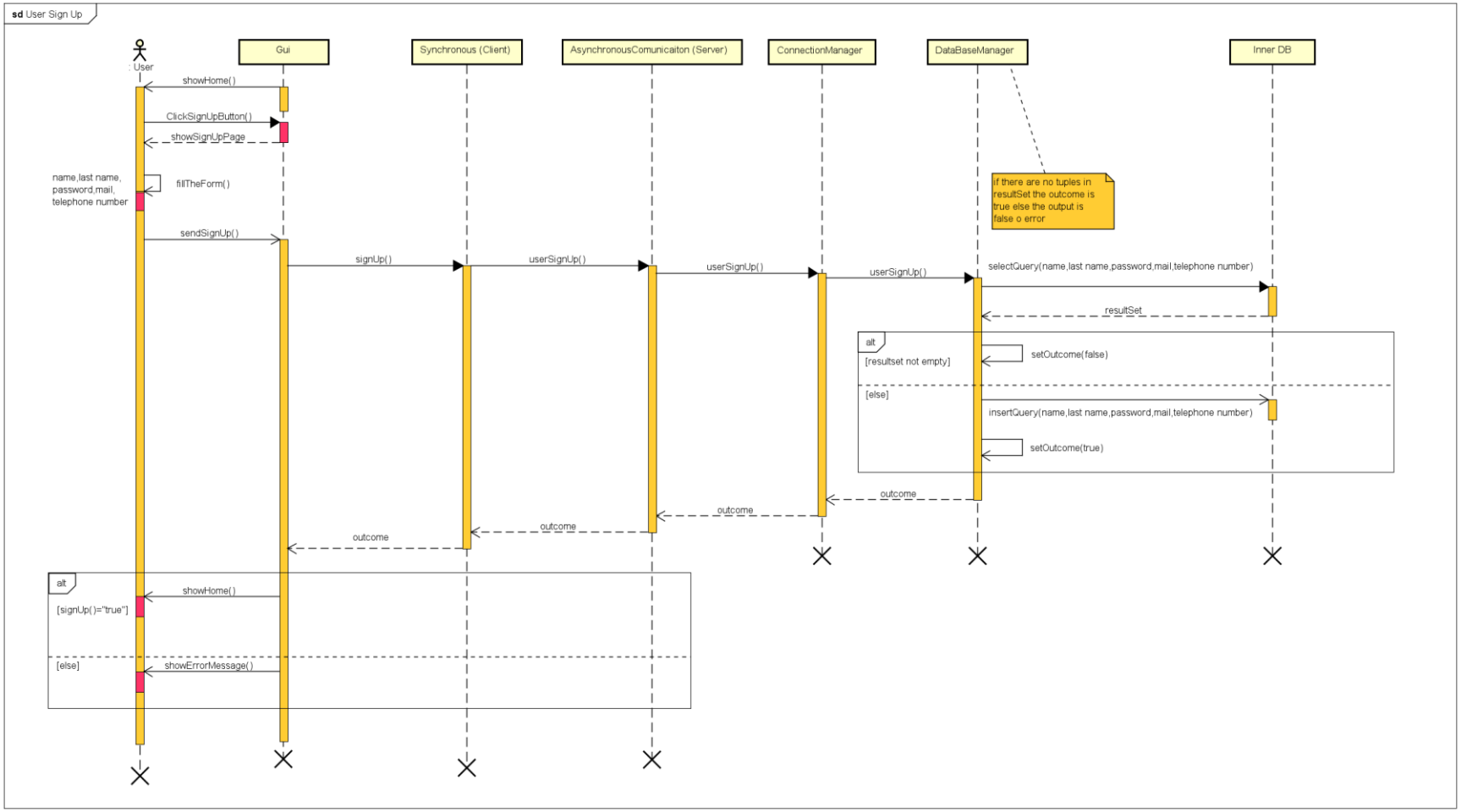
One of the goals that the architecture has to satisfy is the ability to offer its functionalities by calling a set of APIs in other projects. In order to do this, our server needs to follow the Web Service style properties.

We think that the system should allow other software to make taxi requests and reservations without using the usual Client Interface. Therefore, the architecture must add a sort of interface, in addition to the communication one, able to call the Dispatcher methods.

This interface should work in parallel with the Communication one, and, doing so, nothing should change from the point of view of the Dispatcher which should receive these messages as if were belonging to the ordinary classes of input.

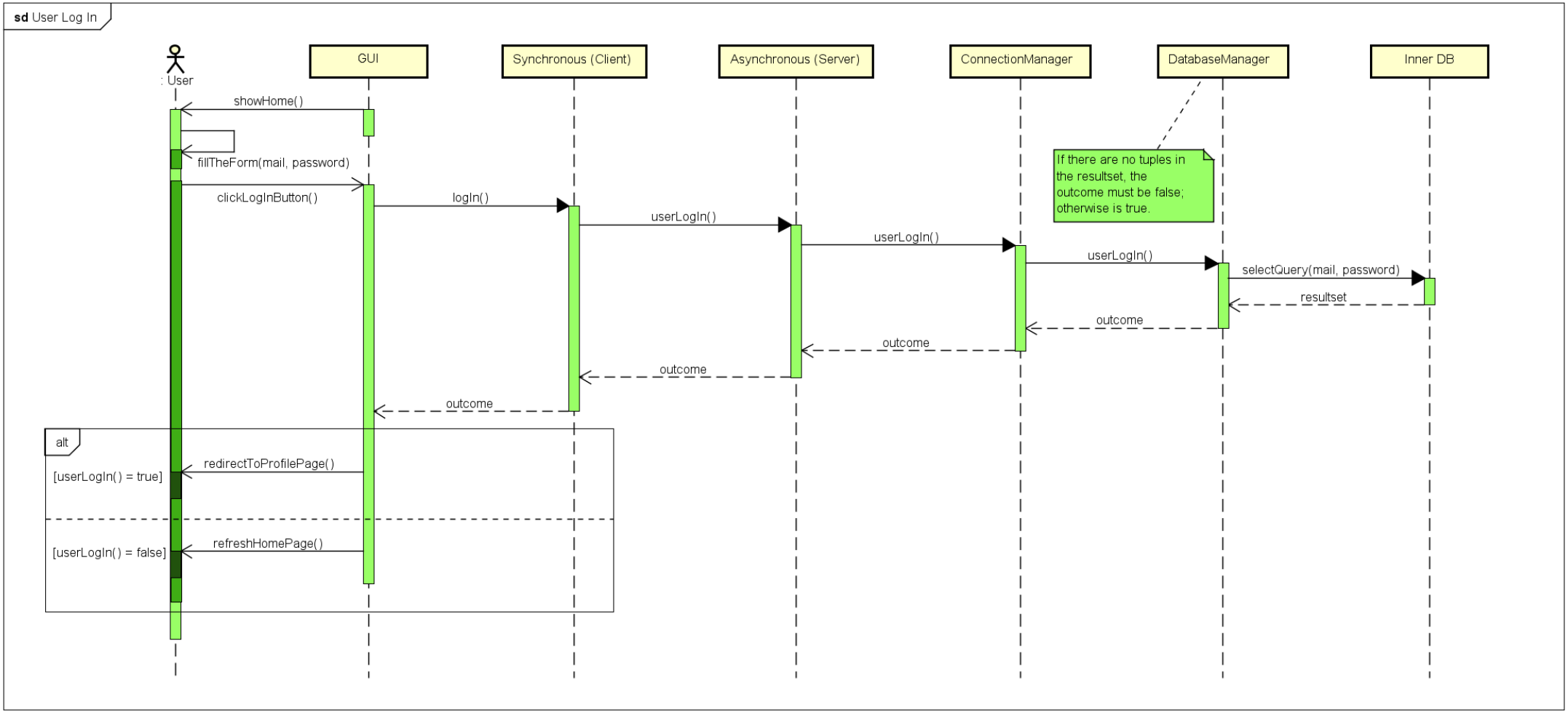
2.5 Runtime view

"User Sign Up" sequence diagram



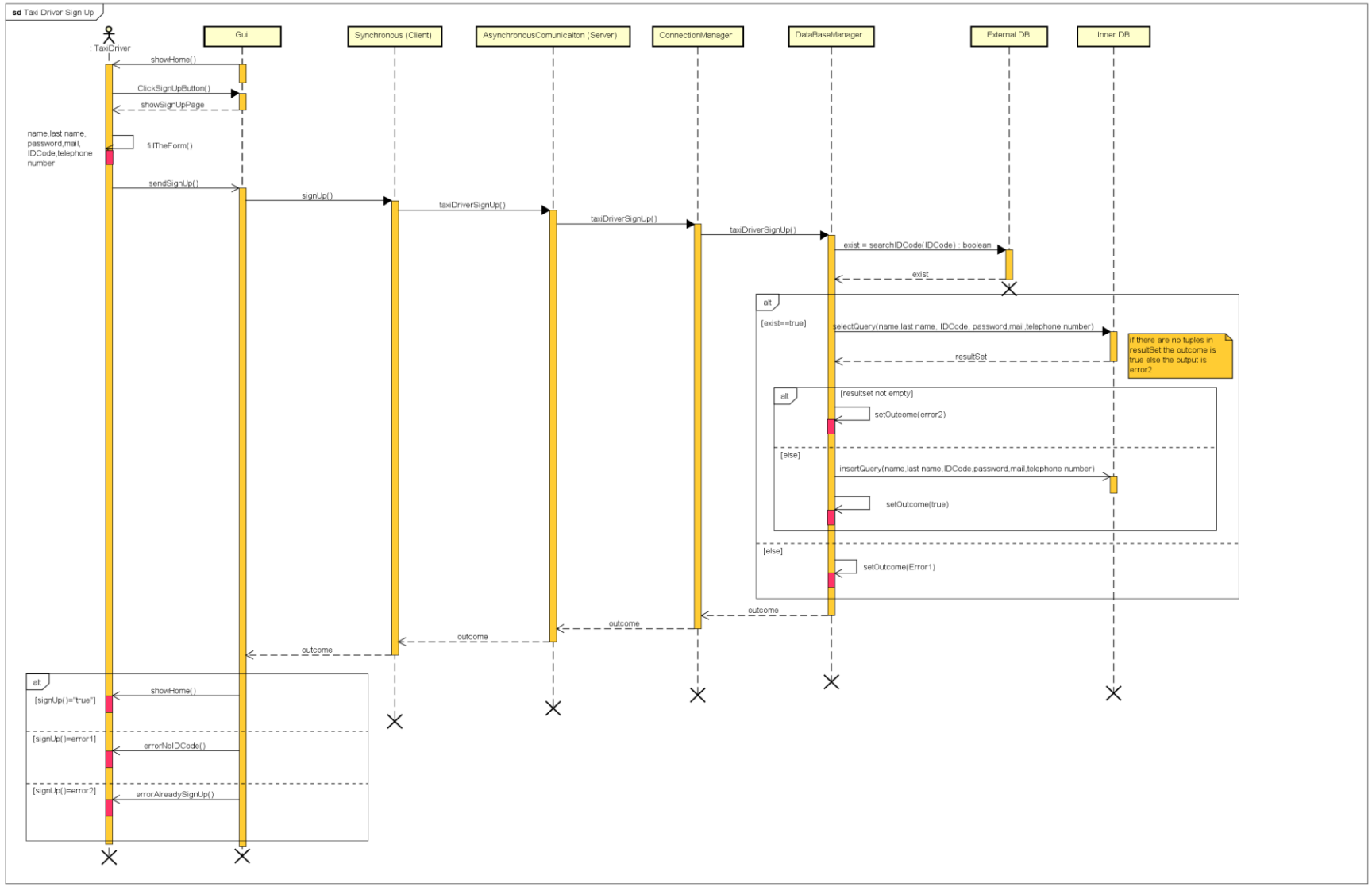
When the system receives a sign up request, it delegates the operation to the Connection Manager component which checks the validity of the inserted data and inserts them in the inner DB. The Client GUI reacts depending on the outcome.

“User Log In” sequence diagram



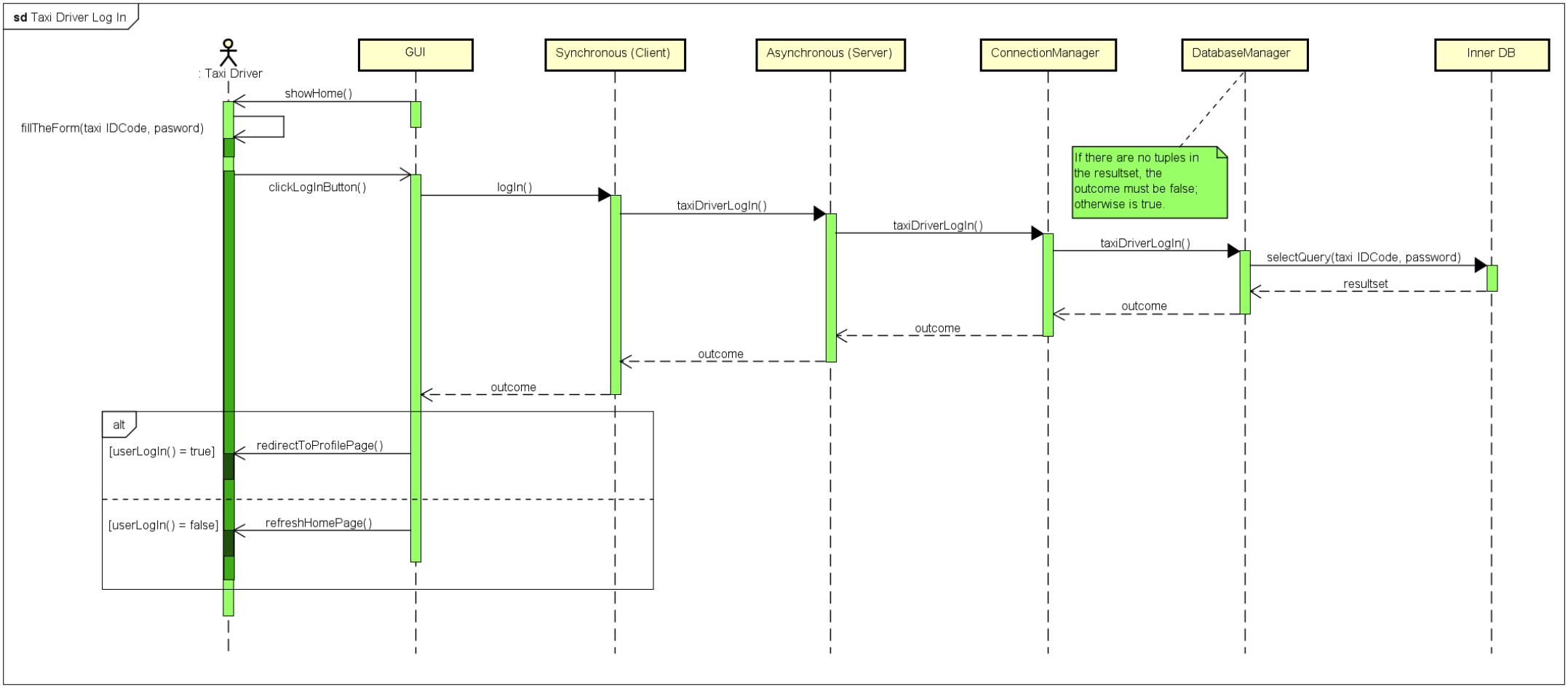
When the system receives a log in request, delegates the operation to the Connection Manager component which checks the validity of the inserted data. The Client GUI reacts depending on the outcome.

“Taxi Driver Sign Up” sequence diagram



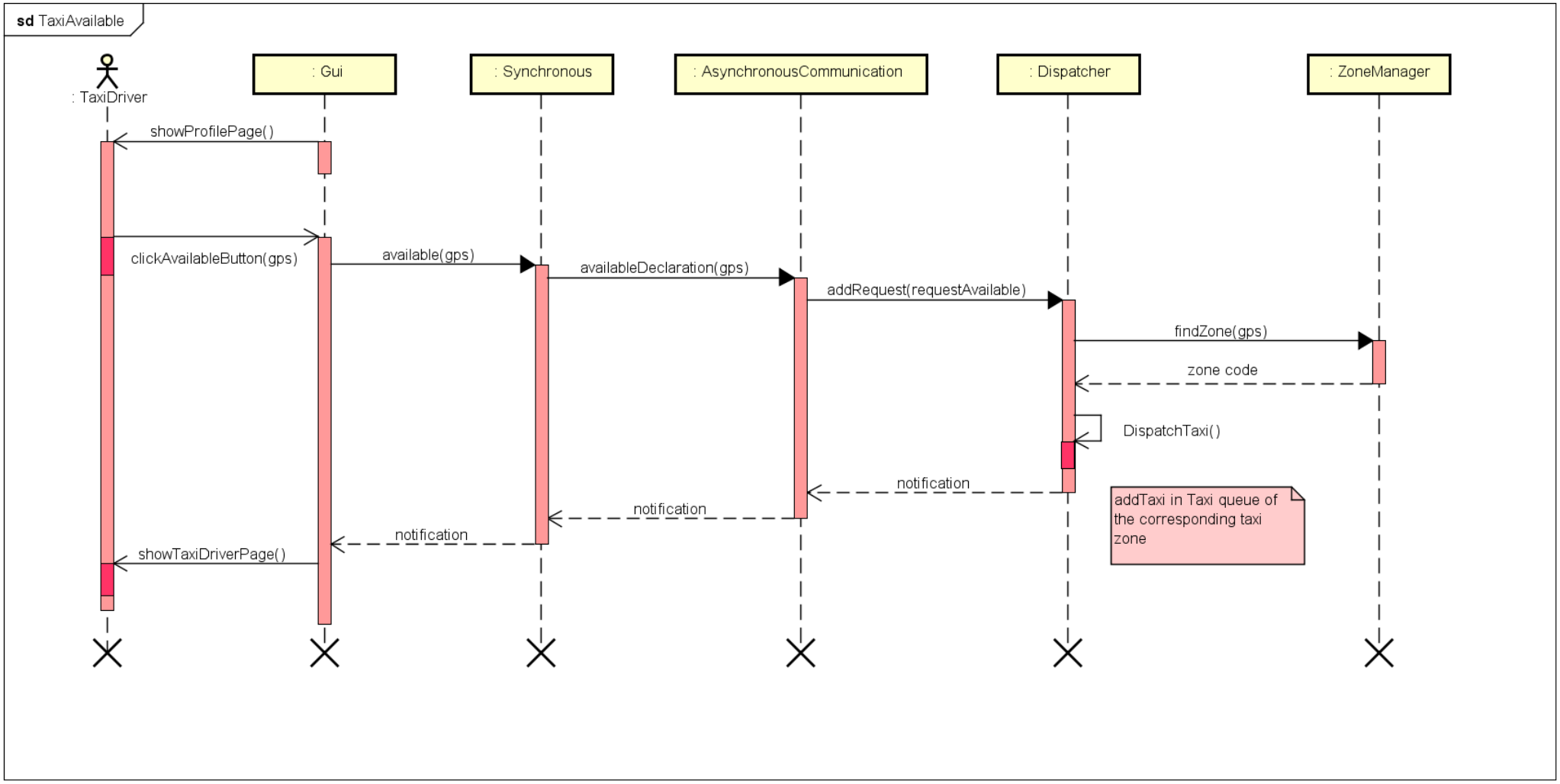
When the system receives a sign up request, it delegates the operation to the Connection Manager component: it checks the validity of the IDCode in the external DB; if the value exists, then it checks the validity of the inserted data and inserts them in the inner DB. The Client GUI reacts depending on the outcome.

“Taxi Driver Log In” sequence diagram



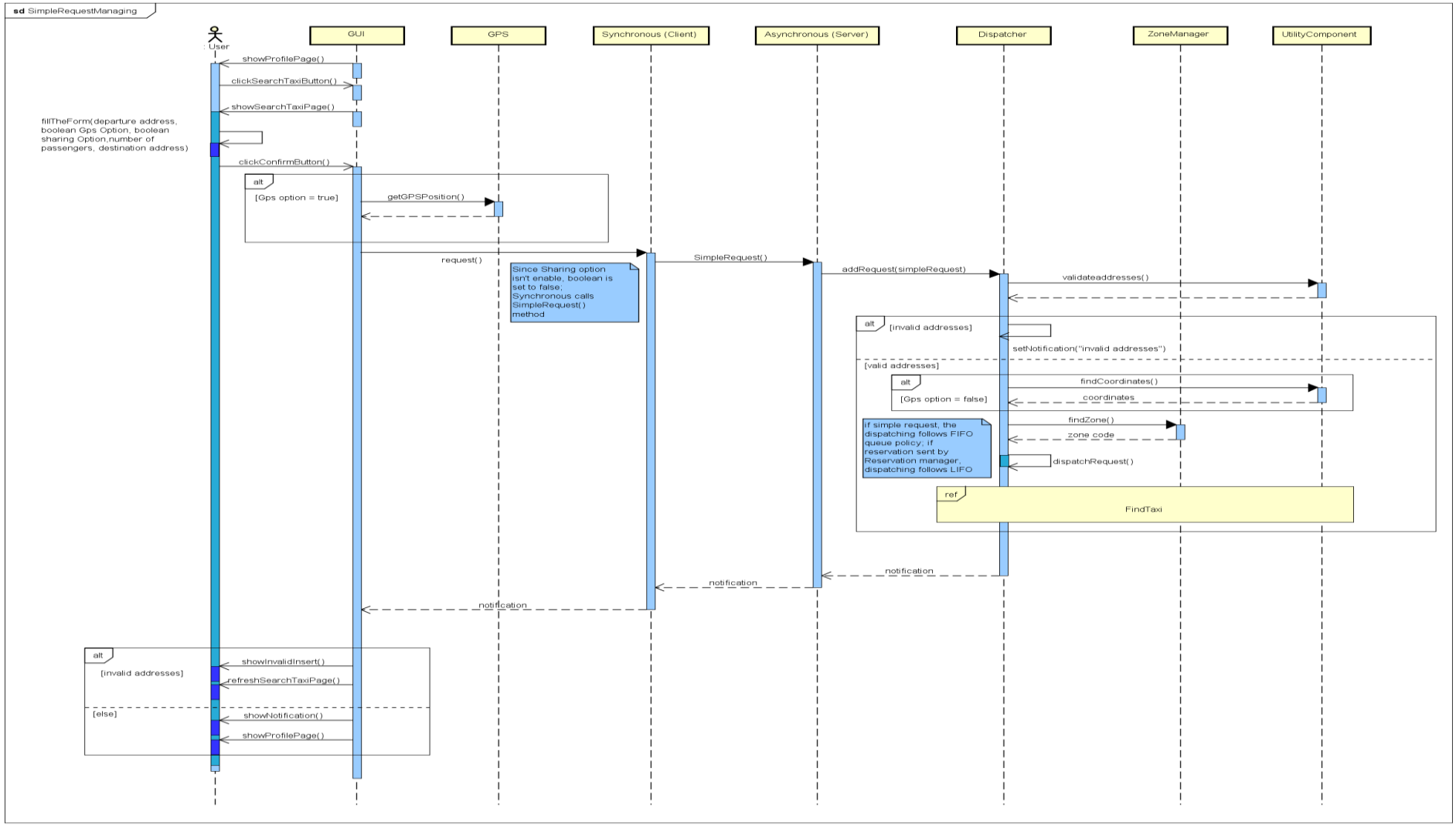
When the system receives a log in request, delegates the operation to the Connection Manager component which checks the validity of the inserted data. The Client GUI reacts depending on the outcome.

"Taxi availability" sequence diagram



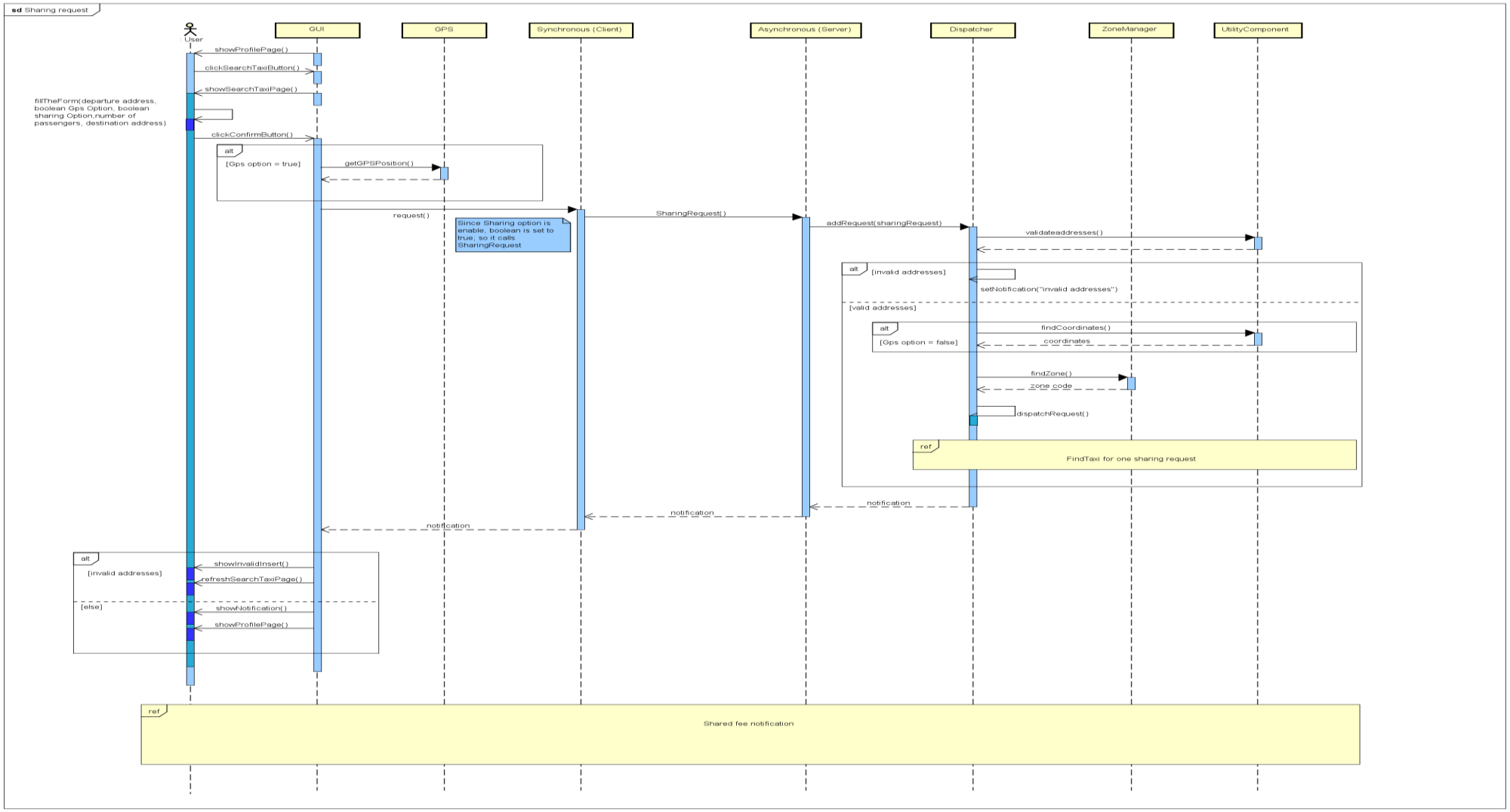
When the Taxi Driver clicks on the Available Button, the system mandates the Dispatcher to insert the taxi in the relative zone queue. When the Client GUI gets the request response, it shows the new Taxi Driver page.

“Simple Request Managing” sequence diagram



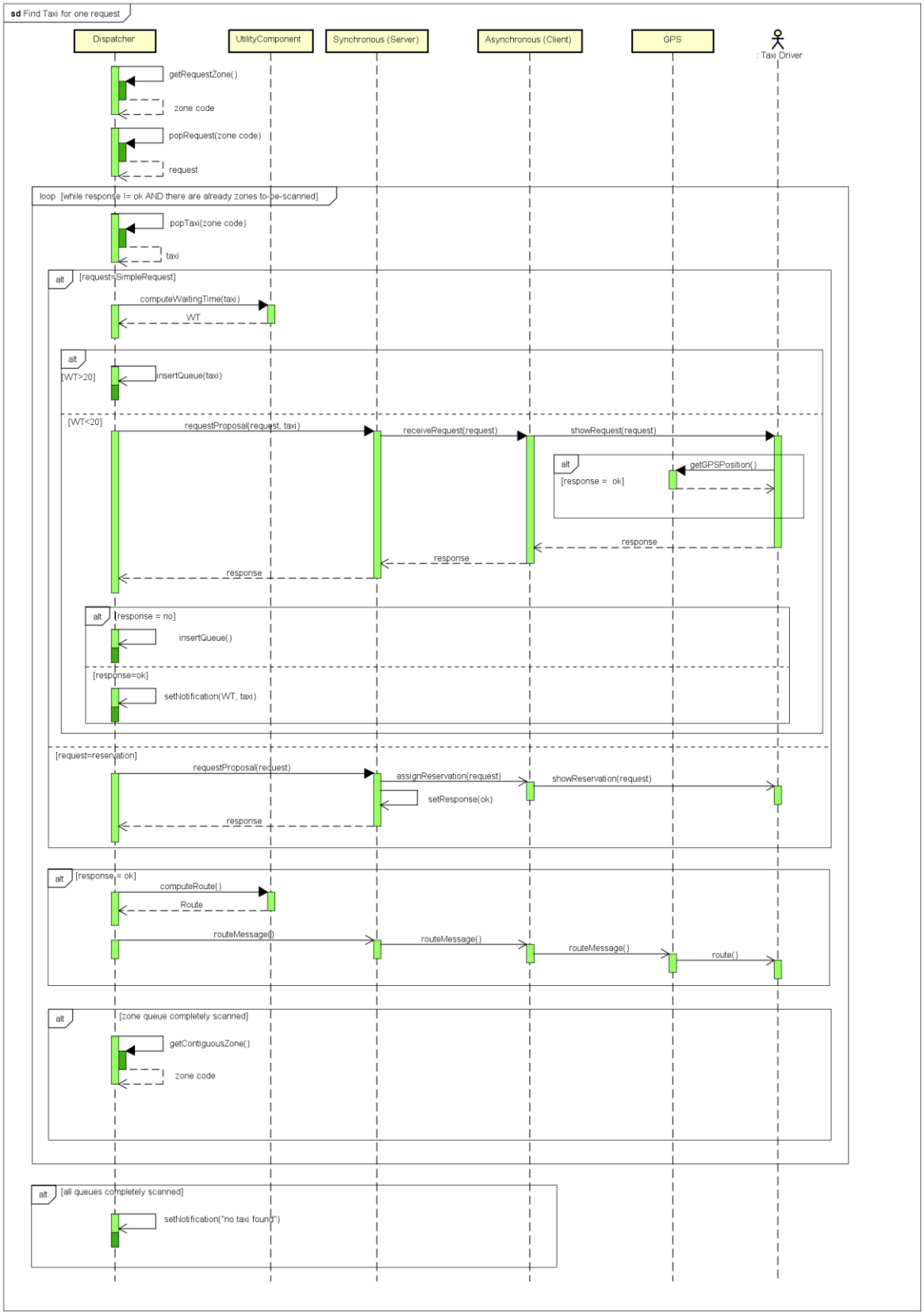
This diagram describes the system behavior when it receives a request belonging to the simple requests class. The request is sent to the dispatcher, which mandates the Zone Manager and the Utility Component to compute all the needed information and then manages the allocation of a taxi.

“Request with sharing option Managing” sequence diagram



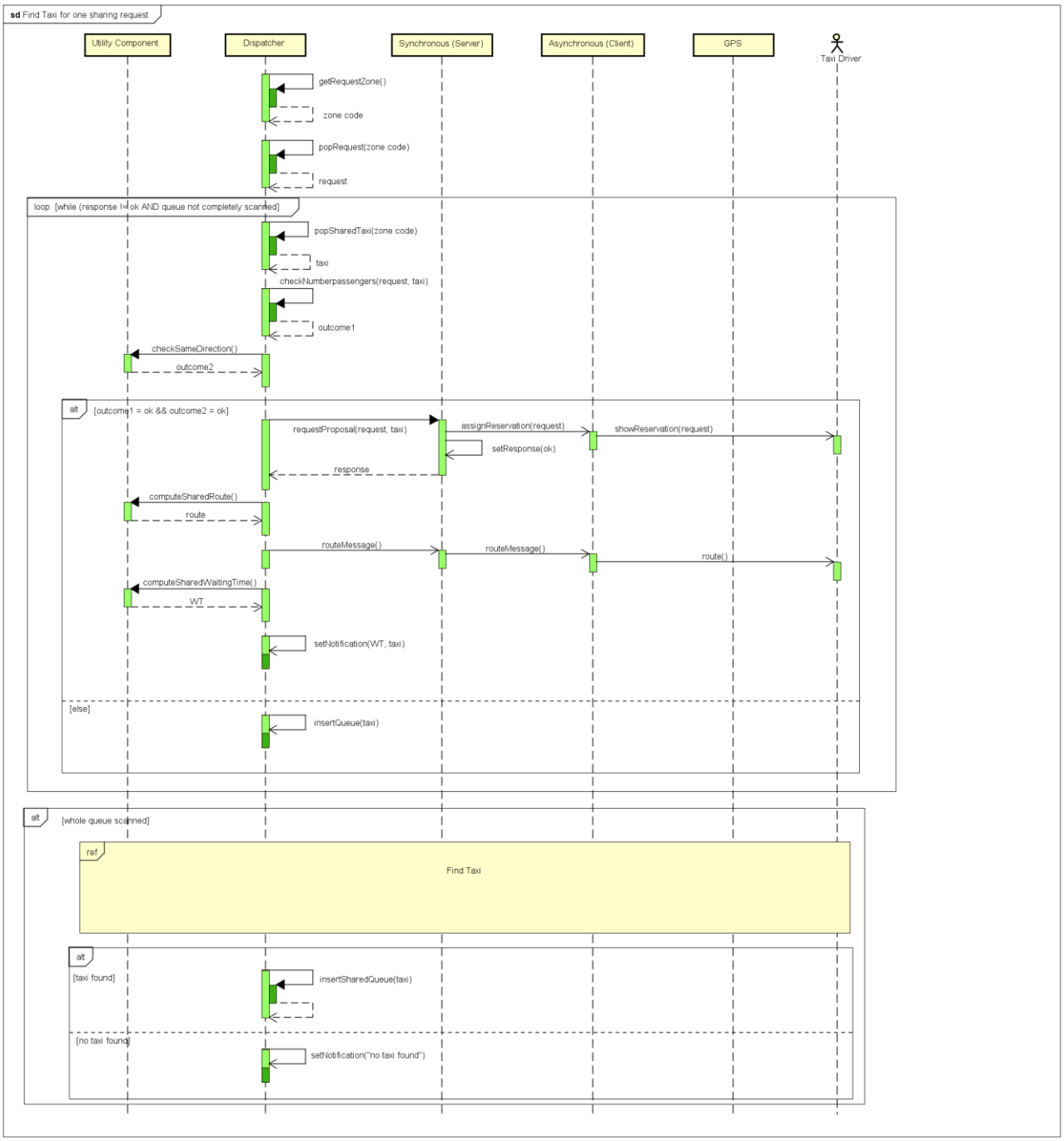
This diagram describes the system behavior when it receives a request belonging to the sharing requests class. It changes with respect to the previous one because, first of all, the allocation of a taxi for this kind of requests works in a different way, and also it needs an additional operation which will be described in the “Shared Fee Notification” sequence diagram.

"Find Taxi" sequence diagram



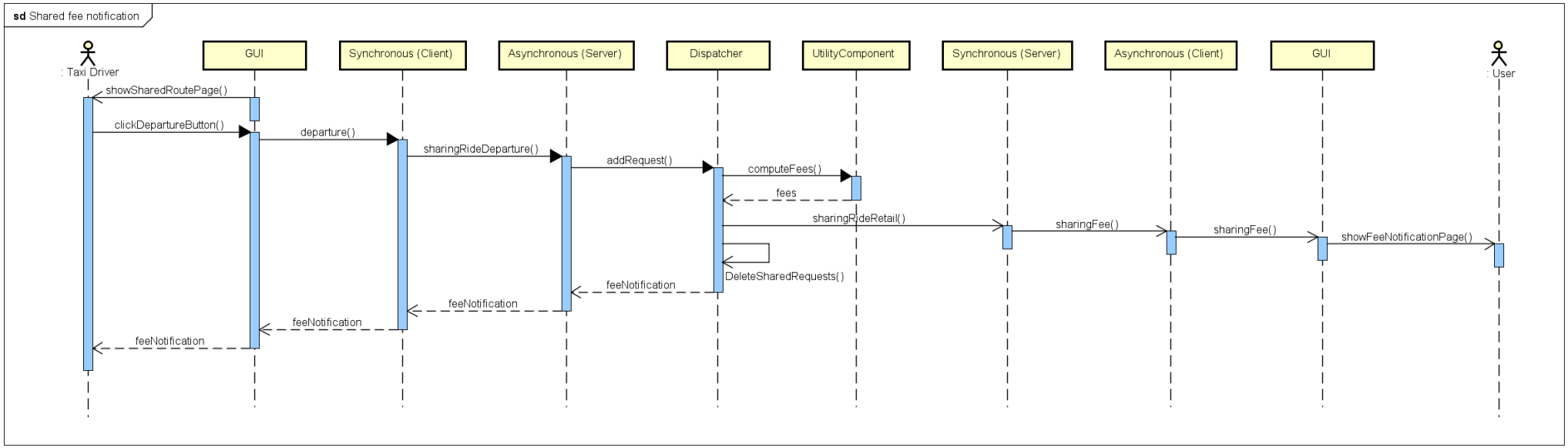
During a taxi allocation, the Dispatcher has to search an available taxi in the queue which corresponds to the served request zone. It sends message to each taxi driver sequentially, until it finds one accepting. Taxi queues are managed in according to the policies analyzed in the RASD Document.

“Find Taxi for one sharing request” sequence diagram



This diagram describes the Request Manager behavior during the allocation of a shared taxi. If the system doesn’t find an existing shared taxi able to serve the request, then it has to create a new one: this operation use the “Find Taxi” operation represented in the previous diagrams.

“Shared Fee Notification” sequence diagram

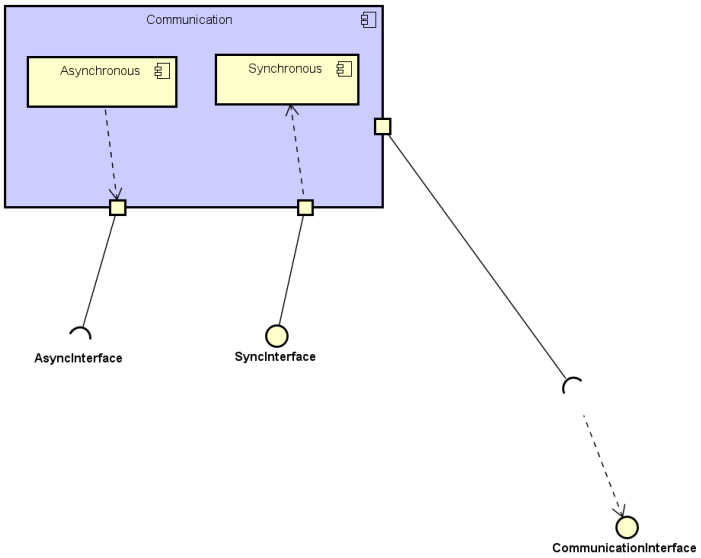


At the departure of a shared taxi, the system has to send all the fees to the involved user. This behavior is represented in this diagram.

2.6 Component interfaces

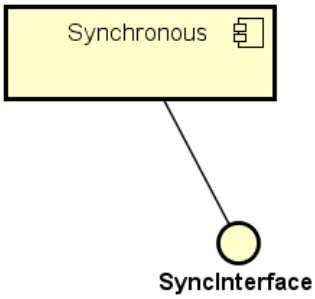
INTERFACE User

Communication



Requires the communication interface provided by Server.

Synchronous

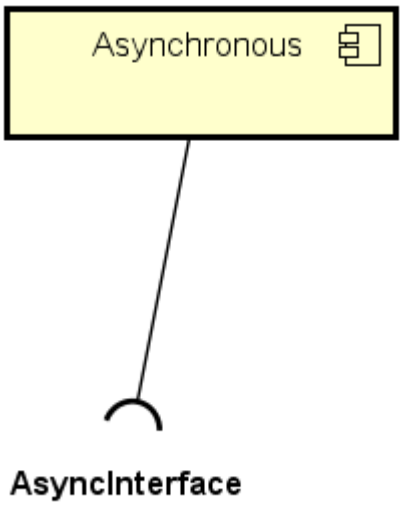


SyncInterface(provided): this interface allows the gui to contact the communication component when the user does an action.

- logIn
  - INPUT: mail and password
  - OUTPUT: message that contains if the information are correct or the username is wrong or the password is uncorrect
  - Calls the CommunicationInterface of server

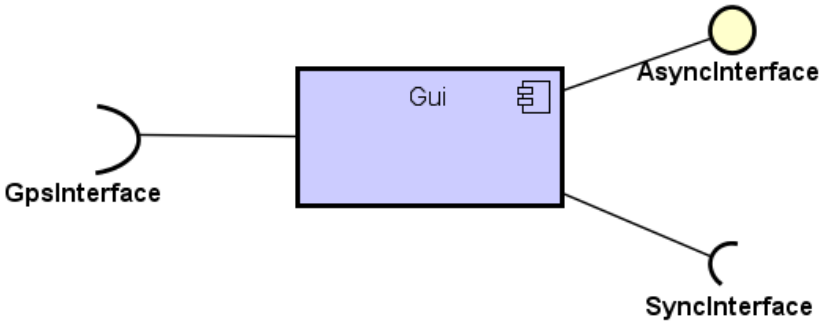
- signUp
  - INPUT: mail, last name, password, mail, telephone number
  - OUTPUT: message that notifies if the registration process is executed or not (in that case: why is not executed)
  - Calls the CommunicationInterface of server
- Request
  - INPUT:
    - departure address ,
    - Gps Option(Boolean),
    - sharing Option (Boolean) ,
    - number of person (int) ,
    - destination address(can be null if sharing option is false)
  - OUTPUT: return if a taxi is available (and includes also IDCode and waiting time) or not
  - Calls the CommunicationInterface of server
- Reservation
  - INPUT:
    - departure address ,
    - Gps Option(Boolean),
    - destination address,
    - date,
    - time,
    - sharing Option (Boolean) ,
    - number of person (int) .
  - OUTPUT: notify if the reservation is done accurately or there are some invalid input
  - Calls the CommunicationInterface of server
- Delete reservation
  - INPUT:
    - date,
    - time
  - OUTPUT: notify if the reservation is deleted or if it's not acceptable
  - Calls the CommunicationInterface of server
- ShowTaxi
  - INPUT: no input needed
  - OUTPUT: data structure that contains the information about the reservations that this user has done and if there is a request of taxi
  - Calls the CommunicationInterface of server

Asynchronous



AsyncInterface of GUI(required): this element contacts the Gui between the asyncInterface

Gui



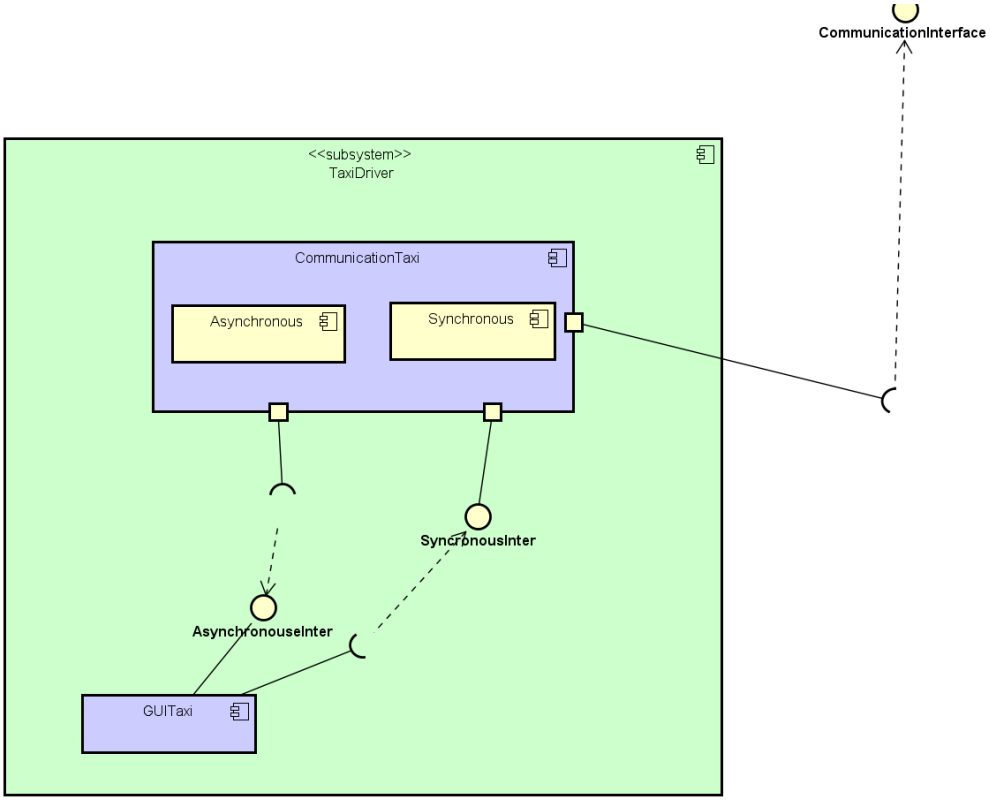
AsyncInterface(provided): this interface is called by the communication to notify messages to user

- sharingFee:
  - INPUT: fee of user, total cost of ride
  - OUTPUT: no output
  - Called by the Communication

SyncInterface(required): is called by the Gui when the user interact with the application or the web application

GpsInterface(required): Gui calls an external service provided by the device

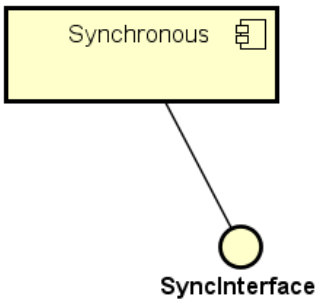
INTERFACE Taxi Driver



Communication

Requires the communication interface provided by Server

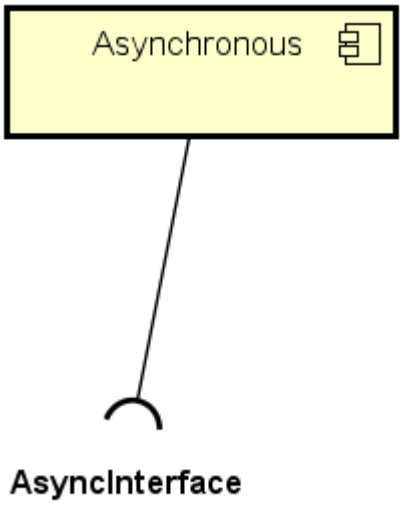
Synchronous



SyncInterface(provided): this interface provides the gui to contact the communication component when the taxi driver does an action.

- logIn
  - INPUT: Taxi IDCode and password
  - OUTPUT: message that contains if the information are correct or the IDCode is wrong or password is wrong
  - Calls the Communication component of server
- signUp
  - INPUT: name, last name, password, Taxi IDCode, telephone number
  - OUTPUT: message that notifies if the registration process is executed or not (in that case: why is not executed)
  - Calls the CommunicationInterface of server
- Available
  - INPUT: GPS coordinate
  - OUTPUT: no output
  - Calls the CommunicationInterface of server to notify the availability of new taxi driver
- Departure
  - INPUT: no input
  - OUTPUT: return the fee of each passenger aboard the taxi and the total cost of the ride
  - Calls the CommunicationInterface of server to communicate the departure with all sharing passengers

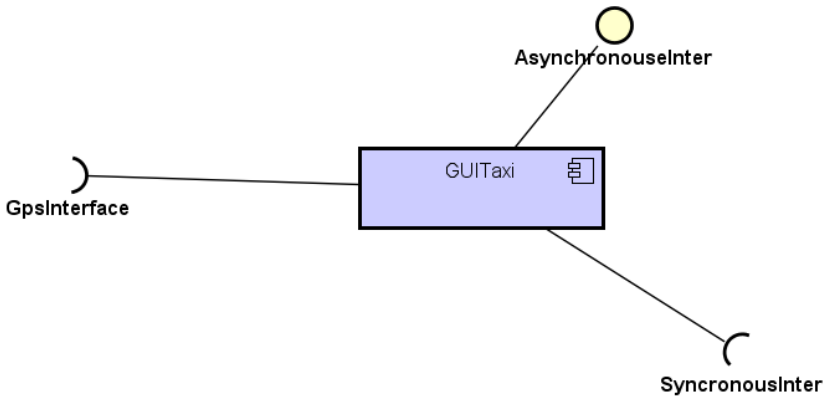
Asynchronous



AsyncInterface of GUI (required): Asynchronous Communication element contacts the Gui between the asyncInterface :

- ReceivedRequest
  - INPUT: data structure of request
  - OUTPUT: return if the taxi driver accepts or not
  - Called by the Communication component of server
- RouteMessage
  - INPUT: route details of a request
  - OUTPUT: none
  - Called by the Communication component of server

Gui



AsynchronouseInter(provide):

- Reservation
    - INPUT: data structure of reservation
    - OUTPUT: no output
    - Called by the Communication component of server to notify a reservation

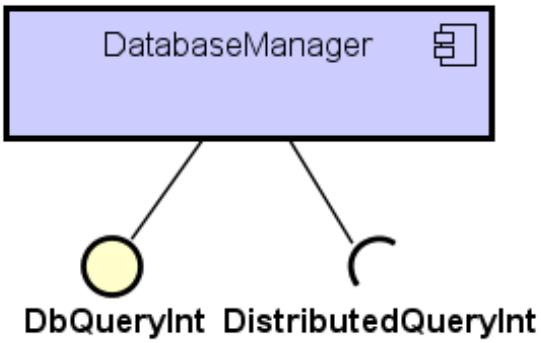
- NewSharingPassenger
    - INPUT: data structure of sharing ride with the new route
    - OUTPUT: no output
    - Called by the Communication component of server to notify a new passenger with sharing ride

SynchronouseInter(required): is called by the GUITaxi when the user interact with the application

GpsInterface(required): Gui calls an external service provided by the device

INTERFACES Server

DatabaseManager



DbQueryInt: this interface is called by the ConnectionManager, the ReservationManager and the ZoneManager.

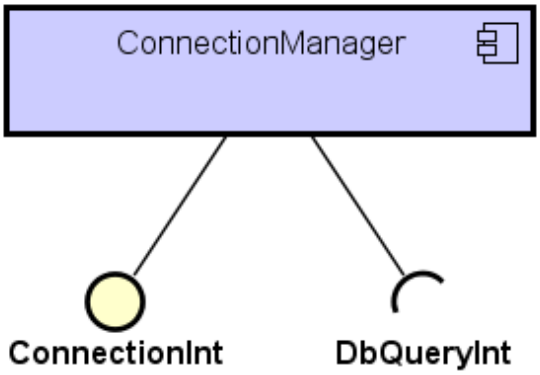
- UserSignUp
  - INPUT: required User data for sign up operation (name, last name, mail, password, telephone number);
  - OUTPUT: outcome of the insertion operation into the Database;
  - The operation executes an update in the Database;
- TaxiDriverSignUp
  - INPUT: required TaxiDriver data for sign up operation (name, last name, mail, password, ID Code, telephone number);
  - OUTPUT: outcome of the insertion operation into the Database;
  - The operation check if a corresponding TaxiDriver exists in the Urban Taxi Service Database and then it executes an update in the System Database;
- UserLogIn
  - INPUT: required User data for login operation (mail, password);
  - OUTPUT: outcome of log in operation;
  - The operation executes a query in the Database;
- TaxiDriverLogIn
  - INPUT: required TaxiDriver data for login operation (IDCode, password);
  - OUTPUT: outcome of log in operation;
  - The operation executes a query in the Database;
- AddReservation
  - INPUT: User requestor, Date and Time of the reservation;
  - OUTPUT: outcome of the operation;
  - The operation executes an update into the Database;
- DeleteReservation

    - INPUT: User requestor, Date and Time of the reservation;
    - OUTPUT: outcome of the operation;
    - The operation executes a delete in the Database;
- ReservationSetImport
    - INPUT: it doesn't need an input value;
    - OUTPUT: list of all the reservation inserted in the database;
    - The operation executes a query in the Database;
- ZoneInitalPointsImport
    - INPUT: it doesn't need an input value;
    - OUTPUT: list of the points needed as input for the SetupZones (algorithm);
    - The operation executes a query in the Database;
- PricePerKmImport
    - INPUT: MyTaxiDriver Id;
    - OUTPUT: value of the price per kilometer related to the Company of the specified MyTaxiDriver;
    - The operation executes a query in the Database;
- SharingIncrementImport
    - INPUT: MyTaxiDriver Id;
    - OUTPUT: sharing increment value related to the Company of the specified MyTaxiDriver;
    - The operation executes a query in the Database;

DistributedQueryInt: this interface is required by the DatabaseManager; it's the administrator of the External Database that have to take care of the DistributedQueryInt development.

- TaxiDriverVerification
    - INPUT: name, last name and IDCode of a TaxiDriver;
    - OUTPUT: true if exist a corresponding tuple into the Urban Taxi Service database, false otherwise;
    - The operation executes a query in the External Database;

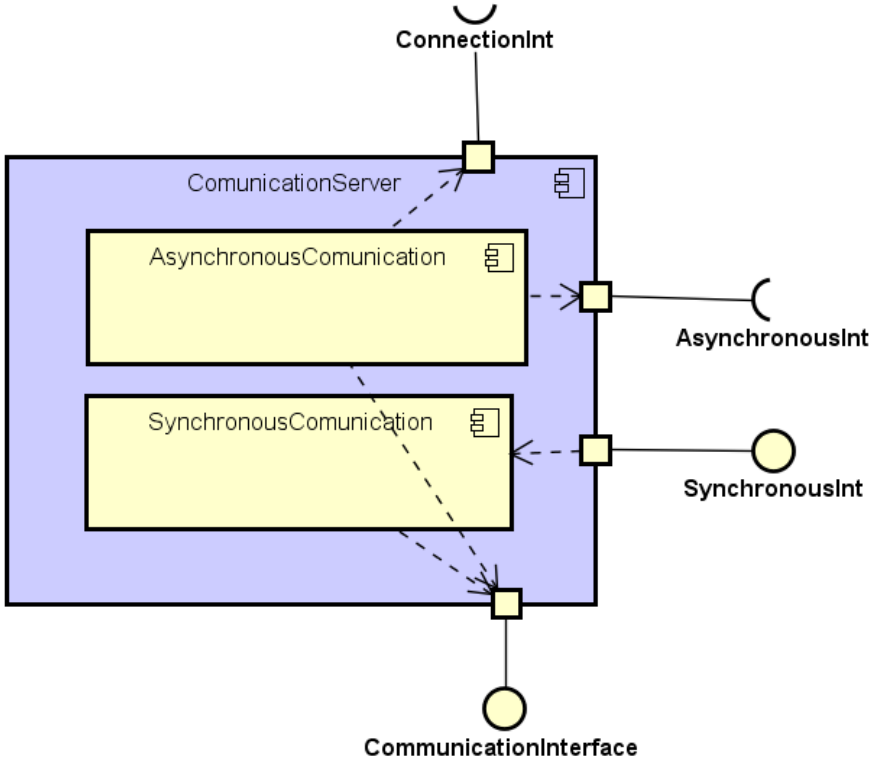
ConnectionManager



ConnectionInt: this interface is called by the CommunicationServer

- UserSignUp
  - INPUT: required User data for sign up operation (name, last name, mail, password, telephone number);
  - OUTPUT: outcome of the sign up operation;
  - The operation call the corresponding DbQueryInt operation;
- TaxiDriverSignUp
  - INPUT: required TaxiDriver data for sign up operation (name, last name, mail, password, ID Code, telephone number);
  - OUTPUT: outcome of the sign up operation;
  - The operation call the corresponding DbQueryInt operation;
- UserLogIn
  - INPUT: required User data for login operation (mail, password);
  - OUTPUT: outcome of the log in operation;
  - The operation call the corresponding DbQueryInt operation;
- TaxiDriverLogIn
  - INPUT: required TaxiDriver data for login operation (IDCode, password);
  - OUTPUT: outcome of the login operation;
  - The operation call the corresponding DbQueryInt operation;

CommunicationServer



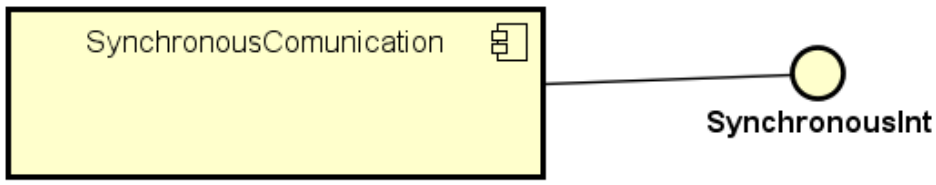
CommunicationInterface: this interface is called by UserClient and TaxiDriverClient

- UserSignUp
  - INPUT: required User data for sign up operation (name, last name, mail, password, telephone number);
  - OUTPUT: outcome of the sign up operation;
  - The operation call the corresponding ConnectionInt operation;
- TaxiDriverSignUp
  - INPUT: required TaxiDriver data for sign up operation (name, last name, mail, password, ID Code, telephone number);
  - OUTPUT: outcome of the sign up operation;
  - The operation call the corresponding ConnectionInt operation;
- UserLogIn
  - INPUT: required User data for login operation (mail, password);
  - OUTPUT: outcome of the log in operation;
  - The operation call the corresponding ConnectionInt operation;
- TaxiDriverLogIn
  - INPUT: required TaxiDriver data for login operation (IDCode, password);
  - OUTPUT: outcome of the login operation;
  - The operation call the corresponding ConnectionInt operation;

- SimpleRequest
    - INPUT: departure address (or current position), number of people;
    - OUTPUT: IDCode (of the TaxiDriver that will take care of the request) and waiting time;
    - The operation call the corresponding AsynchronousInt operation;
- ReservationRequest
    - INPUT: departure address (or current position), number of people, destination address, date and time of the reservation, sharing option flag;
    - OUTPUT: outcome of the reservation;
    - The operation call the corresponding AsynchronousInt operation;
- SharingRequest
    - INPUT: departure address (or current position), number of people;
    - OUTPUT: IDCode (of the TaxiDriver that will take care of the request) and waiting time;
    - The operation call the corresponding AsynchronousInt operation;
- SharingRideDeparture
    - INPUT: none;
    - OUTPUT: total and partial fee which the passengers have to pay;
    - The operation call the corresponding AsynchronousInt operation;
- AvailableDeclaration
    - INPUT: current position;
    - OUTPUT: none;
    - The operation call the corresponding AsynchronousInt operation;
- ShowUserReservation
    - INPUT: none;
    - OUTPUT: list of the reservations done by the requestor User;
    - The operation call the corresponding AsynchronousInt operation;

The different Client type has different visibility of the CommunicationInterface: UserClient calls only UserLogIn, UserSignUp, SimpleRequest, ReservationRequest, ShowUserReservation and SharingRequest, TaxiDriver calls TaxiDriverLogIn, TaxiDriverSignUp and AvailableDeclaration, SharingRideDeparture.

Synchronous

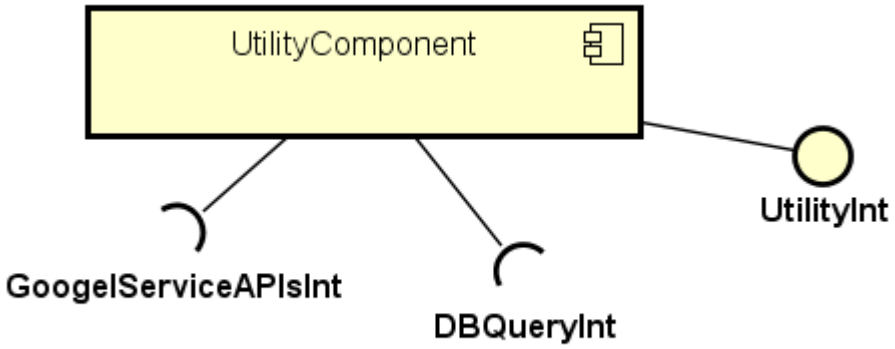


SynchronousInt: this interface is called by the Dispatcher.

- RequestProposal
  - INPUT: request detail, TaxiDriver that has to receive the proposal;
  - OUTPUT: TaxiDriver answer to the proposal;
  - It calls the ClientTaxiDriver;
- SharerAddition
  - INPUT: route to reach the new passenger of the sharing route, TaxiDriver that has to receive the notification;
  - OUTPUT: none;
  - It calls the ClientTaxiDriver;
- SharingRideDetail
  - INPUT: total and partial fee, Users involved in the ride, route of the sharing ride, TaxiDriver that has to receive the ride details;
  - OUTPUT: none;
  - It calls the ClientUser;
- RouteMessage
  - INPUT: route details of a request
  - OUTPUT:none
  - It calls the ClientUser;

INTERFACES Request Manager

Utility Component



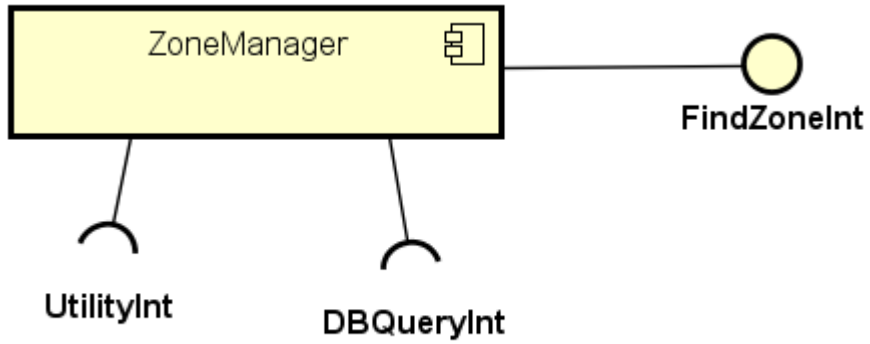
UtilityInt: this interface is called by the Dispatcher component

- ComputeRoute
  - INPUT: two valid points represented by their coordinates (reachable by car);
  - OUTPUT: the shortest route that links the two points;
  - The operation calls Google web services to obtain the route;
- FindCoordinates
  - INPUT: an existing address represented by its name;
  - OUTPUT: the relative pair of coordinates;
  - The operation calls Google Geolocation service to obtain the desired output;
- ComputeWaitingTime
  - INPUT: the taxi GPS position and the gathering point, represented by their coordinates (reachable by car);
  - OUTPUT: the travel time of the shortest route that links the two points;
  - The operation calls Google web services to obtain the route and its travel time;
- CreateSharedRoute
  - INPUT: a shared route and a destination point in the same direction;
  - OUTPUT: a new shared route that includes the point;
  - The operation returns the shortest route that includes both the old shared route and the new destination; if the given shared route doesn't exist, it calls the ComputeRoute function using the taxi GPS position as departure and the given point as destination;
- CheckSameDirection
  - INPUT: a shared route and a valid destination point;

    - OUTPUT: a Boolean value that represents the validity of the property;
    - The operation checks the “same direction” property following its definition (see the paragraph ”Assumption” in the RASD document;
- ComputeFees
    - INPUT: a shared route and the number of served Users;
    - OUTPUT: all the relative fees (total and partials);
    - The operation computes the fees that Users have to pay for a specific shared request (see chapter “Algorithms” to understand the meaning of the function);

It requires the GoogleServiceAPIsInt interface to execute some methods and requires the DBQueryInt.

Zone Manager



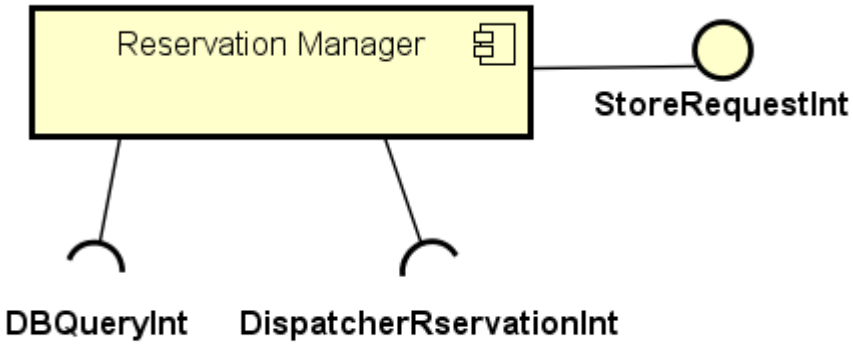
FindZoneInt: this interface is called by the Dispatcher component

- FindZone
    - INPUT: a pair of coordinates representing a point in the city;
    - OUTPUT: the city zone that includes it;
    - The operation searches the range of coordinates that includes the given ones and returns the relative zone;
- SetupZones
    - INPUT: a set of initial coordinates (stored in the internal DB);
    - OUTPUT: a data structure containing the limits of each zone in terms of coordinates;

- The operation is performed once the server starts working and computes the zones dynamically;

It requires the DBQueryInt to recover the parameters of the SetupZones method, and the UtilityInt

Reservation Manager



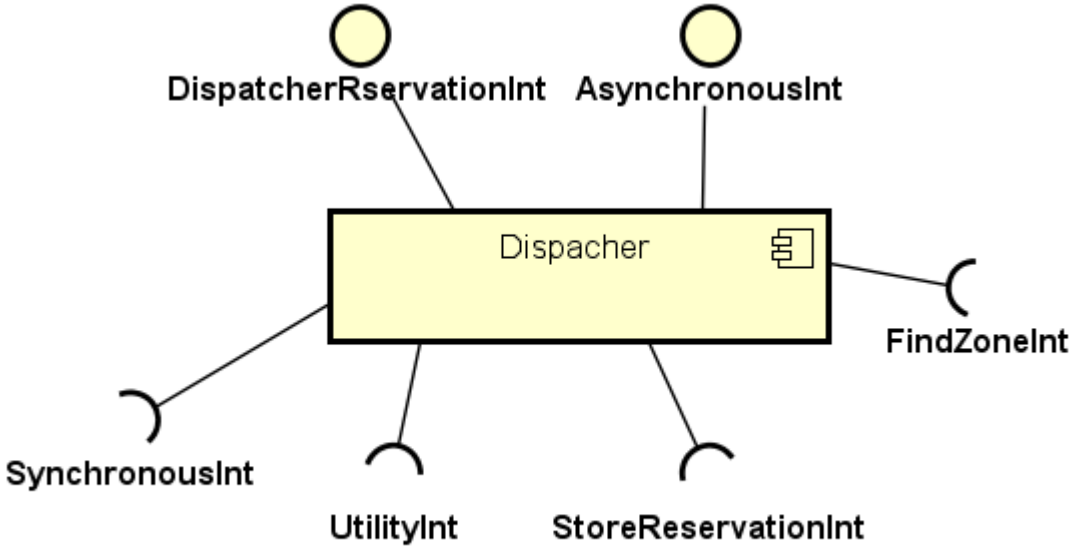
StoreReservationInt: this interface is called by the Dispatcher component

- AddReservation
  - INPUT: a new reservation
  - OUTPUT: this operation doesn't return
  - The operation stores the new reservation both in the internal Database and in a specific data structure, only if the constraints are satisfied (at least 2 hours before);
- CheckReservation
  - INPUT: no input needed;
  - OUTPUT: no return value needed;
  - It checks in the data structure if any reservation is ready to be served: in that case, ready reservations are translated into requests and sent to the dispatcher;
- ListReservations
  - INPUT: the requesting user id;
  - OUTPUT: a list of reservation made by the requesting user;
  - It searches all the reservations made by the requesting user stored in the data structure and return them to the user;

- DeleteReservation
    - INPUT: User, Time and Date of a reservation;
    - OUTPUT: no return value;
    - It searches in the data structure and in the inner DB the specific reservation and delete it; then shows the new list;

It requires the DBQueryInt interface to call the writing operations in the DB and needs to call the Dispatcher to send the translated reservations to it.

Dispatcher



AsynchronousInt: called by the Asynchronous component included in the communication module.

- AddRequest
    - INPUT: a new request (every kind of request)
    - OUTPUT: a notification about the search result
        - An error message if the request is not valid (invalid address, sending error);
        - A message containing the taxi ID Code and the relative waiting time if the request is served;
        - A list of reservation if the request is about listing or deleting reservations;

        - A confirmation if the request is about taxi availability;
        - A confirmation if the request belongs to the reservation request type;
        - A message including the total cost of the travel, the users' fees and the shared route if the request is about the departure of a shared taxi;

DispatcherReservationInt: called by the Reservation Manager module

- ServeReservation
    - INPUT: at least one reservation ready to be served;
    - OUTPUT: no return value needed;
    - The operation instantiates a request for each input reservation and dispatches them in the queue system inside the Dispatcher component;
- SendList
    - INPUT: list of reservation;
    - OUTPUT: no return value;
    - It instantiates a request including the list and call the communication component to send it

It requires UtilityInt, FindZoneInt, StoreReservation, SynchronousInt interfaces in order to call the relative components methods as helpers (it's the case of Utility Component, Zone Manager, ReservationManager) and send messages that are not return values of any Dispatcher method.

2.7 Selected architectural styles and patterns:

Let's focus now on the architectural and design decisions which guide the system composition process: first a brief discussion about the main architectural style, an overview of secondary architectural choices and rejected styles alternatives, then a description of the adopted design patterns.

Client-Server

We have organized our MyTaxiService system as a 3-tiered Client-Server architecture.

In the typical behavior of our system we have an important number of users who need an access to a set of services that has to be provided by a centralized computational unit. The best fit to these kind of problems is surely modeled by the Client-Server paradigm.

Since we have two classes of actors (potential passengers and taxi drivers) making disjointed types of requests to our server, we chose to have two distinct clients: User Client and Taxi Driver Client.

We have chosen a 3-tiered structure because, in addition to the specific tiers for client and server units, we also need to make our system communicating with the urban taxi service Database, which represents the third tier.

Taking into account the fact that Clients have to provide only graphical user interface and a very few logic to manage the interactions with the server, we decided to design them following the "Thin Client" model. In this way, the whole logic is located into the Server and Clients provide only the access to the system services.

The second tier includes an inner Database, too. It stores information about accounts, reservation to-be-served and system configuration. The reason why it is included in the same unit is the small quantity of stored data: even in the worst case, the amount of tuples in the Database will never become critical. Furthermore, this solution guarantees a high level of security on users' confidential data because their access is only located in the server tier.

This structure can be easily extended to an n-tiered architecture: indeed, we can suppose that in the future there will be the urgency to move the inner DB in another tier; in this case, the system won't change its underlying structure, because it will consider it as another external DB. For the same reason, several DBs can be added to the existing system in order to extend its functionalities; the system flexibility is guaranteed by the DB Manager which has been designed in a defensive way, taking into account the possible evolution of the system.

Secondary architectural styles

Our system design also follows the Model-View-Controller style. In our model we have a clear separation between the View, entirely implemented in the Gui module of the Client, the Controller, provided by the Server, and the data Model that is completely decoupled to the rest of the application by the DatabaseManager.

We have chosen to deploy our system without using Cloud computing style. We have considered that both storage capability and processing power will haven't a very high upper bound during the system execution and so the use of dynamic resources seems to be unnecessary; on the other hand we want to make our software as flexible as possible, therefore if in the future the amount of required resources will be subject to unpredictable variations then it won't be difficult converting our system implementation in according to the principles of Cloud computing.

Rejected alternatives

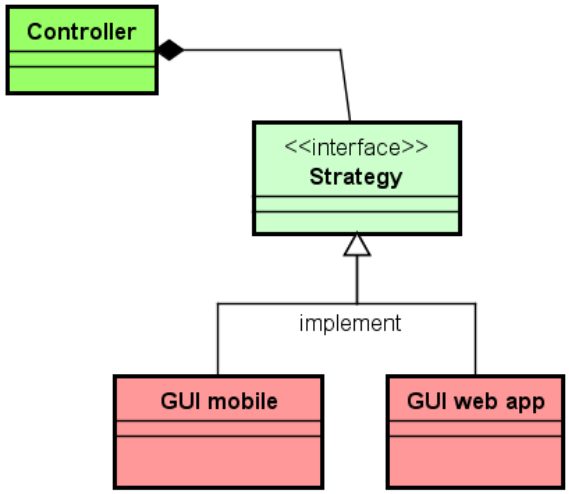
To make clearer the choice of the Client-Server as main architectural style we want briefly discuss the comparison with others two rejected alternatives: Peer-To-Peer and Publish-Subscribe.

Basically, we have considered that P2P cannot be the main architectural choice for our kind of problem because the interactions between the different users are managed by a centralized unit (Server) during the entire session time. Instead, if we consider our Server only as initializer for direct communication channels between Users and MyTaxiDrivers, a P2P implementation will be possible (the Skype model for example); but this solution doesn't fit with the committee needs and so we have rejected this alternative.

The Publish-Subscribe style was also rejected as main style for our system architecture. We have considered that a pure Event-Based system implementation isn't sufficient to cover all the functionalities of MyTaxiService. But there's in our application an important behavior that looks like an implementation of this architectural pattern: the request managing provided by the Dispatcher module (more detailed explanations in the section below: 2.8 Other design decisions).

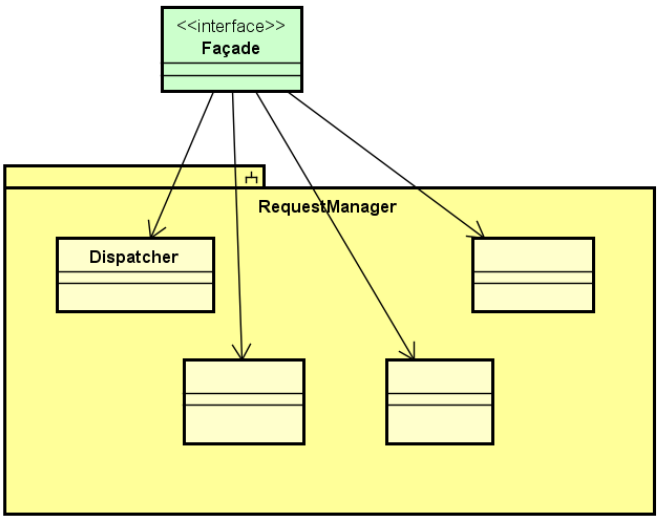
Design Patterns

Strategy Pattern:



The Strategy design pattern is a behavior pattern that changes at runtime the class behavior (by different algorithms implementation). In our system, the controller uses strategy interface to call the functions provided by Gui Component. The advantage of use Strategy Pattern is that the controller calls methods of Strategy Interface without being aware of the concrete implementation, Gui Mobile or Gui Web. We choose to use a common interface between Gui Mobile and Gui Web app because they are called using same methods but in concrete they have different implementation on the Client side.

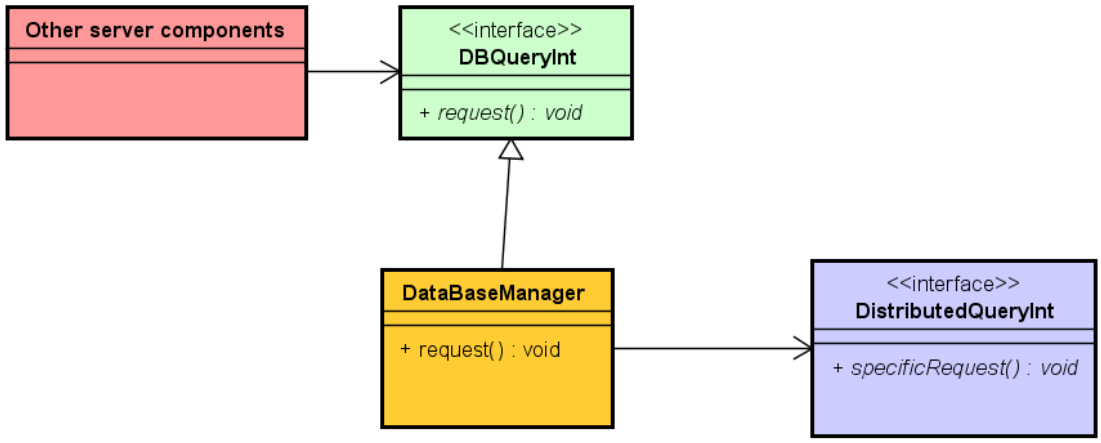
Façade Pattern:



Façade pattern is a structural pattern and it adds an interface to subsystem to hide its complexity and to make the subsystem easier to use.

We use this pattern for Request Manager Subsystem that is a complex component of Server. In particular, it allows us to hide all the inner dependencies of the component making only the provided function calls visible to the other subsystems.

Adapter Pattern:



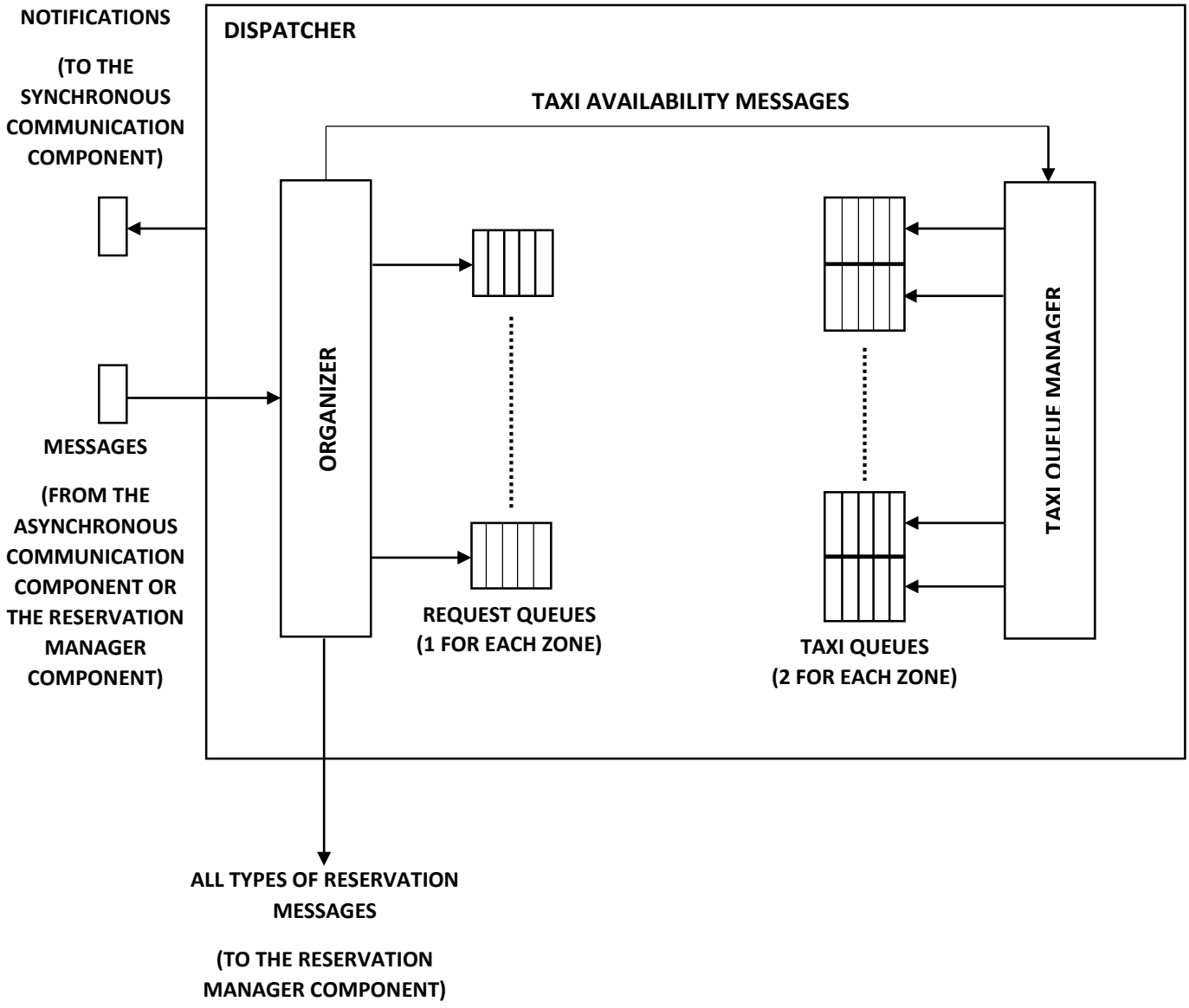
The adapter pattern is a structural pattern, which allows different interfaces of different classes to work together through the Adapter, an intermediate Object.

We use this pattern for modeling the communication between Server components and the external Database, sure enough they need an Adapter Object, the DataBaseManager, that hides the interface of external DataBase and manages the calls to this data model.

Any further modifications on either DataBasaManager component or external DBMS will not affect signatures provided by DBQueryInt.

2.8 Other design decisions

Dispatcher Design



Inspired by the "Publish & Subscribe" concept, we thought the Dispatcher component as an Event-Based subsystem: in this way, the message type is not relevant because its managing is mandated to an inner component, the Organizer.

Therefore, the Dispatcher manages the whole communication traffic in the server, both input and output.

Depending on the result of the classifying operation done by the Organizer component, messages are dispatched and sent to the other units in the Request Manager:

- In case of request message, the Organizer determines the origin zone and sends it to the relative message queue;
- In case of taxi availability message, the Organizer sends it to the Taxi Queue Manager subcomponent;
- In case of reservation message, the Organizer delegates its managing to the Reservation Manager component;

Why Publish & Subscribe: studying the system behavior, we noticed that users making requests for a taxi in a certain zone is very similar to the "Publish" action performed by the same actors.
Indeed, if we think about the origin zones as "topics", we can make as many queues as the number of zones in which the requests are stored, waiting for an answer.
So the user declares his interest in any taxi serving a certain zone (represented by the request) and waits for an event notification (an answer).
In the same way, the taxi driver declaring his availability in a certain zone can be seen as a "Subscribe" action, because in other words he's declaring his interest in receiving requests from only the zone whom he's serving.

It's obvious that we can't design the component using the "Publish & Subscribe" style bolt, but we think it could be useful starting designing it from the basic idea underlying this style.

The Taxi Queue Manager subcomponent should manage two taxi queues for each zone: one is devoted to taxi serving normal requests and another to taxi serving shared requests. In this way, we can separate the specific behaviors in the Dispatcher.

All the methods concerning the searching of an available taxi for each request are owned by the Dispatcher instance: it's the only unit having the complete knowledge about the subsystem.
The way in which it performs these operations satisfies the set of rules and constraints analyzed in the RASD document and formalized in the set of requirements.

Inner DB (E-R model)

The representation of the data storing in the inner DB is modelled by the E-R model described in previous chapter.

One thing to be notice is the absence of entities dealing with requests: we think that, in this first version of the system, looking at its functionalities, it's not necessary to store these elements because the Dispatcher works almost as a "Send and forget" component. Of course, in a possible future in which the system will be able, for example, to manage payments and so on, the request storing will be required. This is not a problem, because it can be easily developed adding entities and consequent relationships to this E-R Model

3) ALGORITHM DESIGN

3.1 SetupZone Algorithm

Introduction: a genetic algorithm

We present now the way in which our ZoneManager setups the partition of the city area.

The algorithm purpose is to create a set of zones that covers the entire city area avoiding overlap of single zones: a point in the city area (represented by its coordinates) has to be mapped to one, and only one, zone. We want also to guarantee a maximum travel time between two points of the same zone lesser than 15 minutes.

Our algorithm should be able to: compute again a valid set of zones at every Server start up, compute the set of zones independently from the underlying city map. In this way our System has a high level of reusability: if another committee will ask to us an application to manage the Urban Taxi Service (in a different city), nothing in the System will be changed except the input data for the SetupZone algorithm.

For the reasons described above, we have decided to design a “genetic algorithm”. This kind of algorithm guarantees to find a good solution to our problem, simulating the behavior of biological evolution process.

Preliminary operations

We have to initialize a particular input data set before the call of the recursive genetic algorithm.

An operator has to insert, manually, in the Database (in the corresponding ZonePoint table) a set of point that constitute the frontier of the area to-be-mapped. These points has to be inserted in pair with the same latitude or the same longitude.

Starting from these input data, the application generates a reticulum of points, called “primary points”. The reticulum is generated tracing segments between each pair of points (with the property previously described). Then the System generates a second reticulum of points, thicker than the first one, that constitute the set of “internal points”. This set represent a discretization of the entire city area.

General genetic algorithm structure

We present the generic structure of a genetic algorithm reporting the material belonging to A.Bonarini (Professor of the Soft Computing course at Politecnico di Milano):

1. *[Start] Generate random population of n chromosomes (suitable solutions)*
2. *[Fitness] Evaluate the fitness f(x) of each chromosome x in the population*
3. *[Test] If the end condition is satisfied, return the best solution in current population, otherwise*
4. *[New population] Create a new population by repeating the following steps until the new population is complete*
    a) *[Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)*
    b) *[Crossover] With a crossover probability cross over the parents to form new offspring. If no crossover was performed, offspring is the exact copy of parents*
    c) *[Mutation] With a mutation probability mutate new offsprings at each locus*
    d) *[Accepting] Place new offsprings in the new population*
5. *[Replace] Use new generated population for a further run of the algorithm*
6. *[Loop] Go to step 2*

Introduction to Genetic Algorithms © A. Bonarini

Algorithm features description

- Solution encoding
    - Each primary point is associated to a positive natural number (zero not included);
    - Each zone is defined by a sequence of distinct numbers (at least two): every pair of consecutive numbers represent a zone side, the last number in the sequence is coupled with the first;
    - A solution is represented by a sequence of zones;
    - Zero separates different zones in a solution;
    - Example: 0 1 7 4 0 2 7 8 9 12 0 6 9 11 7 4 0….;

- Constraints
    - An internal point shouldn't be included in more than one zone;
    - Every internal point has to be covered by a zone;
    - The extension of each zone has to be $2km^2$ (with 20% tolerance);

- Mutation definition
    - Right-shift of a zero in the sequence (the first and the last zeros cannot shift)
    - Example: 0 1 2 3 0 4 5 6 7 0… -> 0 1 2 3 4 0 5 6 7 0…;

- Crossover definition
    - Multi point crossover (the number of points bounded by the minimum occurrence of zeros in the two parents sequence);

    - Example: 0 1 2 3 0 4 5 6 0 7 8 9 0, 0 10 11 0 12 13 0 -> 0 1 2 3 0 12 13 0 7 8 9 0;

- Fitness Function definition
    - The fitness of a solution is the max(maximum travel time between two internal points of each zone existent in the solution);

- End condition
    - The execution ends when a solution with fitness value lesser than 15 minutes is found;

3.2 CheckSameDirection Algorithm

We introduced the idea underlying this algorithm in the RASD Document (in chapter 1.2: "Assumptions"), written such in a procedural way. Now we want to formalize it using pseudocode.

Inputs

CR – Current shared route defined for the selected shared taxi;

DP – Destination point of the new request with sharing option; its validity has already checked;

Pseudocode

//computation of the new request route, if it was a normal request, and its waiting time

NR := computeRoute(CR.departurePoint, DP);

WT := computeWaitingTime(CR.departurePoint, DP);

//computation of the shortest route such that contains CR and reaches DP, and its waiting time

NR' := computeRoute(CR.departurePoint, CR.destinationPoint, DP);

WT' := computeWaitingTime(NR'.departurePoint, NR'.destinationPoint);

//check of the validity of the new waiting time

*If(WT' <= 1,1*WT) then return true;*

else return false;

The algorithm should use the computeRoute() and computeWaitingTime() methods defined in the UtilityInt interface, provided by the Utility Component.

The aim of this algorithm is to check the validity of the property and nothing more; for this reason, its return value is a Boolean representing the check outcome.

3.3 ComputeFees Algorithm

This algorithm computes the total fee and partial fees of each passenger in sharing taxi.

The purpose is to have a fee convenient for passengers and for taxi Drivers.

The totalFee is calculated on the number of kilometers of final sharing route to consider the number of extra km covered with the adding of shared passengers.

The company can add an increment to the cost of the shared ride, calculated on the total shared route, and can change this increment whenever it wants: decrease to do advertising campaign of sharing taxi or increase to reward taxi driver.

On the other side, the passenger can split his fee with other passengers. The partial fee is calculated on the number of km that the passenger would done with the single ride , equal to the length of the shortest route from departure address to destination address. We consider the comparison of this number of km with the sum of kilometers of shortest route of each passengers. In this way, the passengers never pay more than the single ride cost and balance the loss of time due to the detour of route.

When no passengers adding to the shared ride, there is only a passengers and the TotalCost is equal to the single ride fee and also to the partial fee.

For this algorithm, we need to know:

- The number of total km of final sharing route (given by utilityInt of the Utility component)
- The number of km of the shortest route from departure address and destination address of each passengers (given by utilityInt of utility component, computeRoute method)
- Number of passengers of the shared Taxi
- The company rate per km saved in InnerDataBase(can be taken with PricePerKmImport method of DbQueryInt of DataBaseManager Component)
- The increment of sharing ride decided by the Company(can be taken with SharingIncrementImport method of DbQueryInt of DataBaseManager Component)

Variable	Explanation	Unit of measure
TOT	km sharing route	[km]
N	number of passengers	
[KM1,KM2,KM3,KM4]	array of km of each passengers , max 4 passengers	[km]
P	rate per km	[€]
C	increment of sharing Ride	[%]

TotalFee: {TOT[1+c*(n-1)]}p

PartialFee of passenger i: $\frac{KM_i}{\sum_{j=1}^{n} KM_j}$ *TotalFee

4) USER INTERFACE DESIGN

For the user interface design we have only improved what we have already presented in the Requirements Analysis and Specification Document (3.Requirements, 3.2.Non Functional Requirements, 3.2.1.User Interface).

In particular we have updated our mockups adding a dynamical behavior and fixing some errors. The update is available as attachment to this Document.

5) REQUIREMENTS TRACEABILITY

In the following section we report a mapping between each functional requirement defined in the RASD document (chapter 3.1 Functional Requirements) and the corresponding set of System components that guarantees its satisfaction. We have decided to not report every component involved in the requirement satisfaction, but only the most significant set.

Requirement	Components
The System should provide to the Guest a graphic interface containing a sign up form	UserClient-Gui
The System should accept a sign up request only if the Guest has filled all the mandatory fields in the sign up form	UserClient-Gui
The System should provide to the Guest a graphic interface containing a log in form	UserClient-Gui
The System should accept a log in request only if the User has inserted into the log in form a couple (mail, password) that match with the same couple of a corresponding registered User	ConnectionManager DatabaseManager
The System should consider a taxi request only if the User has inserted: **a) A valid departure address or, in alternative, has enabled the gps option for the current position** **b) Number of passengers in the range [1,4]**	UtilityComponent UserClient-Gui
	Dispatcher

The System should give a positive answer to a request only if a MyTaxiDriver has accepted to take care of it	
The System should give a positive answer to a request only if the position of the MyTaxiDriver, that has accepted to take care of it, corresponds to a waiting time less or equal twenty minutes	Dispatcher UtilityComponent
The System giving a positive answer to a request has to notify waiting time and ID Code of the coming taxi to the requesting User	Dispatcher UtilityComponent CommunicationServer
The System should give a negative answer to a request if there isn't a MyTaxiDriver who has accepted to taking care of it	Dispatcher CommunicationServer
The System should give a negative answer to a request if the position of the MyTaxiDriver, that has accepted to take care of it, corresponds to a waiting time greater than twenty minutes	Dispatcher UtilityComponent CommunicationServer
The System should provide a graphic interface that allow Users to request a reservation	UserClient-Gui
The System has to consider a reservation request only if the User ha inserted: departure time, departure address, destination address, number of passengers in the range [1,4]	UserClient-Gui
The System has to consider a reservation request only if the request has valid departure address and destination address	UtilityComponent
	UserClient-Gui

The System should: accept a reservation request only if the request is submitted at least two hours earlier than the departure time	
The System reserve a taxi ten minutes earlier than the departure time	ReservationManager
The System should provide a graphic interface that allow Users to access his list of hanging reservations	UserClient-Gui ReservationManager
The System should allow Users to delete his reservation until 10 minutes before the ride time	ReservationManager
The system allows Users to select the sharing option on every type of requests (simple or reservation), in order to enable the sharing of taxi rides	UserClient-Gui
The system reserves the same taxi for one or more Users having done a request with active sharing option if and only if they want to leave from the same zone and they are headed to destinations that are in the same direction (see the "same direction statement" assumption)	Dispatcher UtilityComponent
The system adds further Users to a shared ride if and only if the related taxi has enough free seats	Dispatcher
The system adds further Users to a shared ride if and only if the myTaxiDriver who takes care of it hasn't declared the departure to the system yet	Dispatcher
The system has to compute the route of the shared ride and notify the myTaxiDriver about it	UtilityComponent CommunicationServer

The system computes the total cost of the shared ride and all the partial fees that each User has to pay	UtilityComponent
The system sends the payment information to the myTaxiDriver and all the Users sharing the ride	CommunicationServer Dispatcher
The system displays a sign up form and lets the TaxiDriver view it	TaxiDriverClient-Gui
The system approves the registration if and only if the ID Code is valid and it doesn't belong to another TaxiDriver	ConnectionManager DatabaseManager
The system provides an interface that allows a myTaxiDriver to signal his availability	TaxiDriver-Gui
The System should provide to the MyTaxiDriver a graphic interface containing a log in form	TaxiDriver-Gui
The System should accept a log in request only if the MyTaxiDriver has inserted into the log in form a couple(IDcode, password) that match with the same couple of a corresponding registered TaxiDriver	ConnectionManager DatabaseManager
The system sends requests only to available myTaxiDrivers	Dispatcher
The system informs the available myTaxiDrivers about new requests by sending notices to them	Dispatcher CommunicationServer
The system provides an interface that allows myTaxiDrivers to accept or deny a received request	TaxiDriver-Gui

The system lets a myTaxiDriver delete a request only if it's not a reservation request	TaxiDriver-Gui
When a myTaxiDriver declares his availability, the system adds his taxi ID Code in the last position of the taxi queue related to the zone in which he's located	Dispatcher
When a request occurs, the system sends it to the first taxi stored in the queue, where the queue is related to the same zone to which the request belongs; then the system wait for his answer	Dispatcher CommunicationServer
If a myTaxiDriver doesn't answer to the request within 1 minute from its notification, the system moves him to the last position in the queue	Dispatcher
If a myTaxiDriver confirms a request, the system pops from the queue	Dispatcher
If a myTaxiDriver rejects a request or doesn't answer to it within 1 minute, the system moves him to the last position in the queue and repeat the procedure	Dispatcher
The System should provide fees in order to guarantee a fair distribution among the Users	UtilityComponent
The System should provide fees in order to guarantee extra earnings to the MyTaxiDriver	UtilityComponent

6) REFERENCES

The tools we used to create the DD document are:

- Microsoft Office Word 2013: to release and format document
- Balsamic Mockups3: to draw interface
- Astah: to draw sequence diagram, component and deployment diagram

Number of hours

We have spent 43 hours per person to redact the DD document.

Greta Ghiotti: 43 hours

Raffaele Malvermi: 43 hours

Mirco Mutti: 43 hours