MyTaxiService Design Document - Software Engineering 2 project

Gabriele Giossi (id: 854216)

v1.0 04/12/2015

Contents

1 Introduction

1.1 Disclaimer

This Design Document (DD) is meant to provide a deeper understanding of the system that is being developed, showing from an architectural and algorithmic point of view how the application that is designed has been conceived to work; it is to be consulted upon any attempt at modifying or implementing new functionalities, or to explain the inner working of the system and how its components and their interactions have been conceived; every change to this document must be reported in the Changelog subsection of the Appendix chapter; as it has been formalized and redacted almost in parallel with the Requirements Analysis and Specification document, some of the general sections are very similar between this document and the aforementioned one; it is meant to provide all the necessary background information in a single document without having to consult the other every two minutes.

1.2 Purpose

The purpose of MyTaxiService is to provide a reliable, performing and easy-to-use tool so that a city's taxi service is optimized with regard to taxi ride and queue management and provide a base program that can be improved by means of APIs in order to meet the specific requirements of the context it is operating. The application is designed to be implemented and used in any city that has a taxi service, and is meant for use by both taxi drivers and passengers.

1.3 Scope

MyTaxiService will be a web-based application, supported both on smartphones and PCs; its aim is, as said, optimizing the taxi service of a city: the system divides the city in which it operates into "zones" of about 2 square kilometers, and to each of them computes a taxi queue

When a passenger, by means of the application, requests a taxi ride, the system assigns an available taxi, the first in queue near the starting point of the ride; the taxi driver will confirm a ride before actually carrying it out.

Some of the benefits of implementing the MyTaxiService system are that it is possible to achieve a real-time balance of numbers of rides requested in a given zone with respect to the available taxis in a zone, providing taxi drivers with an almost-real-time response to spikes of ride requests, improving service quality while simplyfing the interaction of requesting a taxi ride and finding a taxi for any passenger, wherever in the city he might be.

1.4 Definitions, Acronyms, Abbreviations

- Users: any human actor that interfaces with the app in order to request or carry out a ride, in short passengers and taxi drivers;

- System: the server-side portion of the application that manages queues, assigns taxis to requests, tracks the cab GPS systems, and manages/stores user data;
- RASD: Requirements Analysis and Specification Document, the document that formalizes the requirements of the system to develop; it will be referred to in several points of this document;
- HTTP: HyperText Transfer Protocol, the cornerstone of the web as we know it, it is the procotol used to send and receive requests that MyTaxiService will need to interface with in order to be a proper web-based service;
- DBMS: DataBase Management System, a system that is used to organize, interrogate to retrieve and present information retrieved from an arbitrarily large organized data amount;
- GPS: Global Positioning System, a triangulation-based system to provide real-time localization in the world of an entity;
- API:Application Program Interface, a set of routines, protocols and tools for building software applications;
- ADT: Abstract Data Type, a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

1.5 Reference Documents

The documents referred to in this documentation, that are to be consulted whenever there are inconsistencies or the need of a clarification, are the following:

- The Requirement Analysis and Specification document for MyTaxiService, available on the same GitHub repository as this one;
- The informal specification document for MyTaxiService by the client, available on the GitHub repository aswell, for reference;

1.6 Document Structure

This document is structured as follows:

- Section 2 will highlist all possible architectural design choices and aspects for the application; it is expected that, during development, this section will be the one that will be subject to the biggest amount of changes during development;

- Section 3 will present a handful of interesting algorithms and/or implementations of critical parts that are present in the application (mostly in pseudo-code);
- Sections 4 and 5 will create a bridge between this document and the Requirement Analysis and Specification Document, explaining how the mock-ups and goals that were stated in the RASD map into the design choices of this design document;
- Section 6 provides all references to algorithms, tools, and so forth;
- Section 7 will hold the changelog of the document, the list of used tools to redact it and the amount of work hours put into creating it;

2 Architectural Design

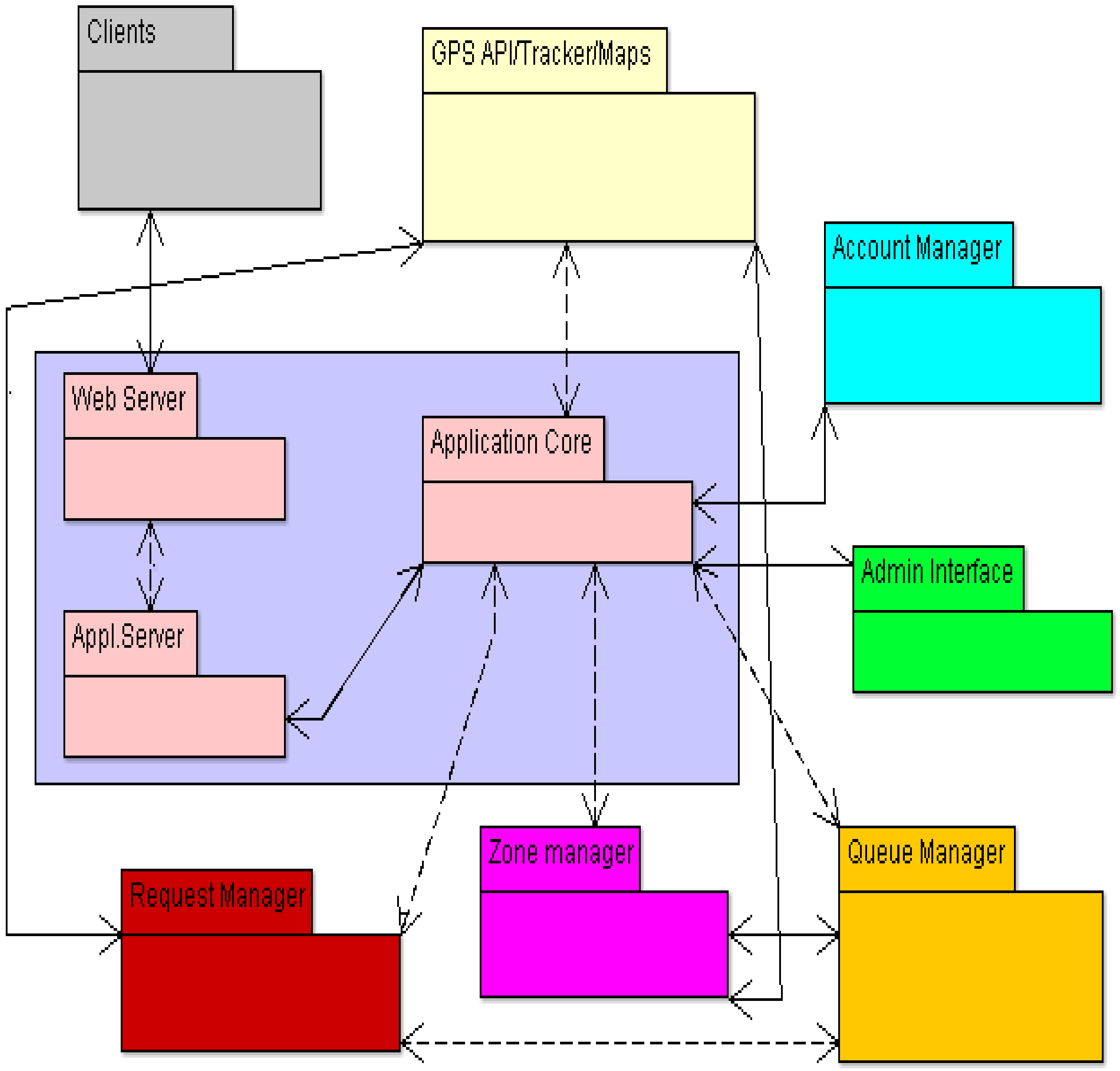
2.1 Overview

The first assumption that is made in the architectural design of MyTaxiService is that, being this a web-based application, will have to work based on the widespread general architecture of a web application: a web server and an application server, where the web server is dedicated to receiving HTTP requests, forwarding them to the application server, receiving responses from it and forwarding them to the intended destination client; in short, it works as a "communication channel" that forwards requests and responses between different application modules. The application server will handle calling the necessary application executables, sending data and receiving data, and formatting the responses to the requests it receives; It will be clearly shown in the next section, which will present a general, high-level view of what has been conceived as a possible application architecture.

The server side of the application is conceived as to be deployed by using a Java Enterprise Edition platform, thus relying on all Java EE Web based technologies and development methods; the application and web servers that are thought to be good and reliable enough to operate the application are those of the Apache Tomcat family, that are rather easy to configure and implement, using a Java EE development IDE such as Eclipse or NetBeans.

The clients of the application are either based on PCs or smartphones; the PC client can be implemented by housing web pages that through servlets, Java Server Faces, and beans interface with the application and provide all functionalities even on notebooks or desktop PCs and can be done via the same Java EE development IDE mentioned earlier on; the mobile versions are to be developed using resident development methods and tools for any and all mobile operating systems that we want the application to be deployed and made available on.

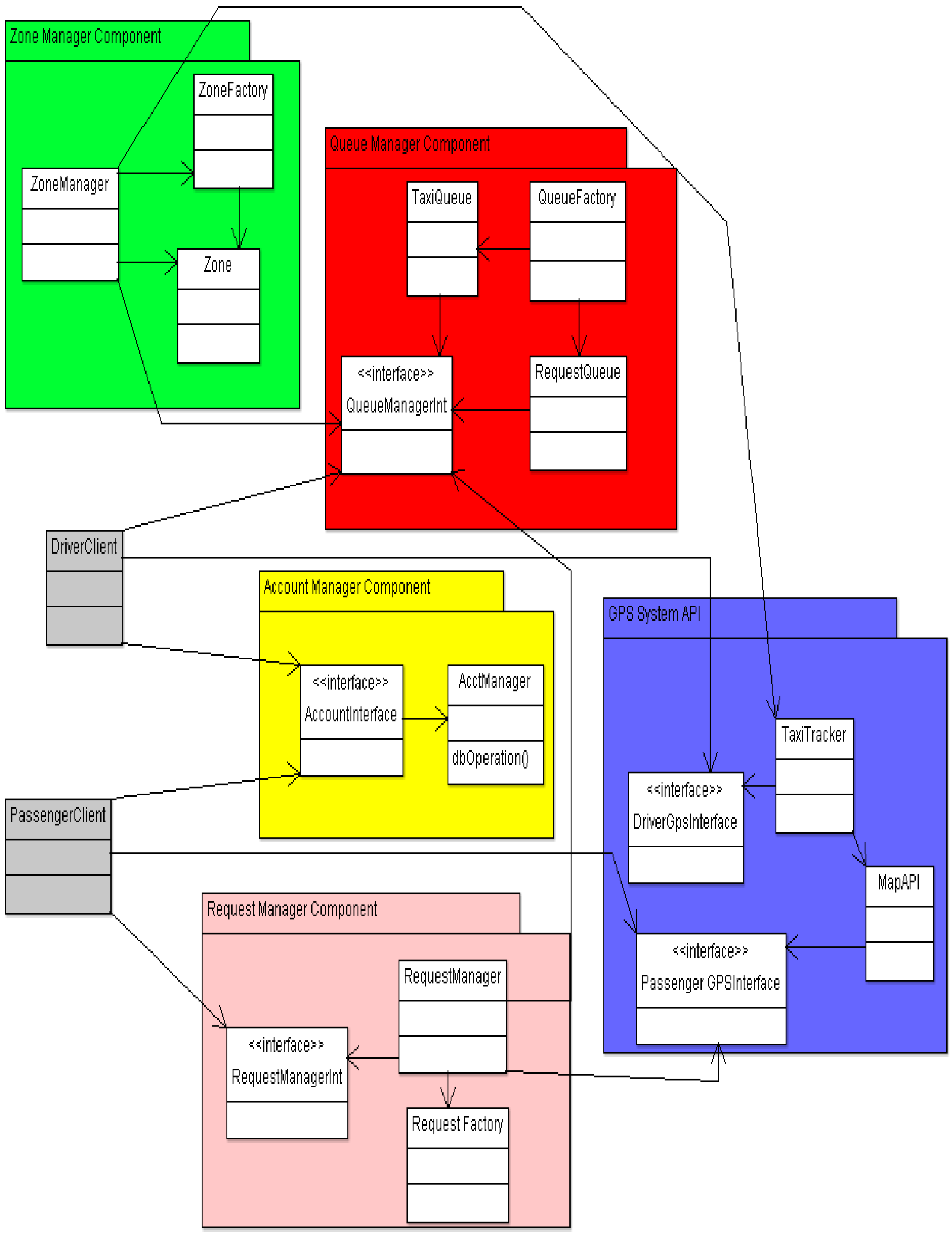
2.2 High Level Components and their Interaction



The diagram above, while being only a high-level view of the system, shows very clearly how it has been conceived: the two servers are, in fact, monolithic, and we could consider the web server to be some sort of "meta-client" that receives requests from all clients connected and forwards them to the application server, that does its “magic” by calling all the necessary components to do some sort of operation, receives the results and sends back to the web server a response to be forwarded to the client that made the original request. The application core will be in charge of receiving from the application server all the necessary function calls and undertake the necessary steps; for all purposes, many details will be explained further in other diagrams in this document, while others will be simply described in words.

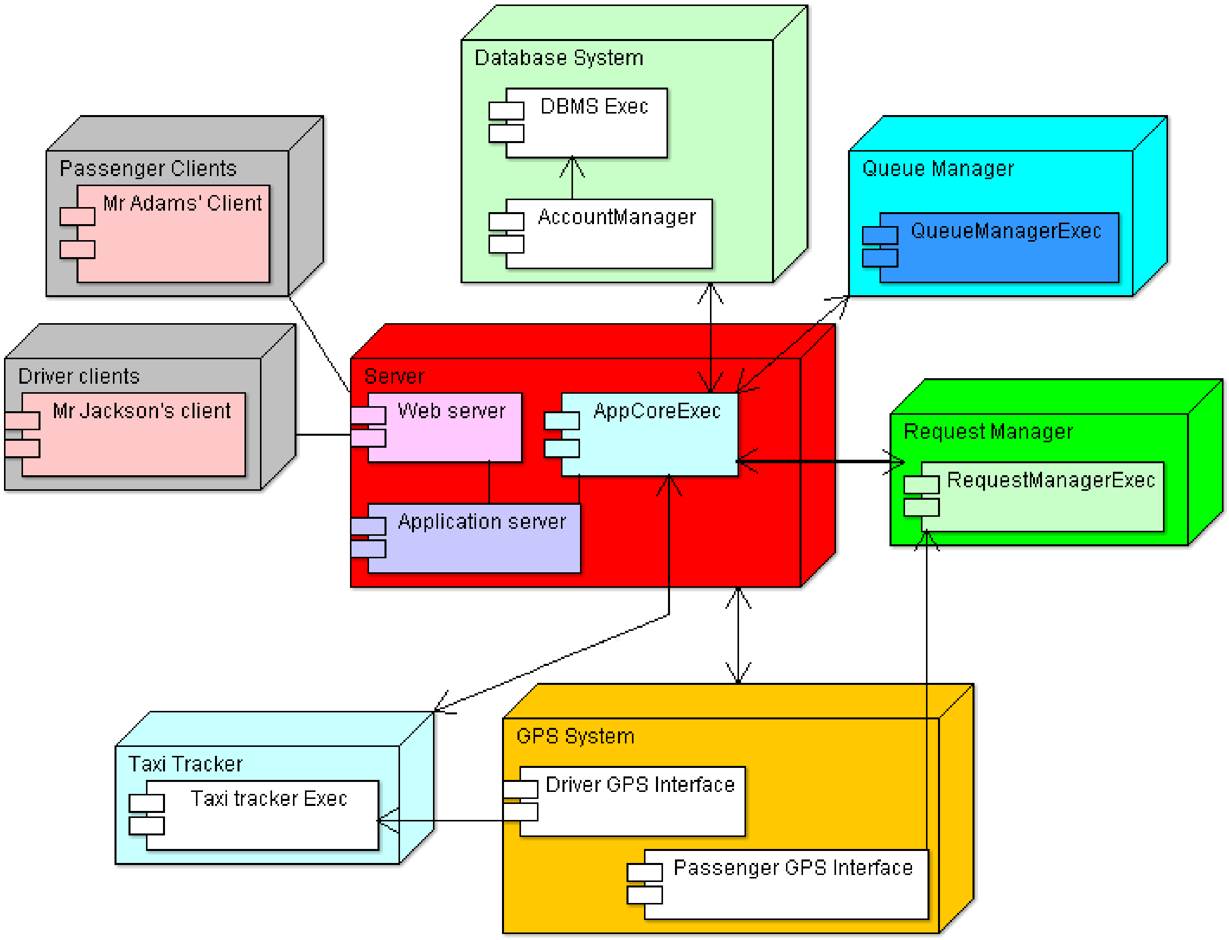
2.3 Component View

The following diagram will focus on the component view and interactions related to clients in the application; all operations are, obviously, supervised by the application server that receives from the web server the requests from the client, even though for the sake of simplicity the server here, including the central application core module that has been inserted in the “server” component has been omitted:


Zone Manager Component
ZoneFactory
ZoneManager
Zone
Queue Manager Component
TaxiQueue
QueueFactory
<<interface>>
QueueManagerInt
RequestQueue
DriverClient
Account Manager Component
<<interface>>
AccountInterface
AcctManager
dbOperation()
GPS System API
TaxiTracker
<<interface>>
DriverGpsInterface
MapAPI
PassengerClient
<<interface>>
Passenger GPSInterface
Request Manager Component
RequestManager
<<interface>>
RequestManagerInt
Request Factory

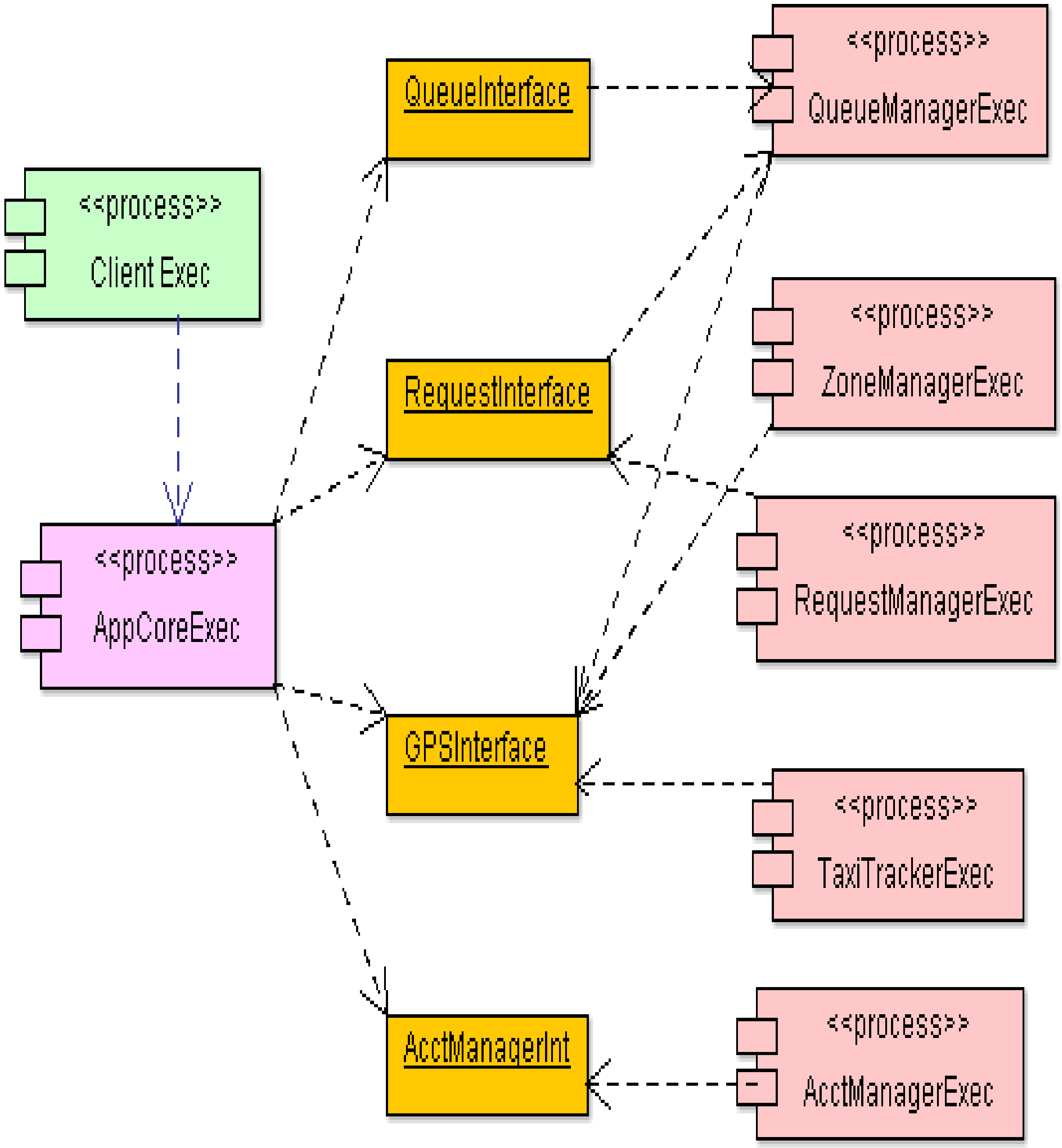
2.4 Deployment View

The deployment view will show deployed modules and their interactions: it has been decided to formalize this diagram with the primary intention to show and stress how every communication and request done by the clients is processed by the server, as shown in the diagram below:



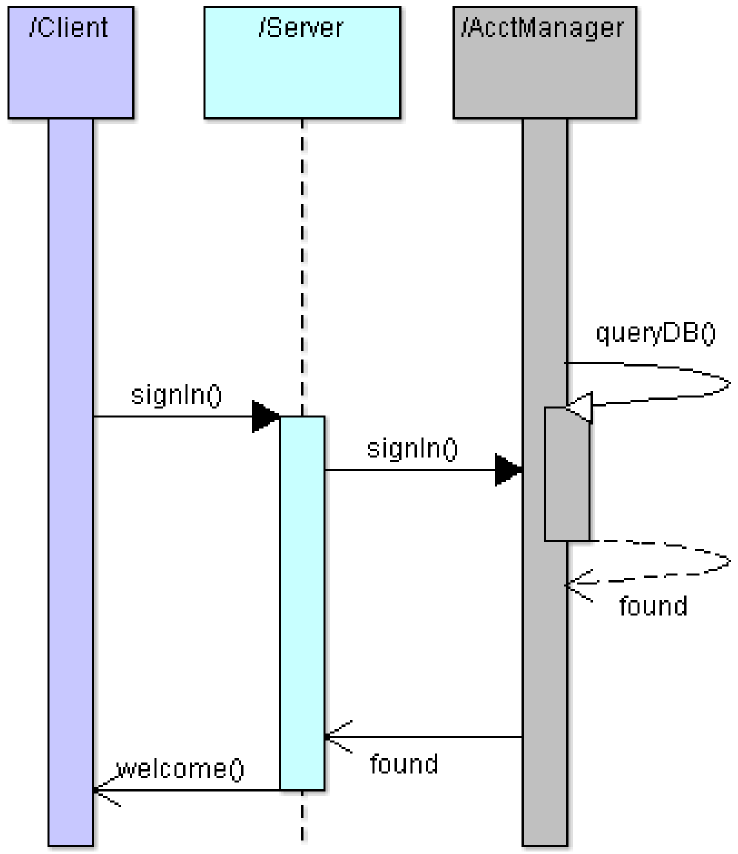
2.5 Runtime View

General runtime view of the system and its executables:

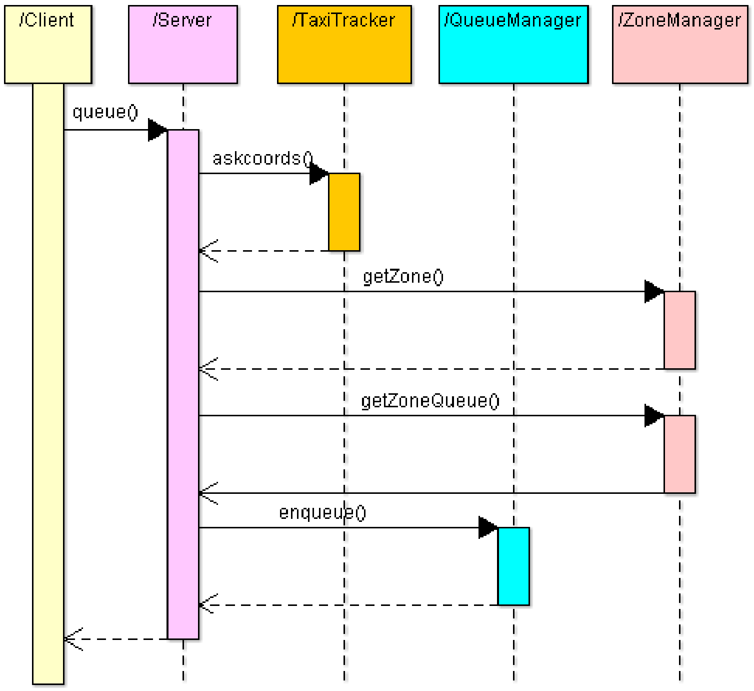


In addition to this, some of the most important and fundamental use cases will be explained in their interaction between modules and components through sequence diagrams (refer to the RASD's Use Case diagram for further details of these use cases):

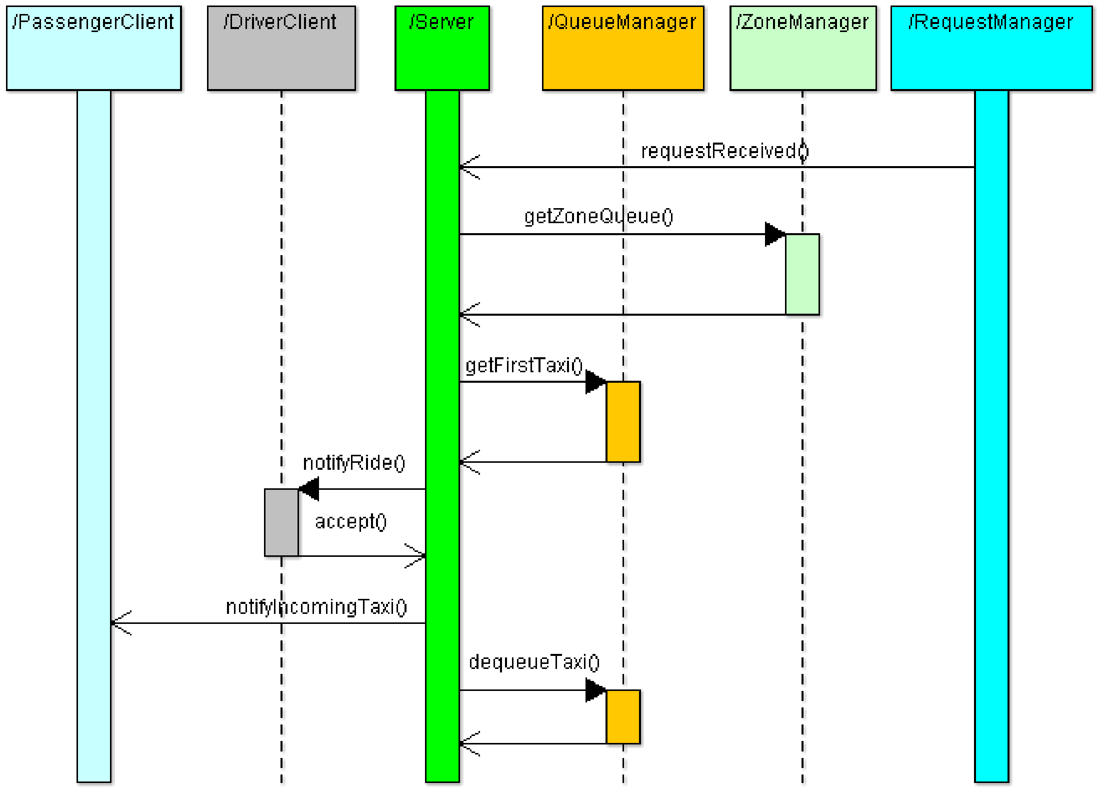
2.5.1 "Sign in" use case:



2.5.2 "List Availability" use case:



2.5.3 "Assign Taxi to Request" use case:



2.6 Component Interfaces

Here will be listed and detailed all the interfaces that are present for each component, what they are supposed to do and several details that allow a precise implementation:

- Account Interfaces: This interface will provide log-in, log-out, sign-up, and all account-related functionalities that will also need to interface with the database, moreover, it will provide references to clients that need notifications for the server;
- GPS Interfaces: These interfaces are split in two based on the type of client's logged user accessing them: the driver GPS interface will track the drivers' GPS and update its zone reference whenever needed, while

the passenger GPS interface provides methods which allow interfacing with the Map API for pointing out the requested drives' start and end point;

- Zone Interfaces: The zone manager interface provides mostly getter methods for retrieving a particular zone's queue, or edge coordinates, and similar;
- Queue Interfaces: The queue manager interface will provide all methods to manage, create, clear, destroy and in any other way manage both the taxi request queue and the taxi queue for each zone;
- Request interfaces: The request manager interface will provide methods to manage requests, notify the servers of new requests and so forth;
- Administration interface: it is a particular interface that exposes methods only for diagnostics, statistics collection and data insertion for registering taxi drivers, as assumed in the RASD; it is to note that the interface is NOT accessible from a client, and is resident on the server machine (web page, accessible only from a specific administration-level login and terminal)

2.7 Selected Architectural Styles and Patterns

As previously stated, there has been a choice of a client-server achitectural style, based on a multiple-tier architecture with thin clients, simply communicating with the central servers and presenting the data/notifications that are received; computations, manipulation of data and such are mostly done by the server; most components managing zones, queues or any object on server-side have been designed with a factory design pattern, in order to limit the istantiation of most critical objects to a handful of precisely dedicated modules.

Below follows a list of points that ca be used as guidelines during the code implementation procedure:

- For now, it is supposed each component is, in fact, restricted to be a single entity, thus all "manager" classes in components will be subject to a singleton design pattern;

2.8 Other Design Decisions

Several of the following points have to be formalized and noted in this document, but the actual implementation and verification of functionalities will be finalized in later stages of development.

- Of course, as there are users that access to the application, a session context will have to be present, taking into account a possible implementation using Java EE, Java beans, context variables, session objectification and such will be needed in order to maintain open connections of users (specifically, drivers should not have a session expiration due to time-out because of lack of requests);

- Any other memory space besides queues is, in fact, a heap memory area, thus necessary algorithms will be needed in order to retrieve all the needed information from memory areas; plus, there will be context parameters that will be valid in the whole application context;

- The Account Manager will be interfacing with a Database, so some JDBC connection managing must be present;

- Taking into account an expected goal of availability values for all components of this system, it's expected for the system to have an availability of 3-nines to 5-nines, based on the importance of taxis in each domain (a city with less traffic might accept a smaller availability, while cities with high amount of traffic must have the highest availability achievable

3 Algorithm Design

The following section contains a possible implementation of linked queues (for managing taxi queues and request queues for each zone in the system), given the importance of maintaining a good and ordered queue and avoiding drivers or requests having to wait abnormal amounts of time due to lack of efficiency in managing it.

3.1 Linked Queue implementation

Since the queues of requests and taxis are implemented as effective queues, a C++/pseudo-code implementation of a linked queue follows below, with all declarations and methods that are relevant:

```
//Linked queue implementation
template <typename E> class LQueue: public Queue<E>{
        private:
                Link<E>* front;         //Pointer to front queue node
                Link<E>* rear;          //Pointer to rear queue node
                int size;               //Number of elements in queue

        public:
                LQueue(int sz =defaultSize) //Constructor
                        {front = rear = newLink<E>(); size = 0;}

        LQueue() {clear(); delete front; }      //Destructor

        void clear(){           //Clear queue
                while(front->next != NULL){     //Delete each link node
                        rear=front;
                        delete rear;
                }
                rear=front;
                size=0;
        }

        void enqueue(const &E it){      //Put element on rear
        rear->next = new Link<E>(it,NULL);
        rear = rear -> next;
        size++;
        }

        E dequeue(){    //Remove element on front
        Assert(size != 0, "Queue is empty");
        E it = front->next->element;    //Store dequeued value
        Link<E>* ltemp = front-> next;  //Hold dequeued link
        front->next = ltemp->next;      //Advance front
        if (rear == ltemp) rear = front;         //Dequeue last element
        delete ltemp;   //Delete link
        size--;
        return it;
        }

        const E& frontValue() const{    //Get front element
        Assert(size!=0, "Queue is empty");
        return front->next->element;
        }

        virtual int length() const {return size;}

};
```

The ADT for the factories in the various component follows below: it can be thought as an abstract superclass, that will have its declared methods implemented by the actual implemented factories:

```
public abstract class FactoryADT {

public elem createElem();        //To be implemented using override in factories

}
```

Lastly, a pseudo-code sketch of the main system initialization algorithm, in order to understand how the system, once deployed and activated, will istantiate every component and start working:

```
//Initialization procedure
void init(){
        system.loadMap();        //Loading zone mapping for the domain
        zonefactory.initialize();
        queuefactory.initialize();
        requestfactory.initialize();
        accountmanager.initialize();    //Initialization of all factories

        forall(zones){
                zonefactory.createElem();                //Creates zone objects
                queuefactory.createTaxiQueue();
                queuefactory.createRequestQueue();
        }

        accountmanager.DBconnect();     //Connect the account manager to the DB
        RMI.initialize();        //start listening on a port
        diagnostics.initCheck();         //Diagnostics after initialization
        admininterface.showDiagnosticsResult();
}
```

4 User Interface Design

In this section, the mock-ups that are presented in the RASD will be furtherly detailed from a design point of view. Refer to the RASD section presenting these mock-ups.

4.1 Client depth

As it has been stated in previous sections, this design formalization for MyTaxiService follows a "fat server, thin client" architectural style, thus the clients images that are presented in the mock-ups are strictly data-presenting; any and all functions that are effectively implemented are, in fact, requests for a precise functionality present on the server, that will take care of all data manipulation, computation, and notification back to the user of results.

This design, of course, is vulnerable to server-side faults; in fact, a severe server fault will effectively knock out the system for good; this, however, is offset by the fact that implementing clients on several completely different platforms will be easier because the clients will be far less complex than in other design choices.

4.2 Interface Design

The interface will be available on smartphones or PCs, and it will be fundamentally different:

- The PC/notebook interface will be all based on web technologies: it will be a series of web pages, thus able to be consulted with any Internet browser, and through these pages, it will be possible to interact with MyTaxiService; it will be developed and deployed alongside the server, which, when the user visits a certain URL, will provide the first page of the interface to the browser; the interaction will be based on standard browser interaction: mouse clicks, and so on;

- Smartphone and, in general, mobile interfaces will be developed in each of the native development environment of any and all software and hardware mobile platforms that MyTaxiService will need to be deployed on (for example, Goggle Android or Apple iOS); this will be a self-contained application that will be able to interface thourgh the Internet with the application's main server and receive responses, displaying them to the user; inputs and outputs will be based on mobile technologies: touch screens, keyboards and so on;

While very different in concept, the end result of these two interfaces will be the same: they will offer the very same functionalities, with slightly different final graphic design and different code implementation: whereas the PC/notebook interface will be developed using HTML, JavaScript, JSF, Beans and any other applicable and useful technologoy for web applications, the smartphone versions will be developed in the appropriate language for each platform, and will interface with the same technological infrastructure.

5 Requirements Traceability

Refer here to the Requirements listed in the RASD for reference:

5.0.1 Functional requirement traceability

- The basic requirements for the functionality of the application map exactly into the architecture shown in section 2 of the present document: a web-based, client-server architecture that communicates and allows for taxi queuing, request handling and taxi dispatching; of course, the real-world facts that have to occur are external to the context of the application, and any non-smart human user can make mistakes that won't, however, impede the general functionality of the system.
- The requirement of a server-side interface has been formalized and met, as can be seen in section 2;
- The requirement of a dedicated queue-management system and logic has been fully met, with both the queue manager accounted for in the architectural design and the queue algorithm that is presented in section 3;
- Database encryption is external to the application context and dependant on the choice of database system that is integrated into the release version of the application, whereas the clients will need to provide in their code the showing of a notification about privacy, as stated by another requirement;
- The presence of a map and GPS system and a zone manager will allow the system to perform the division of the domain city in zones, as requested;
- All functional requirements related to limit cases and special situations are to be taken care of and into account during the writing of the application's code;

5.0.2 Other Requirements traceability

- The mock-ups presented are fully replicable in web pages for the PC/notebook client;
- The same mock-ups provide all the client functionalities, hiding all the operations done server-side, but still providing all the necessary information;

6 References

The linked queue implementation is taken from the queues chapter of the book by Clifford A. Shaffer, "Data Structures & Algorithm Analysis in C++", Third Edition, Dover Publications (all credit is due to the author).

UML disagrams try to follow the UML standard to its letter, so refer to the UML standard documentation in order to clear interpretative doubts (`http://www.uml.org`).

The RASD (Requirements Analysis and Specification Document) for MyTaxiService that is referenced in multiple sections of this Design Document is available at the GitHub repository `https://github.com/GGioss/SEeng2`

7 Appendix

7.1 Group working hours

Towards the formalization of the document, the partitioning of work for the group has been so:

Gabriele Giossi: 20 hrs

7.2 Changelog

v1.0 Initial release; document expected to be modified during further development stages (04/12/2015)

7.3 Used tools

- To create and export the UML diagrams the tool used was ArgoUML `http://argouml.tigris.org/`
- To format and redact this document LyX was used.