POLITECNICO DI MILANO

Design Document (DD)

Computer Science and Engineering (CSE)

Software Engineering 2 Project

Year 2015/16

Date: 04/12/2015 – Version: 1.0

STUDENTS:

*Martino Andrea (**788701**)*

*Marchesani Francesco (**852444**)*

PROFESSOR:

Mirandola Raffaela

1. Table of Contents

2. Introduction

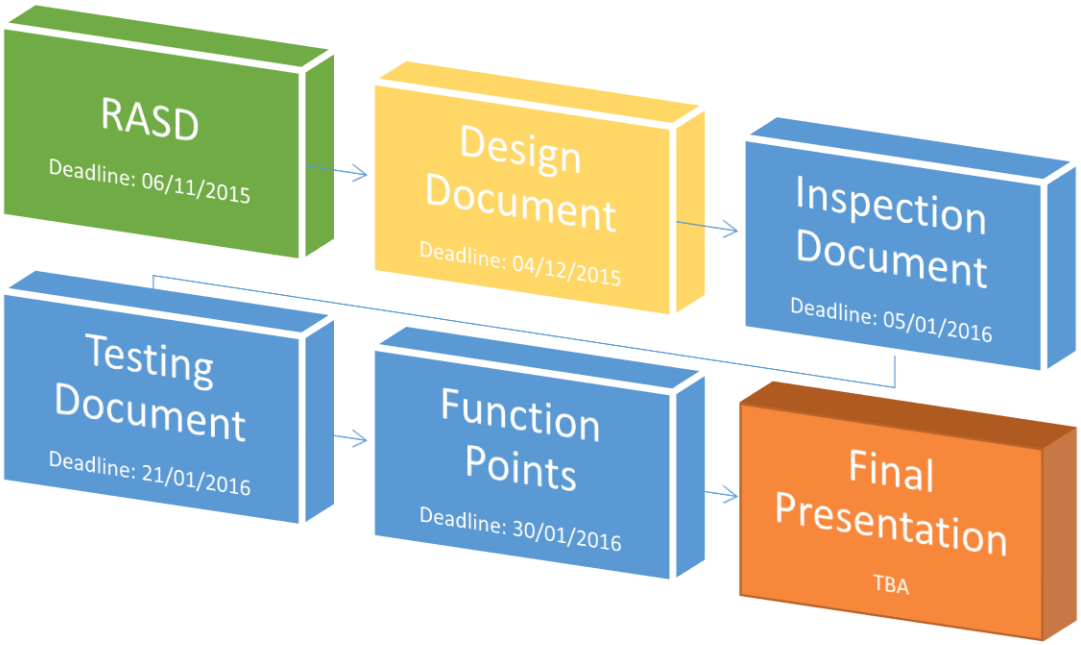
2.1. Purpose

This **Design Document** contains information about the **architecture** and other features of *myTaxiService design* such as **high-level algorithms** and **user interface** (with recalls from the previous **RASD**). In order to maintain a **modular** and **scalable** system for the future, we will enter in details only if strictly necessary (otherwise, we will keep a higher level of abstraction).

This **document** is coherent with *the official template* of the project on the *Beep platform*.

As we said for the **RASD**, it is important to underline that some parts of this document may evolve in the future (this may occurs for several causes).
Anyway, we will try to maintain coherence as much as possible.

Here is a resume of the steps of the project, with the related deadlines (in green documents already delivered, in yellow the current document):



2.2. Scope

The main scope of this **DD** (*Design Document*) is to give an overall guidance to the **architecture** of the **project**, which is *myTaxiDriver* (**Software Engineering 2 project** of year 2015/16 - **Politecnico di Milano**).

We described the main **goals** and **objectives** of the project in the previous *Requirements Analysis and Specification Document*.

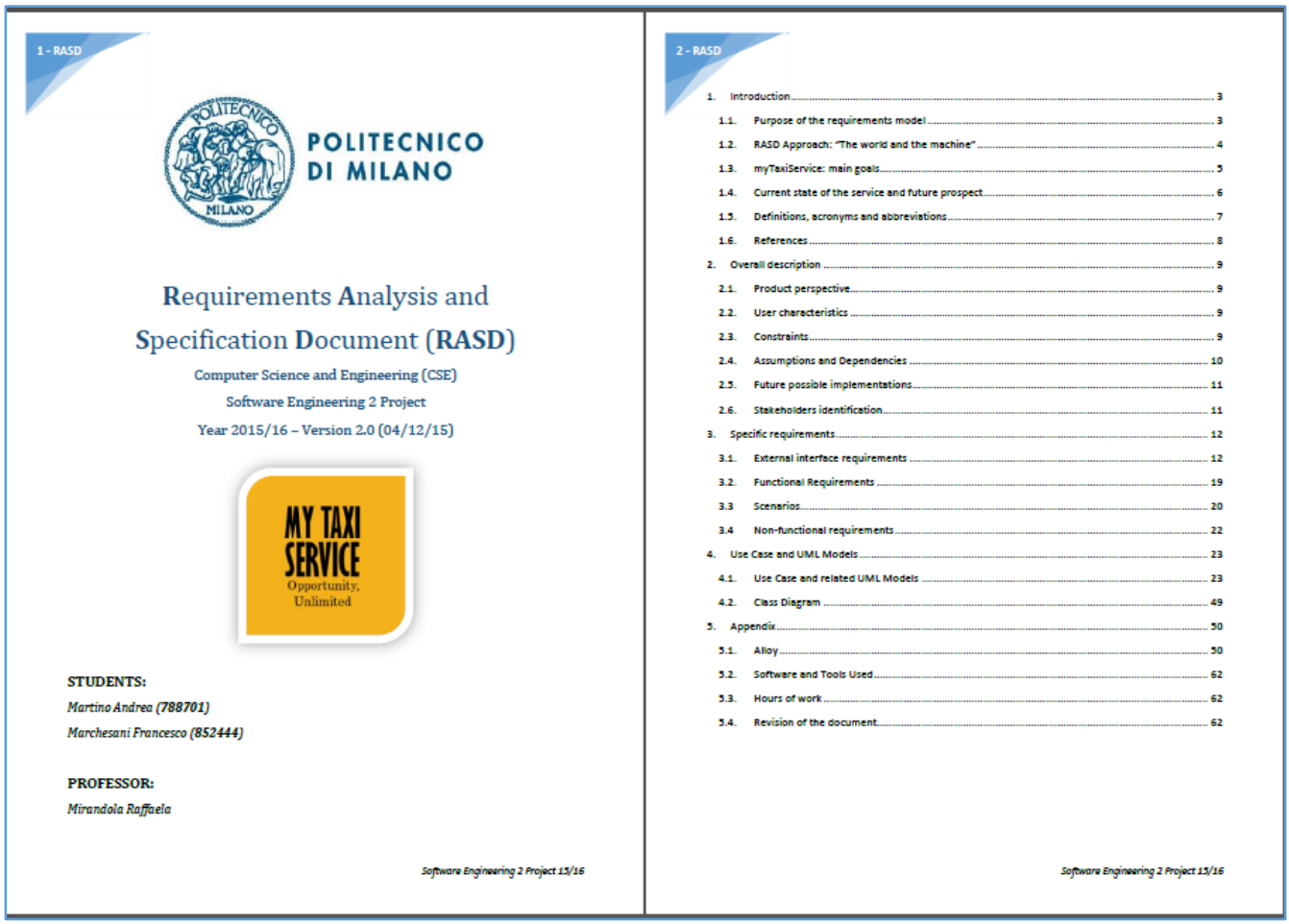
2.3. Definitions, Acronyms, Abbreviations

- **DD:** *Design Document*
- **RASD**: *Requirements Analysis and Specification Document*
- **mTS**: *myTaxiService*
- **JEE**: *Java Enterprise Edition*
- **UML**: *Unified Modelling Language*
- **CC**: *Customer Client*
- **TDC**: *Taxi Driver Client*
- **SAC**: *SysAdmin Client*
- **UI**: *User Interface*
- **UX**: *User Experience*
- **Servlet**: A servlet is a program that extends the capabilities of a server.
- **Layer**: logical level of the architecture.
- **Tier**: physical level of the architecture.
- **Markov chain**: is a random process that undergoes transitions from one state to another on a state space.
- **M/M/s**: (also known as *M/M/k*) is a multi-server queueing model.
- **PayPal**: famous online payment service.
- **Google Maps**: web-mapping service developed by *Google*.

Note: *for the full Glossary may be helpful to see also the paragraph 1.5 of the RASD.*

2.4. Reference Documents

The main reference document is the **Requirements Analysis and Specification Document 2.0 (RASD Revision)** of *myTaxiService*.



1 - RASD

POLITECNICO DI MILANO

Requirements Analysis and Specification Document (RASD)

Computer Science and Engineering (CSE)

Software Engineering 2 Project

Year 2015/16 – Version 2.0 (04/12/15)

MY TAXI SERVICE Opportunity, Unlimited

STUDENTS:

Martino Andrea (788701)

Marchesani Francesco (852444)

PROFESSOR:

Mirandola Raffaela

Software Engineering 2 Project 15/16

2 - RASD

Software Engineering 2 Project 15/16

Preview of the Requirements Analysis and Specification Document 2.0 (RASD Revision), our reference document.

2.5. Document Structure

The **Document Structure** completely follows the template hosted on the *Beep Platform* (PDF file *DD TOC – Design Document Template*).

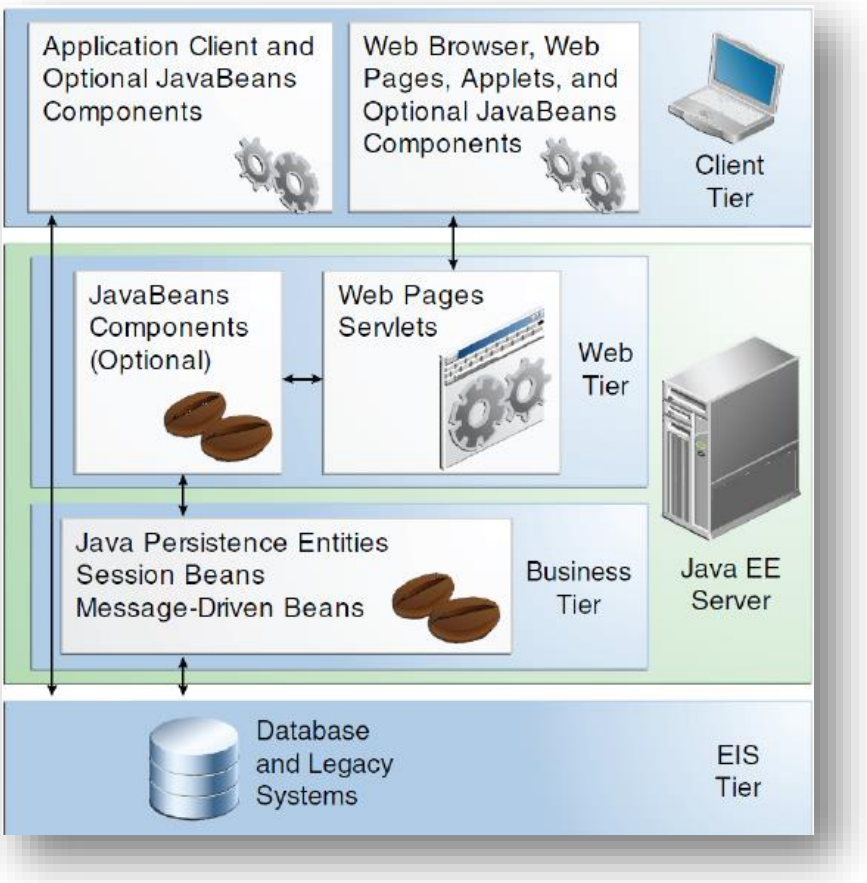
3. Architectural Design

3.1. Overview

We start from a big consideration: *myTaxiService* is a big project. As we have seen before, in the RASD, we have a lot of potential users and a big amount of requirements.
Thus, we considered to develop our application using **Java Enterprise Edition** (**JEE**).
This will also be useful to satisfy important *Non-Functional Requirements* such as scalability, portability, availability, reliability and so on.

The applications of *myTaxiService* (the web application and the mobile one) will be *large-scale*, *multi-tiered, scalable, reliable* and the network will be *secure*.
The application developing takes as reference the standard of **Java Enterprise Edition 7** (**JEE7**), the last release available now.

We will use a *three-tier physical architecture* mapped on *four logical layers*, as the standard of JEE.
Here is the general schema of the **architecture**:

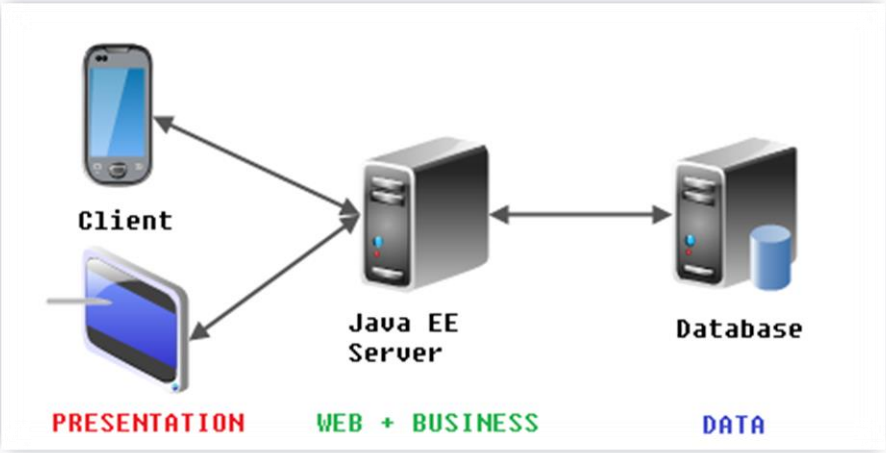


Now we will see in a deeper level of detail the meaning of each layer:

- **Client Layer**: it contains Application Clients and Web Browsers and interacts directly with the actors (Customers and Taxi Drivers). In our application, the Client can access via browser (*web application*) or via smartphone (*mobile application*).
- **Web Layer**: it contains the *Servlets* and *Dynamic Web Pages* that needs the elaboration. This tier receives the requests from the *Client layer* and forwards the pieces of data collected to the *Business Layer*.
- **Business Layer**: it contains the application logic (with the *Java Beans* and the *Java Persistence Entities)*. This will permit the communication between the System of mTS and the target users (Customers and Taxi Drivers).
- **EIS (Data Layer)**: it will contain all the *data* concerning Customers, Taxi Drivers, Calls, Reservations and other useful information. It is crucial to manage data according to strict policies about security and privacy.

It is important to underline that *myTaxiService* will also use external pre-built software products from the point of view of the business logic, such as **Google Maps** for the maps and the **online payment services API** (*e.g. PayPal*). We will cover this aspect in chapter *3.7 - Other Design Decisions*.

Here is a schema of the **three physical levels** (*tiers*) of the architecture:



Note: *It is important to underline that both the Web Layer and the Business Layer are mapped on the same physical machine in our architecture (the Java EE Server).*

3.2. Component View

According to JEE Standard, in order to develop a modular system that can evolve in future without big problems we decided to make use of a *Component-based approach*.
The formal definition of component is the following one:

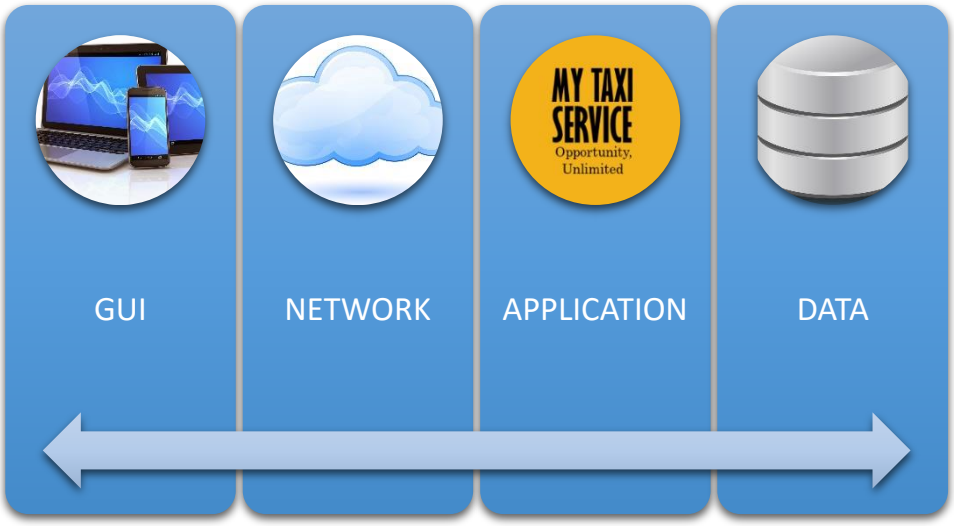
*A **Component** "represents a modular part of a system, that encapsulates its content and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces"*

We selected the main components of *myTaxiService* project with respect to the architecture of *Java Enterprise Edition*. From the point of view of the *logical architecture*, we will have:

- **Client Components**
- **Web Components**
- **Business Components**
- **Enterprise Information System (EIS) Components**

Note: as said before, the *Web Components* and *the Business Components* will be embedded in one physical machine (*JEE Server*).

It is important to underline that the main logic will run on the *Business Layer*.
We can see a little schema to clarify this notion:



Now let us make a distinction between the two macro-categories of Components:

- **Client-Side Components**: components located on the clients (*Taxi Driver Client, Customer Client, SysAdmin Client).* They have *sub-components* (*View* and *Controller*).
- **Server-Side Components**: components located on the server (*Taxi Driver Manager, Customer Manager, SysAdmin Manager, Queue Manager, Reservation Manager, Maps Manager*).
- **External Components**: they are not strictly part of *mTS*. They are external API integrated in *myTaxiService* app. These components are *Google Maps* and *Payment Service*.

As it is possible to imagine, there are also Component Interfaces. We will see the interfaces in detail in chapter *3.5 – Component Interfaces*.

Now let us define the **Components** of *myTaxiService*, with their functionalities:

❖ CLIENT-SIDE

- **Customer Client**: this component is the one that directly interacts with the end-users. It allows the users to do all the actions about reservations (create a new one, delete, see the current price…), account managements (create an account, change password….) and so on.
- **Taxi Driver Client**: this component represent what a Taxi Driver can see on his/her device. It is possible to see details about the reservations, the GPS position and so on.
- **SysAdmin Client**: this is a very crucial component. The System Administrator can have a direct vision of the System through this view. Therefore, he/she can decide to modify the position of the taxi drivers in the zones, to check heterogeneous kinds of information and do technical operations.

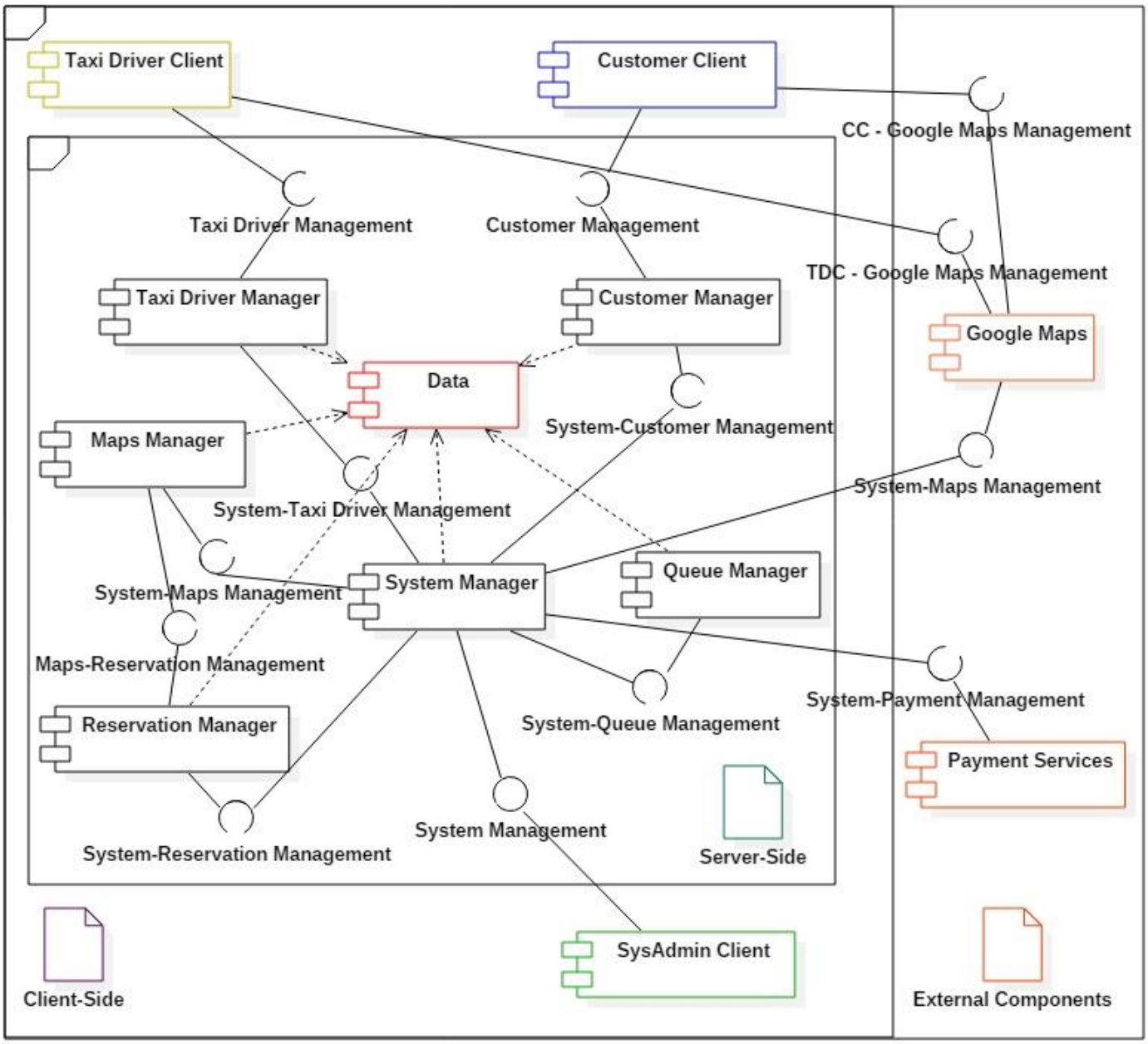
❖ SERVER-SIDE

- **Customer Manager**: this component manages the information about the Customers.
- **Taxi Driver Manager**: this component manages the information about the Taxi Drivers, with the current situation (and disposition in the areas).
- **System Manager**: this is a very important component. It interacts with the other pre-built software (like Google Maps and PayPal), performs the dispatching operations and permits to have a functional overall system. As we will see later, with the component interfaces, it is strictly linked to the other main components.
- **Reservation Manager**: this component manages all the operations about the reservation. For example, it traces the information about the reservations with the GPS positions, also when a reservation is deleted / updated. It works both for the "standard" reservation and for the LiveReservations™.
- **Queue Manager**: this component works on the queues in the different zones in parallel. It uses the *M/M/s model* presented in chapter *4 – Algorithm Design* to optimize the taxi distribution.
- **Maps Manager**: this component manages the Maps on the server side.

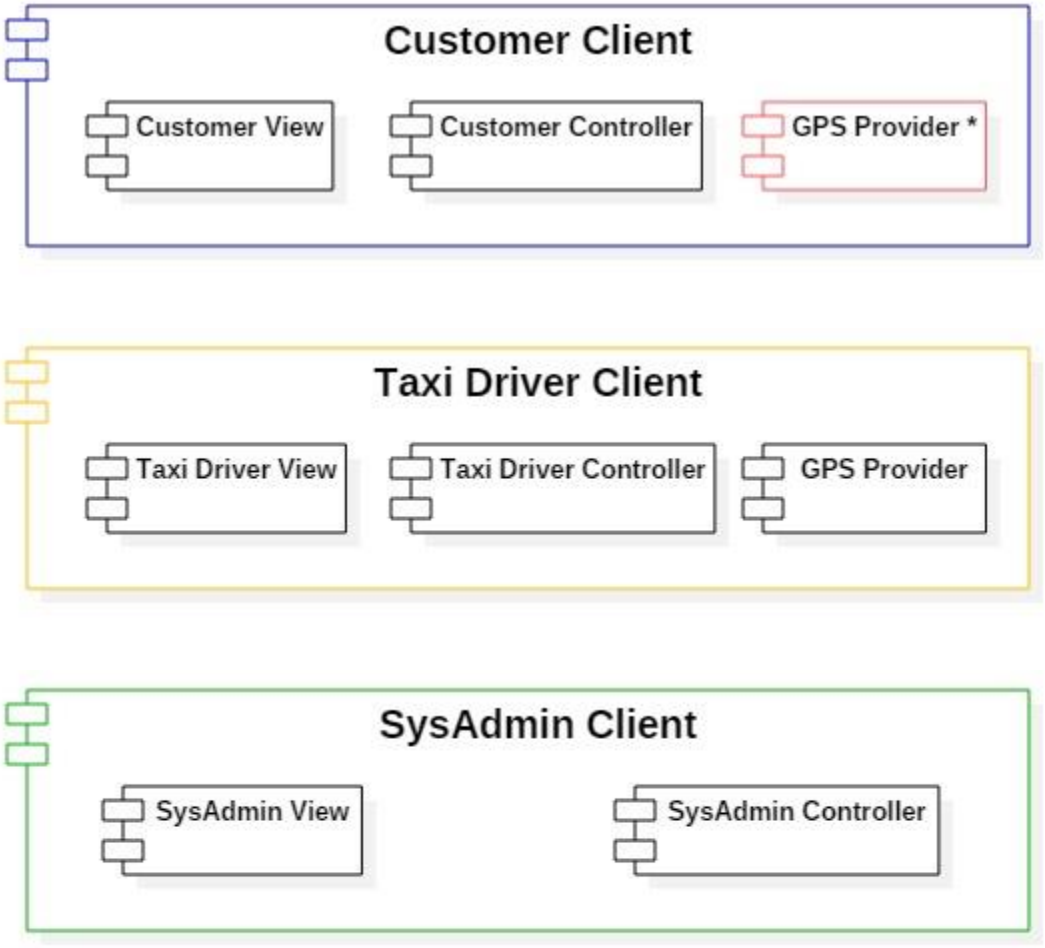
❖ EXTERNAL COMPONENTS

- **Google Maps**: this component integrates the API of *Google Maps* service.
- **Payment Service**: this component integrates the APIs about different payment services (as *PayPal*, for example).

This is the **UML Component Diagram** related to *mTS* project:



The inner structure of the **User-Side Components** is the following:



For each **Client** there are two components: the **View** and the **Controller**.

The **View** shows the GUI and allows the interaction with the Client. This interaction is obviously different between the Customer, the Taxi Driver and the SysAdmin.

From the other side, the **Controller** makes basilar checks on the input submitted by the View, before the direct interaction with the Server (and the main *Business Logic*).

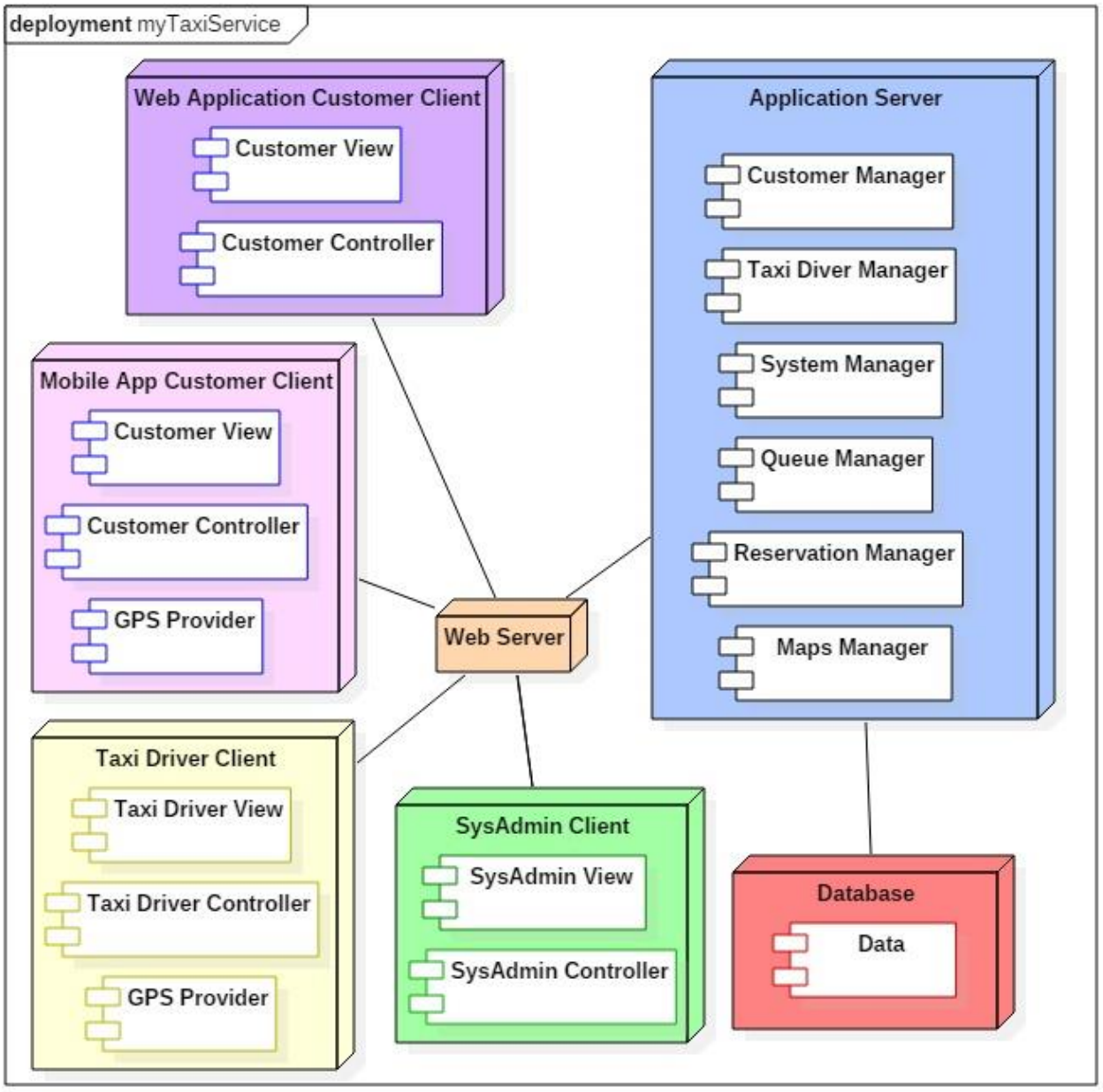
We have in addition the **GPS Provider** sub-components for the Customer and the Taxi Driver, which embedded the GPS.

Note: *The star (*) and the highlighted component mean that only the Mobile App Customer has the GPS Provider component.*

3.3. Deployment View

We briefly discussed about the physical mapping of the logical architecture in the chapter *3.1 – Overview*. Since this aspect is very important, we will go into details with a **Deployment Diagram**, to give the physical deployment of components on physical nodes. In order to grant a *high level of security*, we will use both hardware and software *cryptography*, as said in the **RASD**.

This is the **UML Deployment Diagram**:

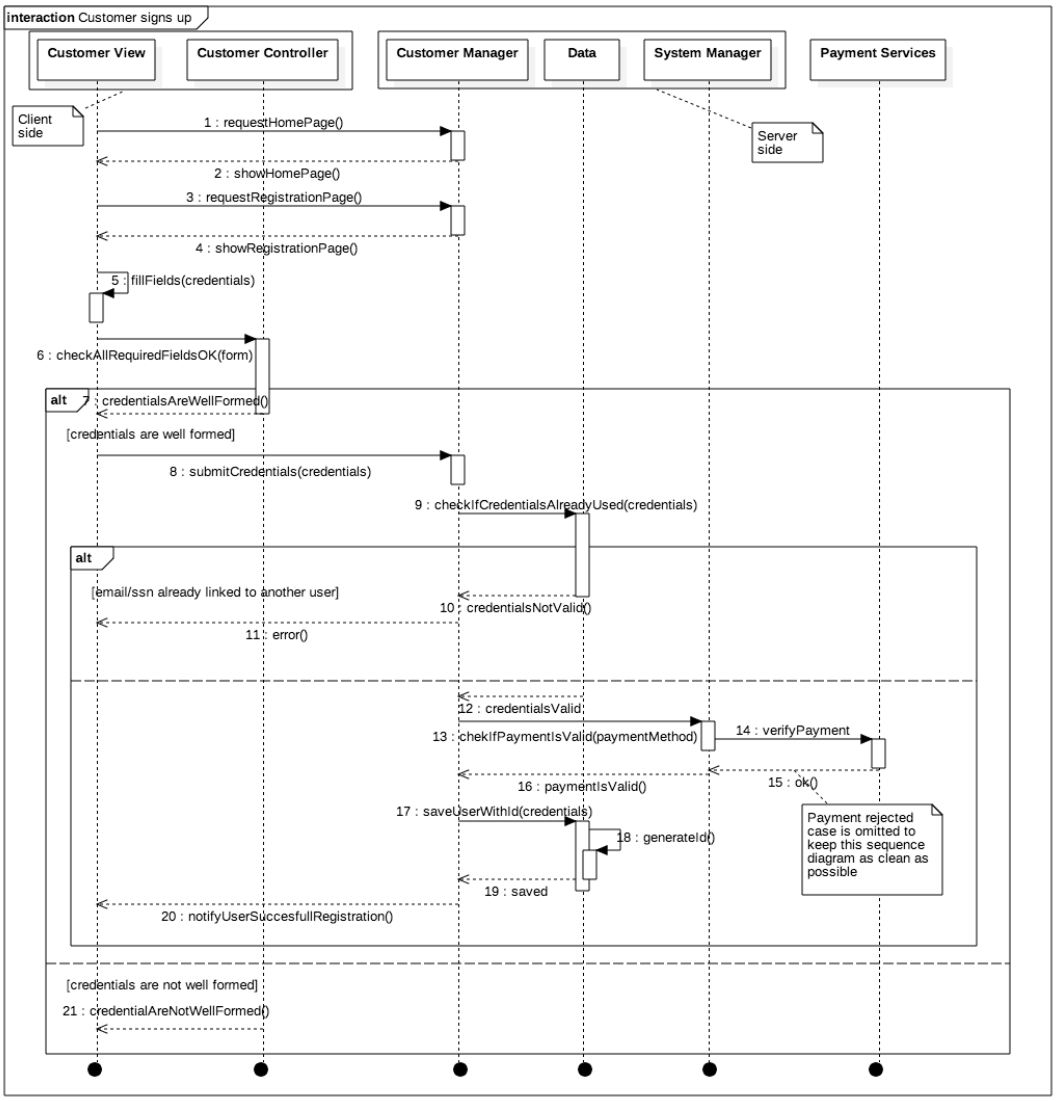


3.4. Runtime View

In this section we will provide **UML diagrams** to analyze some relevant use cases from a component **interaction** point of view. We will try to be as coherent as possible to notations and names we have already used in our RASD. Note that every diagram covers only cases we consider the most important ones.

3.4.1. Customer signs up (4.1.1)

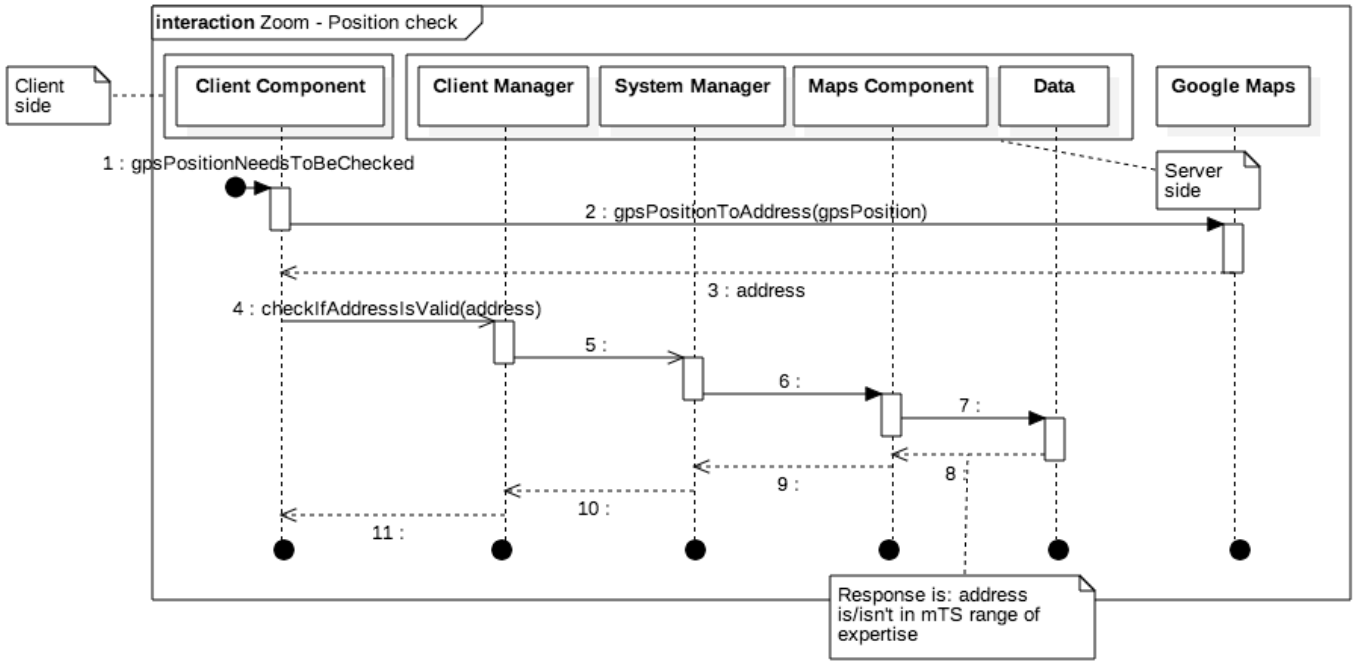
As you can see in this sequence diagram a first credentials check is done in *Customer View* component to avoid any possible submission of customer credentials that for sure are wrong (like malformed email address and empty required fields).
Data component will generate a new available (=not already used) *id* to identify customer with provided credentials.



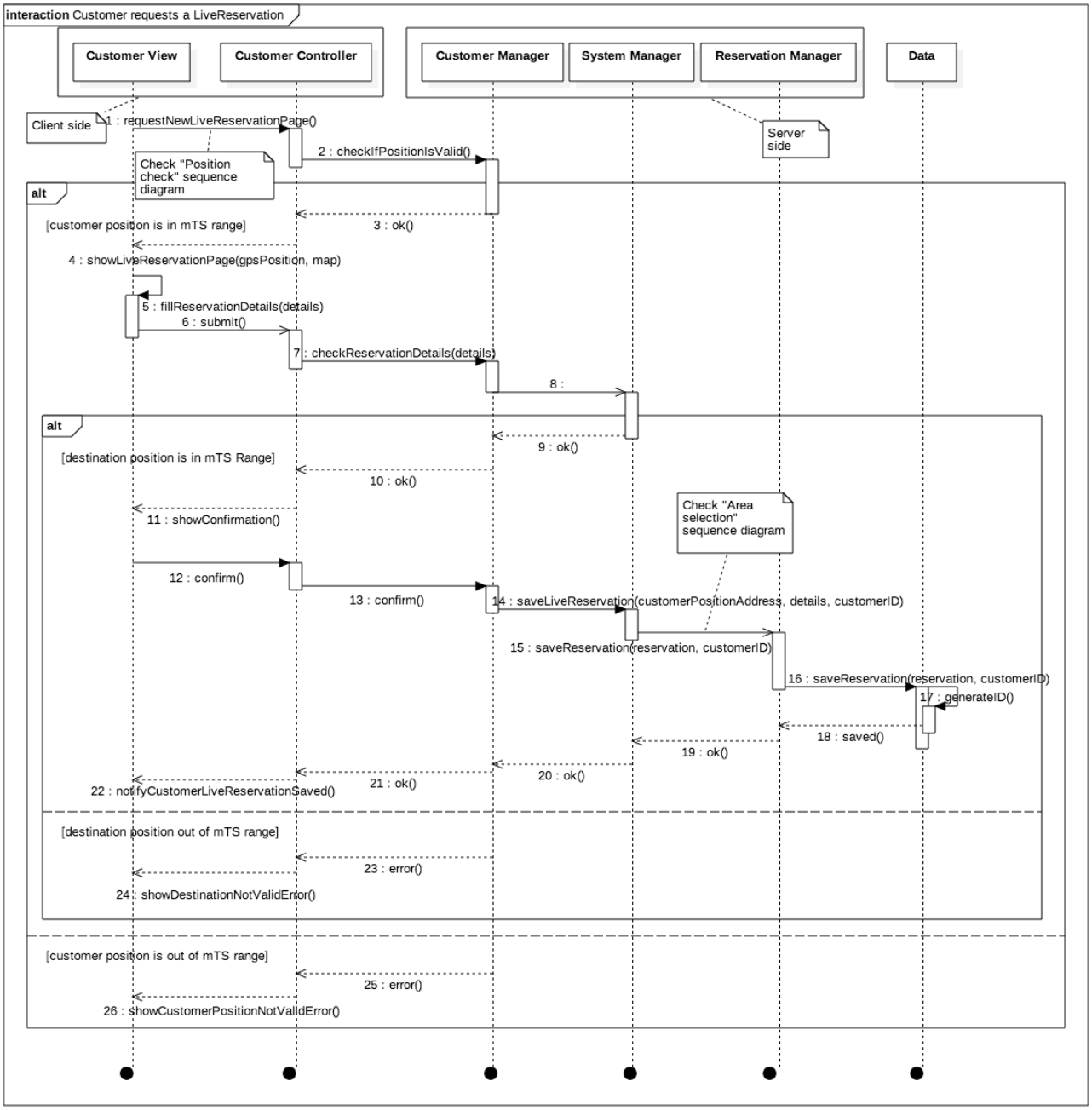
3.4.2. Position check

It is possible, for any authorized client, to verify if a GPS position is inside *mTS* area of expertise. Firstly, using Google Maps an address is found from a raw **GPS position**.

Then, with the following interaction, the system can check if this address is covered by *mTS* service.

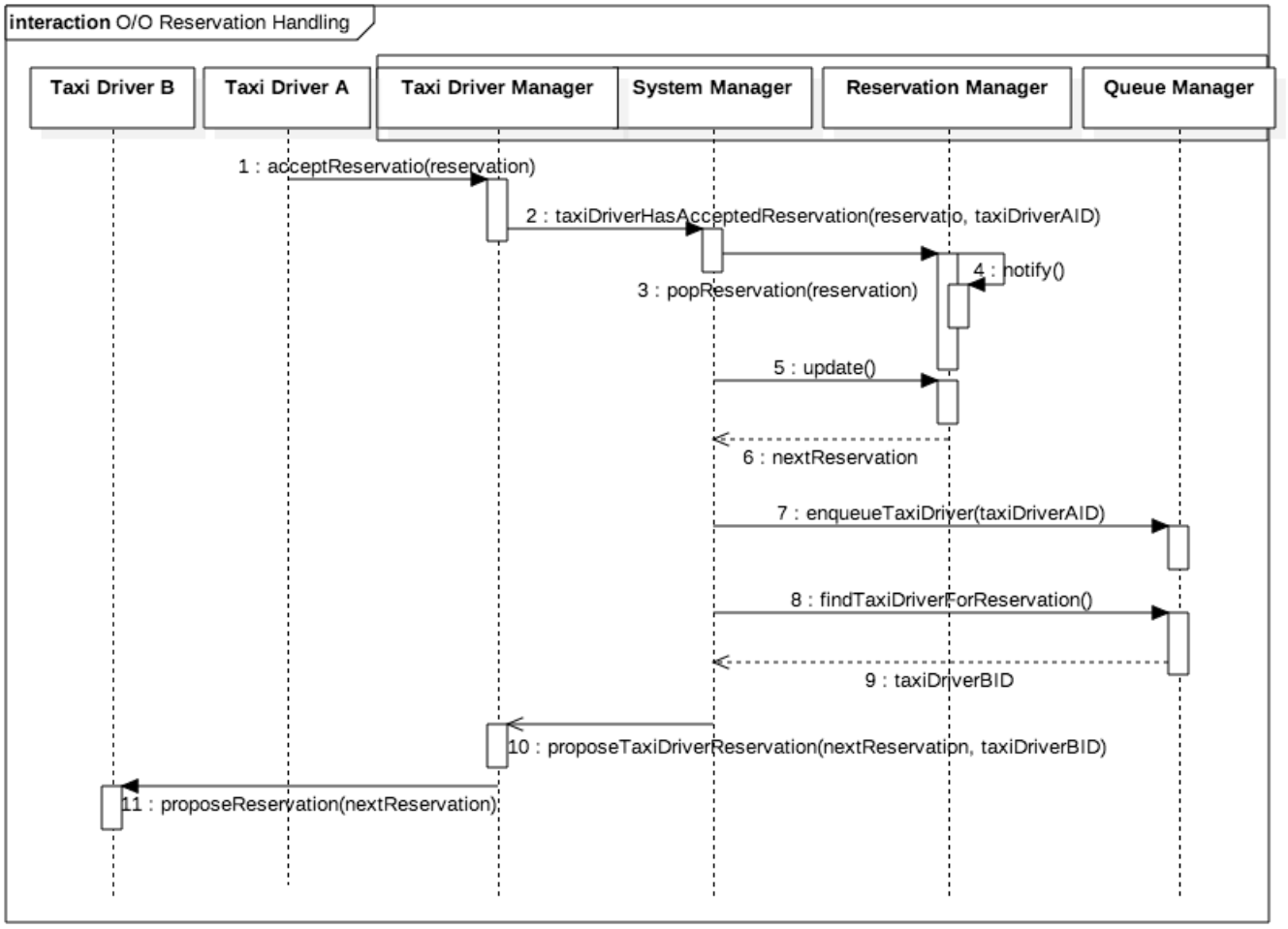


3.4.3. Customer requests LiveReservation™



3.4.4. Observer/Observable in Reservation Handling

Here you can find some idea about possible use of **Observer/Observable** in reservations handling. In this case *Reservation Manager Component* is observed by *System Manager.*



3.5. Component interfaces

One of the most interesting aspect of the components is their interaction. We will see in this paragraph that they interact each other with ad-hoc interfaces. A big amount of interaction will be through the web. This means that the interactions will adopt specific protocols (like *TCP/IP, HTTPs*).

Anyway, this is the list of the **component interfaces**, with the explanation of the interactions between the components:

❖ Interfaces between Client Side and Server Side Components

- **Customer Management**: connects the Customer Client with the Customer Manager Component. This interface contains methods about several actions as login, make a reservation, delete/update a reservation and so on.
- **Taxi Driver Management**: connects the Taxi Driver Client with the Taxi Driver Manager Component. This interface includes useful methods like *acceptReservation(), refuseReservation()* and so on.
- **System Management**: connects the SysAdmin Client with the System Manager Component. It contains many interesting methods; in fact, the System Manager is on the basis of each internal interaction and communication between modules of *myTaxiService*.

❖ Interfaces between Server Side Components

- **System-Reservation Management**: connects the System Manager Component with the Reservation Manager Component.
- **System-Queue Management**: connects the System Manager Component with the Queue Manager Component.
- **System-Customer Management**: connects the System Manager Component with the Customer Manager Component.
- **System-Taxi Diver Management**: connects the System Manager Component with the Taxi Driver Manager Component.
- **System-Maps Management**: connects the System Manager Component with the Maps Manager Component.
- **Maps-Reservation Management**: connects the Maps Manager Component with the Reservation Manager Component.

❖ Interfaces between Server Side Components and External Components

- **System-Maps Management**: connects the System Manager Component with the Google Maps Component. This allows the System to have information about the reservation places in detail, according to Google Maps.
- **System-Payment Management**: connects the System Manger Component with the Payment Services Component. This interface allows the integration of online payment services, very useful nowadays.

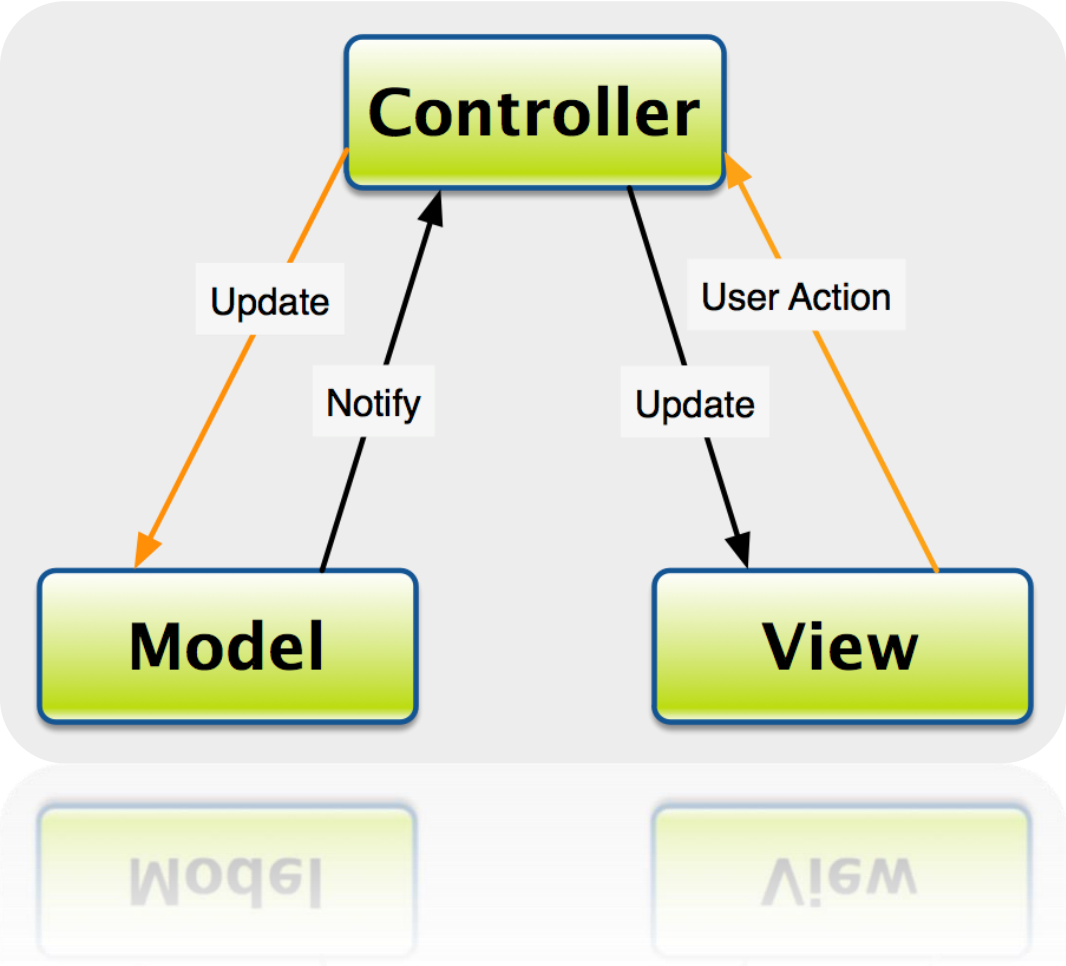
❖ Interfaces between Client Side Components and External Components

- **CC-Google Maps Management**: connects the Customer Client Component with the Google Maps Component. This interaction is deployed only in the case of the mobile app Customer, that have the GPS System integrated in his/her smartphone.
- **TDC-Google Maps Management**: connects the Taxi Driver Component with the Google Maps Component. This interaction is one of the most important. In fact, a very functional Taxi Driver's GPS with Google Maps assures a very good service.

3.6. Selected architectural styles and patterns

3.6.1. MVC (Model-View-Controller)

Model-View-Controller (also known as **MVC**) is one of the most diffused *structural pattern* in the applications with interfaces. It separates three logical parts: the **Model**, the **View** and the **Controller**. These parts interact each other according the following general schema:



Model-View-Controller general schema

In the specific case of *myTaxiService* these three logical parts will have the following *roles* and *interactions*:

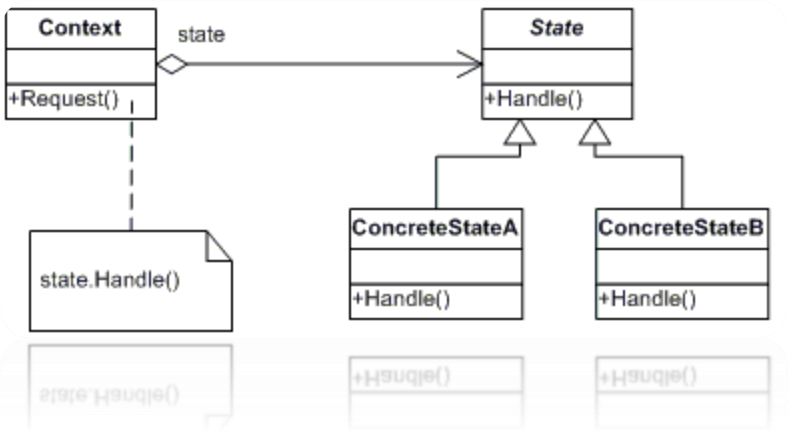
- The **Controller** sends commands to the Model to update the Model's state (e.g., editing tuple in mTS Database with a new reservation). It must also send commands to its associated

view (the Taxi Driver or the Web/Mobile Customer one) to change the View's presentation of the Model (e.g., by scrolling through a reservation form).

- The **Model** stores data that is retrieved according to commands from the Controller and displayed in the View. It contains all the data about reservations, Customers, Taxi Driver and son on.
- The **View** generates an output presentation to the user based on changes in the Model. There will be three different types of View: *Mobile Customer View, Web Application Customer View and Taxi Driver View*.

3.6.2. State pattern

The **State pattern** is a behavioral software design pattern, also known as the **objects for states pattern**. This pattern is used in computer programming to encapsulate varying behavior for the same object based on its internal state. Here is a general schema in *UML*:



State pattern general schema

In the case of *myTaxiService* this pattern can be very helpful with the management of the **Taxi Drivers**. In fact, a Taxi Driver can be in different states from the point of view of the System.

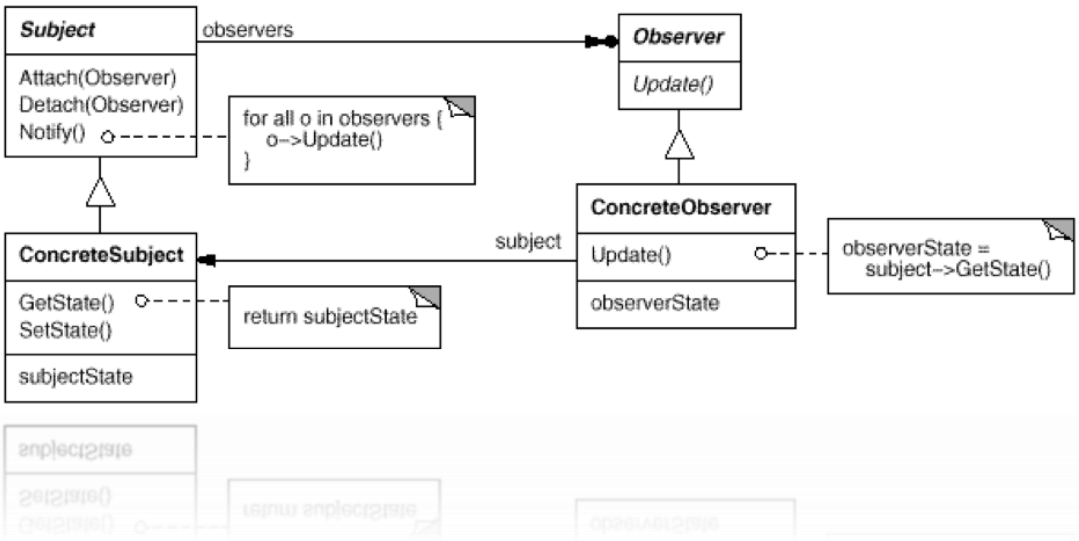
Let us focus on the possibilities. The possible **states** of a Taxi Diver are:

- *At work – Available at the moment*
- *At work – Not available at the moment in the current queue*
- *At work – Currently moving to another area*
- *Not at work*

This will take trace of the status of any Taxi Driver during the whole day and can be used to compute the queue on the base of the available Taxi Drivers.

3.6.3. Observer/Observable pattern

The main purpose of this behavioral software design pattern is to define a one-to-many dependency between objects in such a way that if an *Observable* object change its internal state, every *Observer*s of this particular object are automatically notified and updated. Here is a general schema in UML:



Observer/observable general schema

Note: Observer/observable pattern is a key part of MVC pattern.

3.7. Other design decisions

As we said in the component paragraph, *myTaxiService* will interact with two **external services**, like *Google Maps* and the *Payment* one (as *PayPal*).

In fact, both these services are standards all over the world and the creation of ad-hoc parallel services would not have sense.

This in perfectly coherent with *modularity* and *reusability* principles of Software Engineering.

“ Don’t reinvent the wheel,

just realign it. “

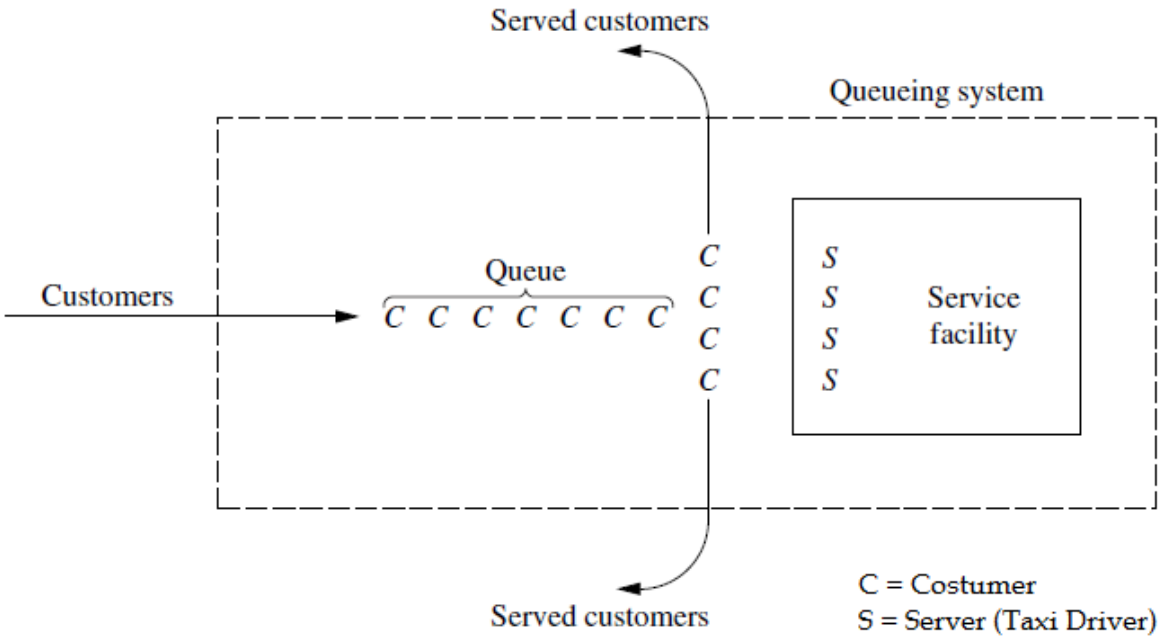
4. Algorithm design

4.1. Queue Management: M/M/s model

In order to explain the management of the queues it is important to recall the basics of the **M/M/s** model of **Queue Theory**. This recapitulation will also help future developers to understand the bases of the queue model of *mTS*.

We will not enter in details about the **implementation** of the model in our system with the source code of the **algorithm**, because it will only be a constraint for the successive phases of developing. In fact, the **Design Document** must have a *higher level of abstraction*.

Anyway, it is good to give an idea of the Queue Management, with the right terminology and the basilar math notation. Here is a high-level schema of *myTaxiService* Queue Model:



Model Assumption: *interarrival times* and *service times* are identically distributed according to an *exponential distribution.*

Now let us define the probabilistic variables and constants of this model in a formal way:

- **State of system** = *number of customers in queueing system.*
- **Queue length** = *number of customers waiting for service to begin.*
- **N(t)** = *number of customers in queueing system at time t ($t \geq 0$).*
- **$P_n(t)$** = *probability of exactly n customers in queueing system at time t, given number at time $t_0 = 0$.*
- **s** =*number of servers (parallel service channels) in queueing system.*
- **λ_n** = *mean arrival rate (expected number of arrivals per unit time) of new customers when n customers are in system.*
- **μ_n** = *mean service rate for overall system (expected number of customers completing service per unit time) when n customers are in system. Note: μ_n represents combined rate at which all busy servers (those serving customers) achieve service completions.*
- **P_n** = *probability of exactly n customers in queueing system.*
- **L** = *expected number of customers in queueing system.*
- **L_q** = *expected queue length (excludes customers being served).*
- **°W** = *waiting time in system (includes service time) for each individual customer.*
- **W** = *E(°W), where E() is the expected value.*
- **°W_q** = *waiting time in queue (excludes service time) for each individual customer.*
- **W_q** =*E(°W_q), where E() is the expected value.*

To simplify the notation, we define some support variables:

$$C_n = \frac{\lambda_{n-1}\lambda_{n-2}\cdots\lambda_0}{\mu_n\mu_{n-1}\cdots\mu_1}, \qquad \text{for } n = 1, 2, \ldots,$$

$$P_n = C_n P_0, \qquad \text{for } n = 0, 1, 2, \ldots.$$

$$P_0 = \left(\sum_{n=0}^{\infty} C_n\right)^{-1}.$$

$$L = \sum_{n=0}^{\infty} nP_n, \qquad L_q = \sum_{n=s}^{\infty} (n-s)P_n.$$

$$W = \frac{L}{\bar{\lambda}}, \qquad W_q = \frac{L_q}{\bar{\lambda}},$$

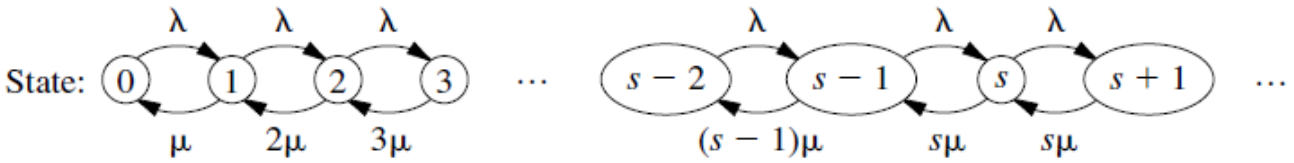
$$\bar{\lambda} = \sum_{n=0}^{\infty} \lambda_n P_n.$$

Now let us introduce the **Markov Rate Diagram** before the final formulation:

Multiple-server case ($s > 1$)

$$\lambda_n = \lambda, \quad \text{for } n = 0, 1, 2, \ldots$$

$$\mu_n = \begin{cases} n\mu, & \text{for } n = 1, 2, \ldots, s \\ s\mu, & \text{for } n = s, s+1, \ldots \end{cases}$$



This is very useful to give an idea of the probabilistic states of the **Taxi Queues**, according to *Markov chains* notation.

Now let us give a **closed formulation** of the useful terms of the model:

$$C_n = \begin{cases} \dfrac{(\lambda/\mu)^n}{n!} & \text{for } n = 1, 2, \ldots, s \\ \dfrac{(\lambda/\mu)^s}{s!}\left(\dfrac{\lambda}{s\mu}\right)^{n-s} = \dfrac{(\lambda/\mu)^n}{s!\,s^{n-s}} & \text{for } n = s, s+1, \ldots. \end{cases}$$

$$P_n = \begin{cases} \dfrac{(\lambda/\mu)^n}{n!} P_0 & \text{if } 0 \le n \le s \\ \dfrac{(\lambda/\mu)^n}{s!\,s^{n-s}} P_0 & \text{if } n \ge s. \end{cases}$$

$$L_q = \sum_{n=s}^{\infty} (n-s)P_n$$

$$W_q = \frac{L_q}{\lambda};$$

$$W = W_q + \frac{1}{\mu};$$

$$L = \lambda\left(W_q + \frac{1}{\mu}\right) = L_q + \frac{\lambda}{\mu}.$$

The probability distribution of waiting time is represented by the following formulae:

❖ Service Time Included

$$P\{W > t\} = e^{-\mu t}\left[\frac{1 + P_0(\lambda/\mu)^s}{s!(1-\rho)}\left(\frac{1 - e^{-\mu t(s-1-\lambda/\mu)}}{s-1-\lambda/\mu}\right)\right]$$

❖ Service Time Excluded

$$P\{W_q > t\} = (1 - P\{W_q = 0\})e^{-s\mu(1-\rho)t}$$

Where:

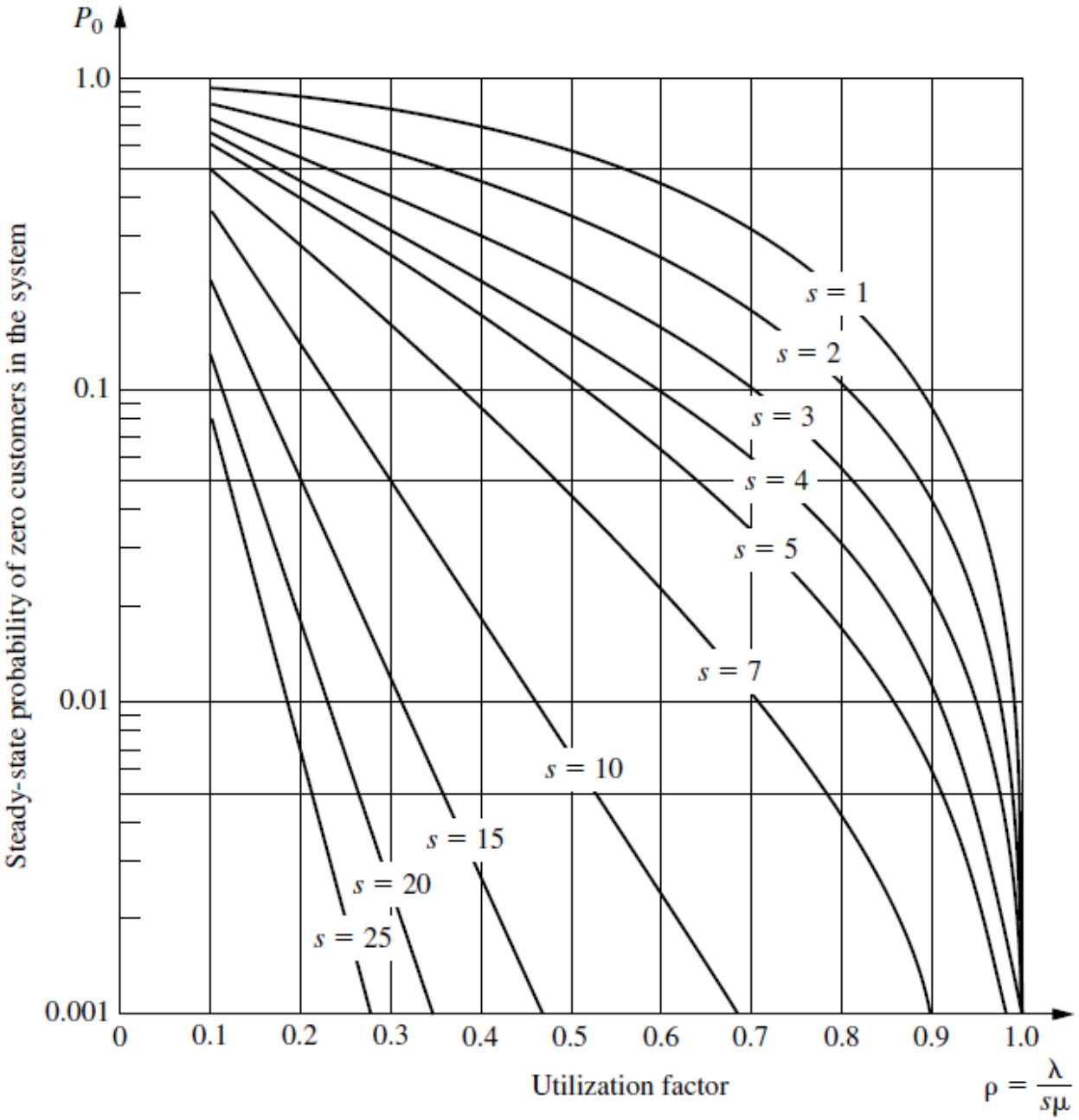
$$P\{W_q = 0\} = \sum_{n=0}^{s-1} P_n.$$

In our model, we assume that all the **areas** have the *same exponential distribution of reservations*. This may be a good assumption in practice if the area are "*balanced*".
However, we plotted on two different graphics some interesting results of the **M/M/s model** of *Taxi distribution.*

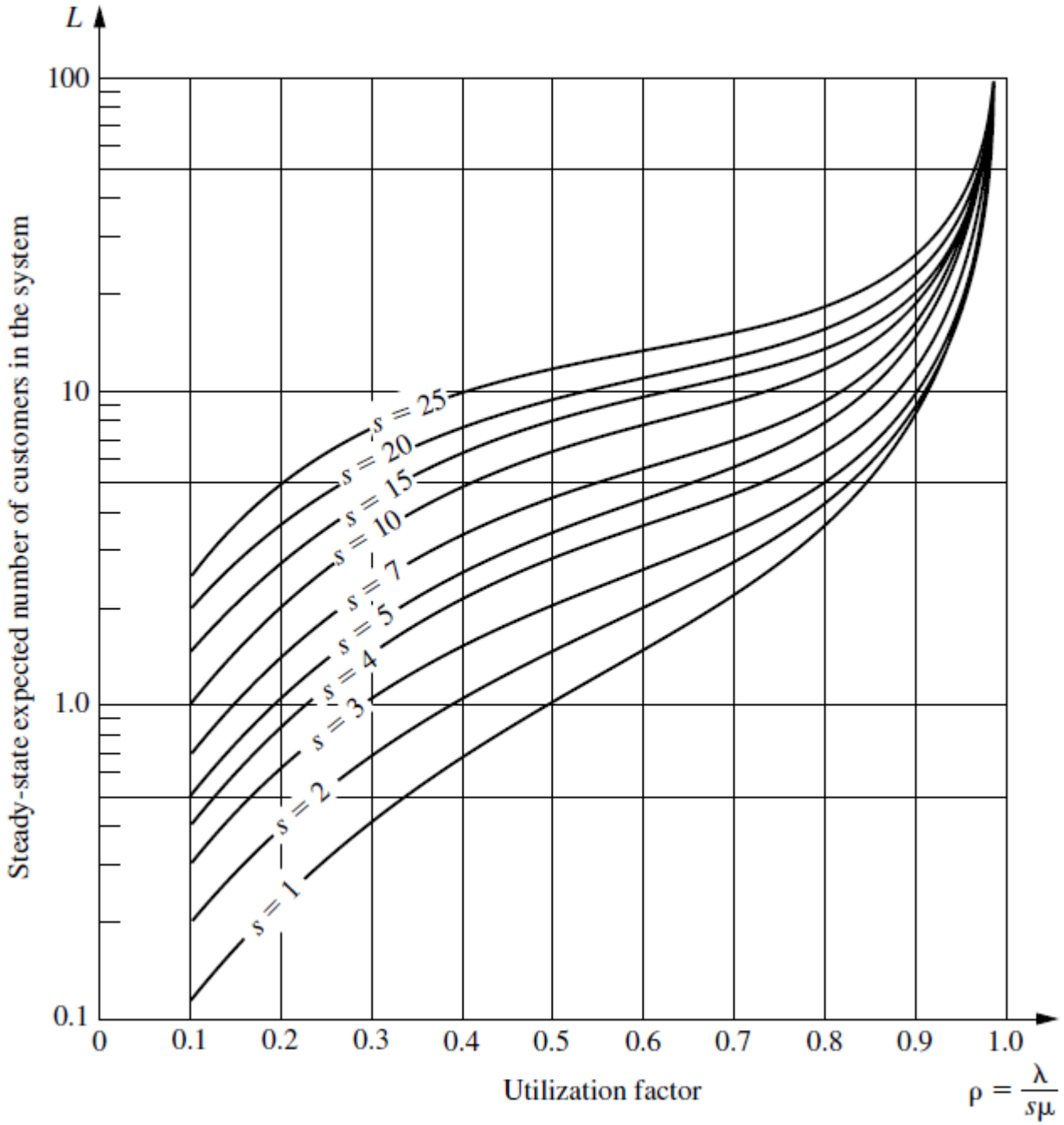
The **first one** is the *steady-state probability of zero Customer in the system (P_0)* as a function of the *servers* (*s, Taxi Drivers*) and the *Utilization factor (ρ)*.
The **second one** is *steady-state expected number of customers in the system (L)* as a function of the *servers* (*s, Taxi Driver*) and the *Utilization factor (ρ).*

Anyway, sometimes graphics are better than words. Therefore, here is the two probabilistic graphics, as promised before:

1. *Steady-state probability of zero Customer in the system (P_0) as a function of the servers (s, Taxi Drivers) and the Utilization factor (ρ).*



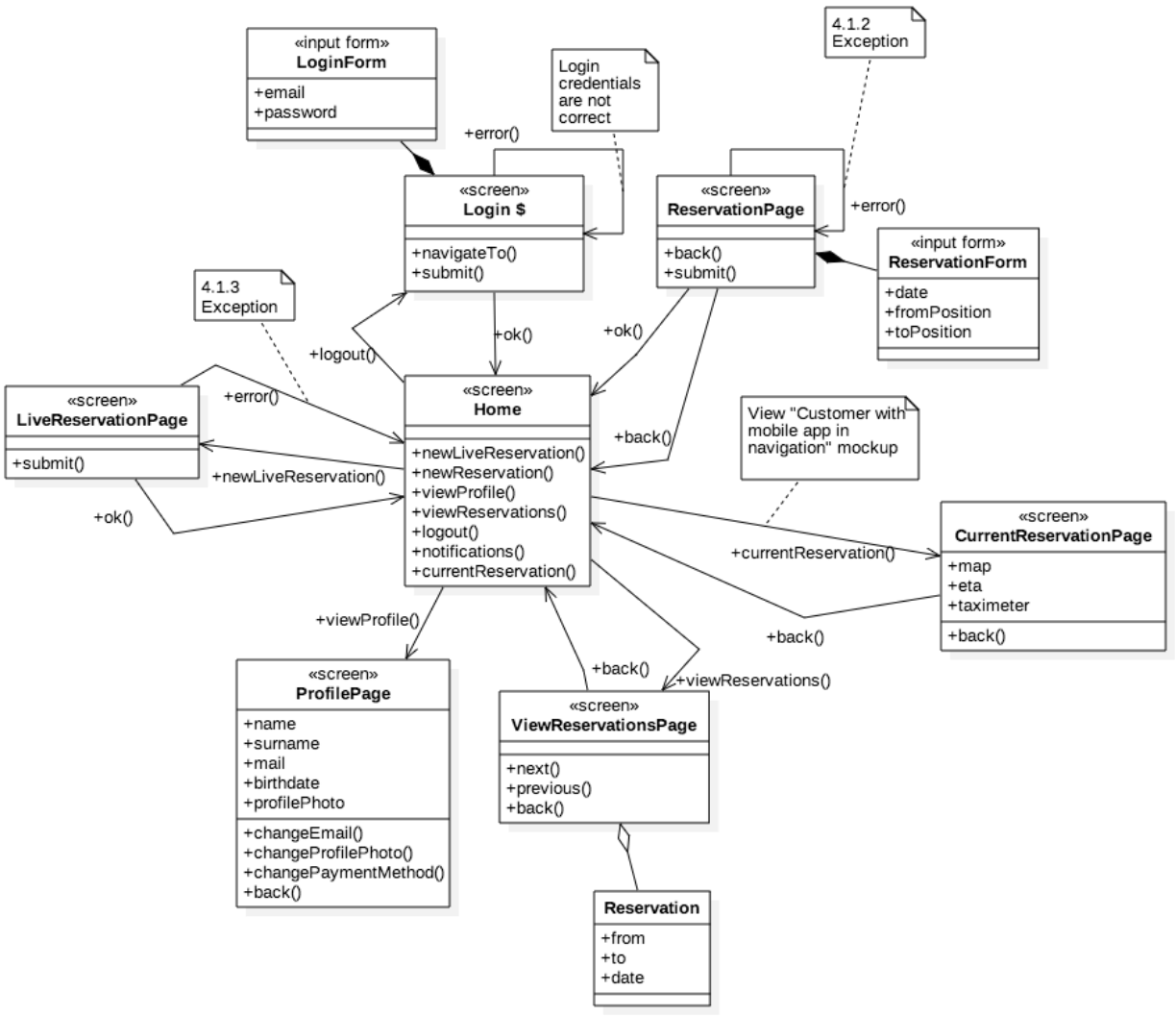
2. *Steady-state expected number of customers in the system (L)* as a function of the *servers* (*s, Taxi Driver*) and the *Utilization factor (ρ).*



5. User Interface Design

User Interface plays a very important role in interactions between machines and humans. Given that, we have assumed that "*no special skills are requested*" to use *mTS* product, **UI** must be kept as simple as possible without sacrificing functionality.
In **RASD** we have provided some mockups to give a general idea of mTS application using visual elements like *buttons*, *form*, *pages*, etc. Now we would like to show some UX Diagrams to have a more abstract model. We encourage you to keep on hand **mockups** during the reading of this paragraph.

5.1. Passenger Application

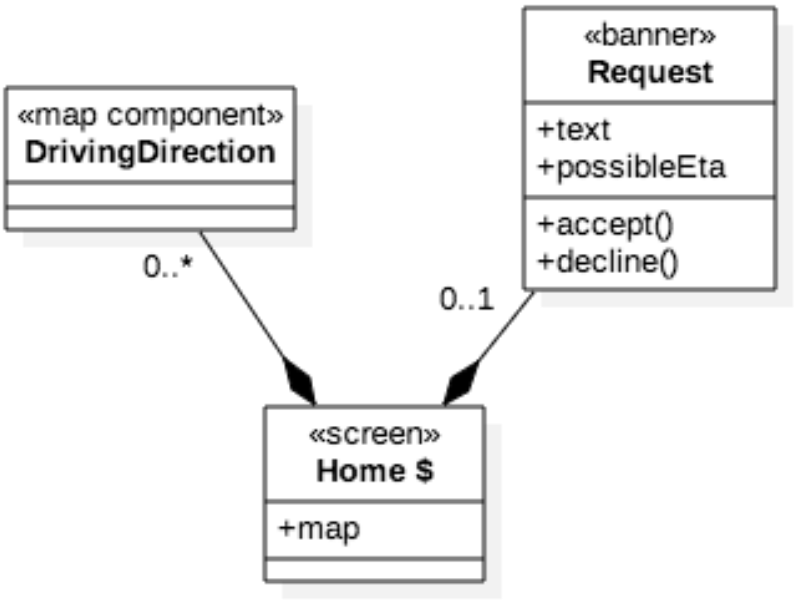


Note: As you can see, there are some references of *exceptions* that can occur in some use cases.

5.2. Taxi Driver Application

As we have said in **RASD** interaction between taxi driver and driver application is very limited. In fact, driver can only see driving directions provided by the system and confirm/decline requests to take care of a reservation.

Please, excuse our little UML "abuse" of this particular diagram. With *"map component"* we actually mean something that is showed over the map piece of the main screen and with *"banner"* we mean something that partially overlays the main screen.



6. Requirements Traceability

At a first sight this table can be found really similar to the sequence diagrams we have previously proposed in *Component View* section. Please, keep in mind that differences between *Functional Requirements* and *Use Cases* are not always well defined. In this table we do **not** cover relations between *Components* and *Use Cases*.

For example for "*Accept/decline system request to take care of a customer"* requirements we have purposely omitted, for example, the interaction between *Data* and *Reservation Manager/Taxi Driver Manager* to retrieve respectively a new reservation/proposed taxi driver. We think that in this example case we only need to understand what components are involved to give the ability to a driver to accept/decline a reservation request.

Keep in mind that:

- With the sentence *"a component contacts another component",* we mean *"a component shares information/makes requests to another component using the interface between them"*.
- With *User* we mean any possible user our system can/will support.
- As we have said in RASD, mTS System supports three different types of registered user: *SysAdmin*, *Customer* and *TaxiDriver* that have different possibilities.
- With *Client* we mean the generic macro component containing *Client View* and *Client Controller* that give ability to a generic authorized type of user to interact with *mTS* System.
- Remember that user must provide a valid payment method during the registration so usually there is no need to interact between *Customer Client* and *mTS* server side during a financial transaction (it heavily depends on payment method chosen during the registration).

Functional Requirement	*Components*	*Description*
Registration of a user to the system	Client, Client Manager, System Manager, Data, (Payment Services*)	How *mTS* components interact to successfully register a new user? An authorized *Client* sends details of a user that wants to be registered to *Client Manager*. *Client Manager* then forwards personal details to *System Manager*. *Data* is then contacted by *System Manager* to respectively check if provided credentials are not already used by another user. If no problem arises in this phases, then *System Manager* contacts *Data* to finally save a new user.

		In this case, it could be useful to have a look at *Customer Signs Up* sequence diagram in *Runtime View* section. In this *Use Case Payment Services* need to be contacted to complete the registration.
Login of a user to the system	Client Manager, Client, System Manager, Data	How *mTS* components interact to successfully identify a user? *Client* contacts *Client Manager* and sends provided credentials (form filled in *Client View*). *Client Manager* forwards credentials to *System Manager*. *System Manager* then contacts *Data* to check if provided credentials are correct. A response is then sent to *Client Manager* (that will eventually provide to *Client View* a proper landing page).
Make financial transaction	Customer Manager, System Manager, Payment Services*, Data	How *mTS* components interact to successfully make a financial transaction from a customer and to *mTS*? For simplicity we will consider the case in which chosen payment method will not require user interaction to complete the payment to *mTS*. If needed *Customer Manager* will automatically contact *System Manager* requesting a payment from a specified customer. Then *System Manager* contacts *Data* to find information about payment method and then contacts the proper *Payment Service* to complete the transaction.
Route taxi driver to certain location	Taxi Driver GPS Provider, Taxi Driver View, Taxi Driver Controller, Taxi Driver Manager, Google Maps*, System Manager, Maps Manager	Given a certain location, how can *mTS* give the right driving direction to a taxi driver? *Taxi Driver View* has a map to show taxi driver current position and driving directions. *Taxi Driver GPS Providers* provides taxi driver current GPS position to *Taxi Driver Controller*. This position is sent to *Maps Manager* through *Taxi Driver Manager* and *System Manager*. *Maps Manager* then send next destination to *Taxi Driver Controller* (through *System Manager* and *Taxi Driver Controller*). *Taxi Driver Controller* than contacts *Google Maps* to retrieve driving directions from taxi driver position and next destination.
Create/delete/update taxi reservation	Customer Client, Customer	How can we interact with a customer to send some given updates about his reservation? *mTS* supports two types of updates: notification and

	Manager, System Manager, Reservation Manager	travel information (current position, taximeter and eta). For simplicity, we only consider travel information. *Customer Controller* periodically contacts *Customer Manager* to retrieve updates about current state of the reservation. Then *System Manager* is contacted by *Customer Manager* to forward the request to *Reservation Manager*. *Reservation Manager* will then retrieve specific information and send them back to *System Manager*. *System Manager* will then send travel information to *Customer Controller* through *Customer Manager*. Finally, *Customer Controller* will updates *Customer View* to show updates to the customer. **Note**: In this case, we do not cover how *Customer Manager* retrieves the required information because this is not what the requirement is about.
Send information about travels/notifications to a customer	Customer Client, Customer Manager, System Manager, Reservation Manager, Taxi Driver	How *mTS* components interact during a reservation handling done by a customer? For simplicity, we only consider how a reservation is created and then saved. Reservation details filled in *Customer View* are sent to *Customer Manager* through *Customer Controller. Customer Manager* will then contact *System Manager* to forward reservation details to *Reservation Manager*. Reservation Details will finally check if provided details are acceptable and then send them to *Data*. See the "*Customer makes a LiveReservation*™" sequence diagram to understand how this requirement can be exploited to satisfy a use case.
Accept/decline system request to take care of a customer	Taxi Driver View, Taxi Driver Controller, Taxi Driver Manager, Reservation	How can *mTS* interact with a taxi driver to propose him a new reservation? *Reservation Manager* provides to *System Manager* a new reservation that needs to be taken care of. This reservation is then forwarded to a specific taxi driver through *Taxi Driver Manager*. *Taxi*

	Manager, System Manager	*Driver Controller* receives the request and show it to the taxi driver using *Taxi Driver View*. Taxi driver interacts with *Taxi Driver View* (read *User Interface Design* paragraph for more details) accepting or declining. *Taxi Driver Controller* then forward the response to *Taxi Driver Manager.*
Request/acquire user position	Client GPS Provider, Client Controller, (Google Maps*), Client Manager	How can *mTS* interact with a user to acquire his position? In truth no real request is forwarded to our user (customer or taxi driver). Position is given automatically when needed by *mTS* Taxi Driver application and *mTS* Customer application. *Client Controller* contacts *Client GPS Provider*. Then, if required, *Client Controller* sends a raw GPS position to *Google Maps* to have an address. This address is then forwarded to *Client Manager* by *Client Controller*. This is actually very similar to "Zoom – Position check" sequence diagram. In that case the address position is checked to understand whether it belongs to mTS area of expertise or not.

Note: *The star (*) is for External components. See Architectural Design paragraph for more.*

7. References

7.1. References list

Here is a short list of the **references** for this **Design Document**:

- **Slides of the Software Engineering 2 course** *(from the Beep Platform)*
- **Design Document Template** *(by Prof. Raffaela Mirandola)*
- **Software Engineering: Principles and Practice** *(Hans Van Vliet)*
- **UML Distilled** *(Martin Fowler)*
- **Introduction to Operations Research** *(Frederick S. Hillier, Gerald J. Lieberman)*
- **Wikipedia** (https://www.wikipedia.org/)

7.2. Software and Tools Used

- **Microsoft Word** (https://products.office.com/it-it/word): redaction, formatting and revision of the document.
- **StarUML** (http://staruml.io/): to create and develop incrementally the UML models of the project (*Component Diagram, Deployment Diagram, Sequence Diagrams, UX Diagrams*).
- **Gimp** (http://www.gimp.org/): graphic elaborations of the project.
- **GitHub** (https://github.com/): to develop the project using a distributed repository.
- **Dropbox** (https://www.dropbox.com/): to share other contents step to step.

7.3. Hours of work

- **Andrea Martino**: ~ 30 Hours
- **Francesco Marchesani**: ~ 30 Hours