# myTaxiService
# Software Design Document
# Version 1.0

Losio Davide Francesco, Luchetti Mauro, Mosca Paolo

November 26, 2015

# Contents

# 1 — Introduction

This documentation will be used to aid in software development by providing further details of how the software should be built. Within the Software Design Document are narrative and graphical documentation of the software design of the project including use case models, sequence diagrams, and other supporting requirement information.

## 1.1 Purpose

The purpose of the Software Design Document is to provide a description of MyTaxiService system design and architecture fully enough to allow software development to proceed with an understanding of what is to be built and how it is expected to be built. To achieve this DD(**D**esign **D**ocument) translates and states more accurately the Requirement Specifications described in the My-TaxyService RASD document. It identifies high-level system architecture and design framework as well as hardware, software, communication and interface components.

## 1.2 Scope

MyTaxiService application is a server/client combination that will allow a user to handle different type of taxi service, keeping track of all the transaction necessary for the completion of each operations. This will include booking a taxi, request a taxi as soon as possible, the handling of the sharing option and, for the taxi drivers, the managing of the taxi queue. All this functionalities will be guaranteed in the way and in the manner explained in the RASD document. Via a Cross Platform Web Environment (by the use of angularJS, Ionic, Cordova and nodeJS frameworks), the MyTaxiService will be able to run on various platforms, including Unix, Linux and Windows based systems, and all the portable devices based on Android and Ios. When a network connection to the server is available, the user will be able to synchronize his PD (**P**ortable **D**evice) or PC with the server, he will be able to log in or register and makes his own request in the case of the passenger-user. Or to set is availability, accepts or rejects request in the case of the taxi-driver user.

Below are stated some main issues with which the system has to be capable to cope with.

- **PD Issues:** Because of memory limitations, a PD will only store data and application parts that are strictly necessary for a PD user. Also, PDs have reduced screen size and limited input capability compared to PCs, so we will design PD standalone functionality in manner that can be easily presented on a typical 240x320 screen.

- **Synchronization:** We will implement server software to serve as an interface between the PC or PD and the Application logic, by the re-using of already existent services offered by third parties company.

- **Transaction and functionalities:** all the transactions and booking procedure will be handled by the application logic layer that will be divided from the presentation layer, as well as the queue managing features and algorithm. PDs will implement only presentation layer and connection functionalities.

The architecture will be developed and structured to support the fulfillment of this main issues.

## 1.3    Definitions, Acronyms, Abbreviations

- **RASD:** **R**equirements **S**pecification **A**nalysis **D**ocument

- **DD:** **D**esign **D**ocument

- **PD:** **P**ortable **D**evice

- **MVC:** **M**odel **C**ontrol **V**iew

- **FIFO:** **F**irst **I**n **F**irst **O**ut, it's a policy applied to the queue managing. It means that the first person to be enqueued will be the first to be dequeued

- **PAAS:** **P**latform **A**s **A** **S**ervice

- **SOA:** **S**ervice **O**riented **A**rchitecture

## 1.4    References

- **MyTaxyService RASD** - November/6/2015, edited by Dadoz+GrinGo+Pol Corporation;

## 1.5    Document Structure and Overview

- **Architectural Design:** this section is the main focus of this document. It provides an overview of the system's major components and architecture, as well as architectural styles, pattern used and other design decision.

  a detailed analysis of modules will also describe lower-level classes, components, and functions, as well as the interaction between these internal components.

- **Algorithm design:** this section is focused on the definition of the most relevant algorithm of the project.

- **User interface design:** this section provide an overview on how the user interface(s) of the system will look like. In particular, referring on what has already stated in the RASD, here some further details is specified.

- **Requirements Traceability:** this section explain how the requirements defined in the RASD map into the design elements that are defined in this document.

# 2 — Architectural Design

## 2.1 Overview

This sections analyze several different part of the design architecture:

- System Components And Their Interactions: here are individuated and stated the main system software components in which the software is split into, and in the interactions among them. To achieve this latter feature, will be provided a high level description of the interfaces to be set-up. Within this description are:

  Component;

  Deployment;

  Runtime;

  UML diagrams. These diagrams are intended as a supply to the better understanding, as well as a more clear and simply specification of our components division.

- Architectural Styles And Patterns here are listed the architectural styles and patterns to use to solve the main interactions and functionality problems. They are taken in this first version of the DD more as suggestions and ideas. They will be updated and revised during the developing of the application where requested by the circumstances. In the sense that if the patterns and architectural styles will actually reveal to be unfeasible, different approach will be evaluated and chosen.

- Other Design Decision: Some other decision that doesn't belong to other section are listed here.

## 2.2 System Components

The system to be produced implement the common MVC pattern. The components are split in order to represent this logic. For each component the main modules are listed.

### 2.2.1 Server Model

This component represent the core logic of the whole system. Modules contained here are responsible for the correct performing of MyTaxiService features.

### 2.2.1.1 Queue Handler

This module has to guarantee the right usage of a FIFO (**F**irst **I**n **F**irst **O**ut) queues into the application. Keeping in mind that each zones has is own queue, the operations that this module handles are: add a new driver to the queue, delete a driver from the queue, select the first driver in the queue and move a driver from top to the bottom of the queue. It has to follow some rules deduced from the RASD document:

- To add a new driver it has to be available.

- A new driver has to be added into the right queue, that is related to the geographical location of the taxi

- The deleting of a driver implies that he has accepted/reject a request or that he is become unavailable.

### 2.2.1.2 Requests Handler

This module is responsible of the correct forwarding of each request coming from a user. It is in straight contact with the Queue Handler module because of his necessity to send requests to drivers. Furthermore, it embed some specific functionalities with respect of the request type:

- It has to forward the request with the sharing option enabled to the shareable ride finder.

- It has to store the booking in a correct structure that provide to notify the system 10 minutes before the reservation time to perform the effective taxi allocation.

### 2.2.1.3 Shareable Ride Finder

This is a sub-component of the request handler, its aim is to list-out the rides which have the same origin and destination position of the analyzed ride. The research is done upon a suitable memorization structure, in which are stated all the pending request, either they are booking request or immediate request. The immediate request will be stored in that structure for at most three minutes, this in order to keep consistent their "immediate" peculiarity. At the end of this time-out, an immediate request with the shareable option enabled will be treated as a non shareable ride and the passenger will have to pay the entire fee. During the performing of the search algorithm, the user will be able to stop this research in any moment, thanks to this, the user can decide to not wait for an outcome.

### 2.2.1.4 Mapping Features Handler

This module has to deal mainly with the google maps API. On the passenger side, it has to support the origin and destination addresses input. It has to guarantee the correctness of these addresses and to show a graphical map representation in order to enhance the interactivity. Whether on the taxi driver side, it has to deal with the queue manager by providing the driver positions, this a the queue manager to assign the taxi to the right queue. Moreover

it has to perform some navigation features. It has to provide the taxi driver with the available paths, distance and travel time to arrive at the passenger location. This can support the accept/reject decision of a request by a taxi driver. In case of acceptance, the mapping handler, has to connect the driver into his GPS navigation system by the opening of the related view, here the path passing through the passenger location and going to his final destination will be displayed.

#### 2.2.1.5 DBMS and Log-in

This module has to deal with the user's personal data. Since the only data needed by the application are the Log-In data related with the personal data, this system has also to handle the log-in procedure. It has to record in the DBMS the registration information and to query the log-in information provided by the user, finally it has to checks if there is a correct match.

### 2.2.2 Server Controller

This component represent the part of the system that is directly in contact with the users. It act as a glue between the view parts, loaded on the user's devices, and the model parts, that is loaded on the server. This part too has also to be loaded and performed by a dedicated server area.

#### 2.2.2.1 Passenger communicator

This module is responsible of the correct forwarding information between the server and a user. The operations that this module handles are: receive a message from a user, send a message to a user, handle timeout errors.

#### 2.2.2.2 Driver communicator

This module is responsible of the correct forwarding information between the server and a driver. The operations that this module handles are: receive a message from a driver (including the availability state and the GPS positioning), send a message to a driver, handle timeout errors.

### 2.2.3 Views

This component is a collection of graphical views related to the suitable devices and types of user. For graphical views we mean the graphical representation of both the user interface and the data computed by the model. This component has to manage the different input possibilities and to represent the results given by this inputs. In any case, the adoption of cross platform technologies allow us to develop one unique system interface that will be automatically adapted to a suitable for both the WEB application and the mobile application. Several useful mockups that picture a complete framework of all the functionalities requested by the user interfaces are stated in the RASD document. Because of their completeness and clearness they are reasonably assumed as an adequate support to the views development.

mainly there are two different views to be developed, according to the user types:

#### 2.2.3.1 Passenger-views

This views has to provide an easy access to all and only the passenger functionalities, it has to be mainly structured for the mobile applications, because it will assumed that the most of the users will access to myTaxiService with this interface. an the functionalities are already sufficiently stated in the RASD.

#### 2.2.3.2 Driver-views

This views has to provide an easy access to all and only the taxi driver functionalities. The same consideration made upon the passenger-view are also valid here.

## 2.3 Components scheme

To have a better overview of the various components with the relevant modules and their connection, it was decided to provide a general outline of the system
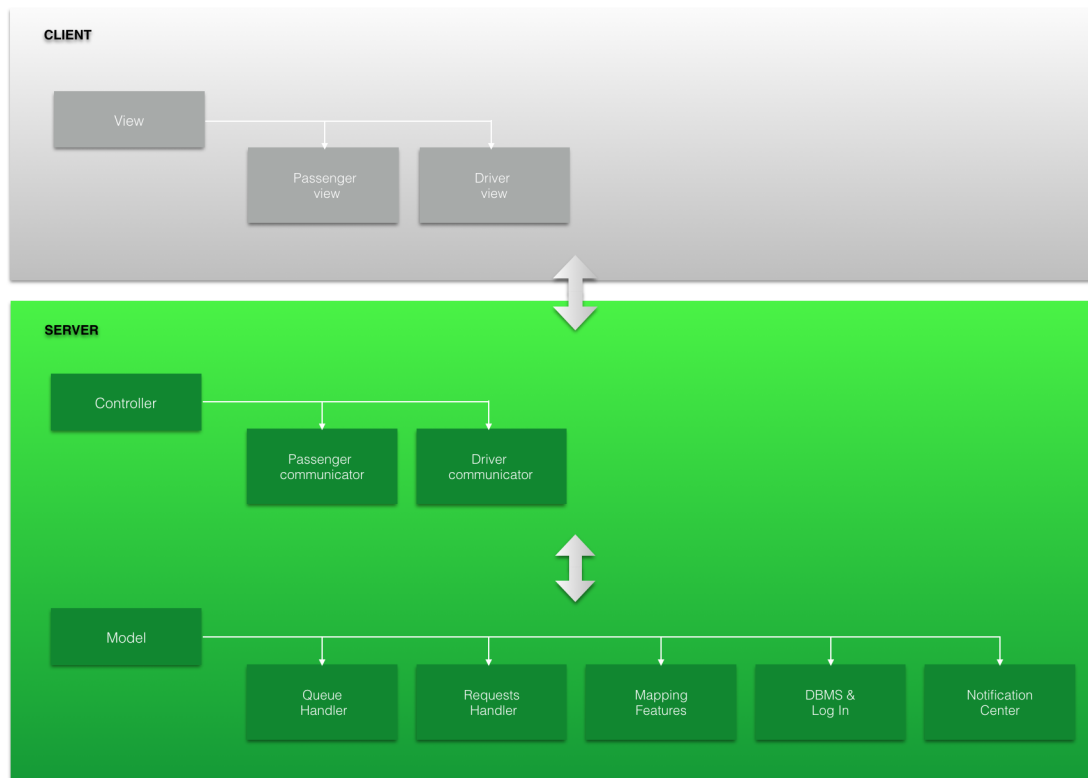
## 2.4   Component View

## 2.5   Deployment View

## 2.6   Run-time View

## 2.7   Component Interfaces

Here are stated the main functionalities provided by the component interfaces, for each one is stated the signature (in a java like pseudo-code) and a little explanation of how it works.

### 2.7.1   Queue handler interface:

```
1    public void addDriver(Driver d, Queue q)
```

**input:** It receives a driver class and a queue class instance as input.

**output:** -

**what he do:** this function adds a driver at the end of the queue.

```
1    public void removeFirstDriver(Queue q);
```

**input:** It receives the queue as input

**output:** -

**what he do:** this function remove the first driver of the queue.

```
1    public Driver selectFirstDriver(Queue queue)
```

**input:** It receives the queue as input

**output:** It return the first driver of the queue, and remove him from the queue.

**what he do:** -

```
1    public void moveFirstToBack(Queue queue)
```

**input:** It receives the queue as input

**output:** -

**what he do:** move the first driver from top of the queue to bottom.

### 2.7.2 Request handler

```
1    public static void handleRequest(Request request)
```

**input:** It receives the request and the queue to which is addressed as input.

**output:** -

**what he** The method implementation has to perform the correct handling of the various request type. It has to forward the immediate request to a Taxi Driver, acordingly to the constraints stated for this operation in the previous analysis. Otherwise it has to store the booking request in a suitable memory structure.

### 2.7.3 Shareable ride Finder

```
1    public static Request sharableCheck(Request request)
```

**input:** It receives the request as input.

**output:** It retu new Request object containing the overall number of passenger joining the ride, as the source and destination and all the other information are the same.

**what he do:** -

### 2.7.4 Mapping feature handler

This component is totally self contained, as it is a third parties services, so all the interface to interact with it are already provided by the Google map API documentation stated below:

- https://developers.google.com/maps/

### 2.7.5 DBMS and log-in

In order to ensure an high level security for this sensible component, it will be designed having a unidirectional communication. This in the sense that the only sensible data are sanded directly by the end-user towards our system and the system can only answer with Boolean values or not strictly related personal data. Acting in this way, the database system never directly send personal data on the network by itself.

```
1    public static boolean addNewUser(UserData uData)
```

**input:** It receives the users data as input.

**output:** It ret a Boolean value that state the success or the failure of the adding operation.

**what he do:** -

```
1    public static boolean userCredentialCheck(LoginData lData)
```

**input:** It receives the users data as input.

**output:** It return a Boolean value that state the success or the failure of the login operation.

**what he do**: -

### 2.7.6  Passenger and Driver Communicators

Actually in all the modern developing technologies they are already ensured by the developers tools and frameworks themselves through the use of listeners class. In any case listeners class can be overwrite in any moment forecasting some developing difficulties.

### 2.7.7  Views

Views as to be capable to receive the model output, and update the graphics feedback of the application. Due to this, view interface only to provide suitable methods to receive the message sent by the Model.

## 2.8  Architectural Styles and Patterns

Here are listed the main architectural styles and patterns choice. Every choice is adequately motivate and is made in order to simplify the design of intricate situations.

### 2.8.1  SOFTWARE

#### 2.8.1.1  MVC Pattern

MVC is an essential, rock solid choice for each distributed architecture. The pattern is perfectly suitable with the tier architecture model and represent the "best practice" choice since many years. Furthermore, every modern development environment automatically embedded itself this structure, so it is in any case you cannot help avoiding his use.

#### 2.8.1.2  Event Based Pattern

Event Based Pattern is implemented in the managing of the request. The passenger-user could be seen as a publisher in the publish-subscribe paradigms, and the taxi driver as the subscriber. In this pattern, a request play the role of the event, it is up to the system to ensure that actually not any taxi-driver subscriber can see the request, but only the first taxi driver in the right queue. We made this choice because firstly, the event base pattern, is another well known, reliable pattern. Secondly, in front of some little modifications in his implementation, it can greatly do his job.

### 2.8.1.3 Plugin Architecture

This architecture is implemented in order to ensure a chance to the future functionality expandability. Because of the forecast of a relative little future sets of different, available plugin, the performance degrading typical of this architecture, due to a massive installation of plugins, is not a big concern. The application will be released correlate with the essential extension point interfaces needed by this architecture.

### 2.8.1.4 SOA Architecture

The System adopt a proper service oriented architecture, either to use or offer services.

As stated in the RASD document, the whole map system is based upon the google map API, so the architecture is necessarily to allow the useful implementation of this features. As decided with our stakeholders, we are going to release our own programmatic interfaces. This will provides third party companies with our module, knowledge and algorithm carefully studied on this application case.

Follow the list of the API to be released:

- Queue Handler API,

- Shareable ride finder API,

- Request manager API.

About the major problems resulting from the adoption of this architecture:

- Security is guaranteed by a careful separation between the sensible data stored and managed only by the DBMS module and the data used by the services.

- The optimal reliability of the message exchanged among the various services shall be guaranteed by the handling of the sending/receiving failure.

- Quality of service with regards of the service adopted is guaranteed by the choice of well known ,widely used and reliable service. Whereas the quality of service released by us, is guaranteed by a careful planning and the adoption of all the necessary precaution.

- Service maintainability will be facilitated by the simplicity of the services offered.

## 2.8.2 HARDWARE

The Hardware Architecture will use the latest technology in the field of Cloud Computing. In particular it will be adopted the PAAS model.

The Platform as a service (PAAS) adoption will provides a platform allowing us to develop, run, and manage web applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching of an app. So our company will controls software deployment and configuration settings, and the provider will provides the networks, servers, storage and all the necessary services to host the application. This translates into:

- Significant cost reduction.

- Significant developing time reduction.

obviously with regards of the other architecture solution allowable (Hardware infrastructure of our own, IAAS cloud architecture etcetera).

# 3 — Algorithm Design

The goal of this section is to provide the most relevant algorithmic part of the project. In order to do this some code is stated relating to the most important modules analyzed in the chapter 2. Moreover the code is not complete. It can be considered as a first implementation which aims to give a general idea of the system core logic. The code is written in a Java-like pseudocode.

## 3.1 Request Management Algorithm

This algorithm, starting from an user request, shows how this can be handled by the system.

1. The system loads the Request Handler

2. The system calls the Handle Request procedure of the Request Handler, passing the request as a parameter

3. The method, relying on the type of request, decides how to manage it by calling the specific procedures

4. If the request is a "Booking request" the system stores properly it in the database

5. If the request is a "Taxi now request" the system checks for the sharing option status

6. If the sharing option is "true" the system starts the Shared Request Matcher Algorithm

7. The system sends the request to the first driver available. In order to do this the following procedure is done:

   - Extract the starting point from the request
   - Find the correct city zone associated to the starting point address
   - Extract the queue associated to the city zone
   - Extract the driver at the head of the queue
   - Forward the request to the driver

A possible implementation of the methods that can handle the procedures stated above is the following:

```
1   //handle a generic request
2   public void handleRequest(Request request){
3     if (request instanceof TaxiNowRequest)
4       forwardToDriver(request);
5     else
6       storeRequest((BookingRequest) request);
7   }
8
9   //add a booking request to the catalog
10  private void storeRequest(BookingRequest request) {
11    requestsCatalog.add(request);
12  }
13
14  //send the request to first queued driver in the corresponding
        city zone
15  private void forwardToDriver (Request request){
16    Request fwRequest = checkSharingOption(request);
17    for (CityZone zone: City.getCity()){
18      if (zone.containsAddress(fwRequest.getStartingPoint())){
                          QueueHandler.selectFirstDriver(zone.getQueue
      ()).sendMessage(fwRequest);
19      }
20    }
21  }
22
23  //check for sharing option and return an handled request
24  private Request checkSharingOption (Request request){
25    if (request.isShareable())
26      return ShareableRideFinder.generateShareableRequest(request);
27  }
```

## 3.2   Queue Management Algorithm

The input for this algorithm is a Queue element on which the operations are executed.

1. The system finds the correct queue using the Request Handler component

2. The system extracts the first driver in the queue. The extraction must be done in order to guarantee that two or more incoming requests are forwarded to distinct drivers.

3. The system forwards the request

4. The system starts a timeout and waits for the driver answer

5. If the time is up the driver must be re-added to the queue

6. Else, if the driver answer "accept", the system sets his status to unavailable, sends him all the informations such as the route and forwards a confirmation to the user

7. Else, if the driver answer "reject", the system re-added him to the queue

A possible implementation of the methods that can handle the procedures stated above is the following:

```java
//add a driver at the end of the queue
public void addDriver(Driver d, Queue queue){
  queue.add(d);
}

//extract a driver
public Driver extractFirstDriver(Queue queue){
  if (!queue.isEmpty())
    Driver d = queue.getFirst();
    queue.removeFirst();
    return d;
  else
    return null; //here an error message can be sent
}
```

## 3.3   Shared Request Matcher Algorithm

The input of this algorithm is a Request element. The goal is to merge shareable requests into one.

1. The system loads the Shareable Ride Finder component and use the request as a parameter

2. The Shareable Ride Finder checks if there's a pending request whit the same starting point and the same destination.

3. If some request is found, the number of passenger is updated in a new request

4. The new request is returned

A possible implementation of the algorithm is the following:

```java
//generate a new request merging the shareable ones
public Request generateShareableRequest(Request request){
  for (Request r: pendingRequests){
    if (canBeMerged(request,r) && r.isShereable()){
      request.psgNum() += r.psgNum();
    }
  }
return request;
}

private boolean canBeMerged(Request r1, Request r2){
  if (r1.getStartingPoint() == r2.getStartingPoint()
     && r1.getDestination() == r2.getDestibnation())
    return true;
  return false;
}
```