Design Document

Filippo Agalbato
850481

Andrea Cannizzo
790469

December 3, 2015

Contents

Chapter 1

Introduction

1.1 Purpose

This is the Design Document for the project software My Taxi Service. The goal of this document is to describe the system in term of architectural choices and design decisions, to provide an overview of the interactions between its components and the users, and to present samples of algorithmic pseudocode for the core systems. This document is targeted to the customer's project managers, to evaluate the project architecture from a high-level point of view, and to the intended designers, developers, programmers, analysts and testers, to build the actual software and to maintain, integrate and expand it in the future.

1.2 Scope

The aim of this project is to build a software system able to manage and optimize taxi services in a large city, targeted both to the city public transportation management council and to the citizens as customers of the taxi service. The system will allow passengers to use the service in a smart and accessible way, and at the same time will offer a better service thanks to an enhanced management of taxi resources and allocation.

In particular, passengers and taxi drivers will be registered and remembered by the system. Passengers will be able to request a taxi by using the application and taxi drivers will be notified of these requests by the system, that will divide the city in several taxi zones each with its own queue of taxis. Taxi drivers notified of passengers requests can then accept them.

1.3 Definitions, acronyms and abbreviations

- Passenger: A citizen recognized by the system as a registered user of My Taxi Service, using the system as a customer does, to call taxis and be served;
- Logged passenger: A passenger that is currently and correctly logged into the system;

- Taxi driver: A citizen recognized by the system as a registered user of My Taxi Service, using the system as a taxi driver, recognized by the parent company and of verified identity;
- Logged taxi driver: A taxi driver that is currently and correctly logged into the system;
- Request for service: A passenger uses the mobile app or web application to send the central management system a formal and well-formed request to be served by a taxi. This request includes a departure time, an itinerary starting point and ending point and the number of people to be served.
- Real time request: A request for service that is intended to serve a passenger immediately and is instantly relayed by the system to an available taxi driver;
- Reservation: A request for service that is not meant to serve a passenger immediately and as such is stored by the system and only relayed to an avaialble taxi driver once the correct time comes;

1.4 References

- Specification Document: My Taxi Service project specification for the Academic Year 2015-16 – *Assignments 1 and 2 (RASD and DD).pdf.*
- Agalbato, Cannizzo, *Requirement Analysis and Specification Document.*
- IEEE Std 1016-2009 Software Design Specifications.

1.5 Overview

This document is structured as follows:

- Introduction: A declaration of the scope, intent and purpose of this document;
- Architectural design: A high level view of the architecture of this system and a low level approach at its basic modules, with emphasis on their interaction, deployment, and lifecycle, including also a review of the architectural choices that were made;
- Algorithm design: A detailed look at some algorithmic design for some of the core systems;
- User interface design: An analysis of the needs of the user and how the system can meet them in terms of interface;
- Requirements traceability: An overview of how the functional requirements defined in the previous analysis map into the architectural choices.

Chapter 2

Architectural design

2.1 Overview

We have chosen to use a top-down approach, so we will start from a higher level of architectural design view and then we will analyse the system deeper.

We have chosen a four-tier architecture in order to keep a logical separation among data level, business level, communication level and client level.

Tier	**Layer**
Client	Terminal/Browser
Web	Application Server
Business	
Data	DBMS

Client Tier is the level used by the end user and it is located in the terminal mobile app or in the web browser:

Web Tier is the level that receives communication requests from client and communicates with the Business Tier to formulate responses. It is located in the application server;

Business Tier is the logical applicative level of the software. It analyses data received from the Data Tier and it provides info's to the Web Tier. It is located in the application server;

Data Tier is the *Model* in the *Model-View-Controller* pattern and it communicates with the Business Tier. It contains all the software data and is located in a DBMS.

2.2 High level components and their interactions

For each tier, we have identified a number of high-level architectural components.

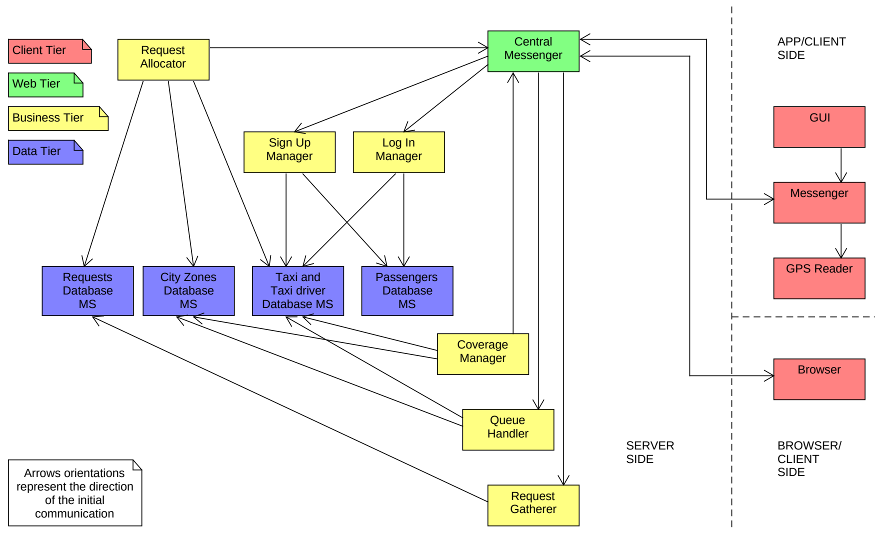


Figure 2.1: The different architectural components and their interaction

2.2.1 Client Tier

Graphical User Interface It is the graphical user interface of the mobile application;

GPS Reader It reads GPS information from the API's of the operative system;

Messenger It sends and receives messages from Web Tier Central Messenger and handles every message and operation of the Client Tier, in fact it communicates with GPS Reader and GUI;

Browser It communicates with Web Tier Central Messenger and it represent the User Interface alternative to mobile app.

2.2.2 Web Tier

Central Messenger It sends and receives messages from Client Tier and it handles every message and operation among Business Tier components. It also elaborates and sends pages to the Browser;

2.2.3 Business Tier

Sign Up Manager It manages the registration function and it operates with the Taxi and Taxi driver DBMS and the Passenger DBMS;

Log In Manager It manages the Log In function and it operates with the Taxi and Taxi driver DBMS and the Passenger DBMS;

Coverage Manager It controls and manages continuously the coverage of taxi in the city. It operates with the Taxi and Taxi driver DBMS and the City Zones DBMS;

Request Gatherer It collects the requests and reservations and it writes them in the Requests DB;

Request Allocator It reads from the Requests DB and it allocates requests to taxi drivers. So it also operates with Taxi and Taxi driver DBMS and City Zones DBMS;

Queue Handler It manages the allocation of taxi drivers in the correct city zone queue. It operates with Taxi and Taxi driver DBMS and City Zones DBMS.

2.2.4 Data Tier

Requests DBMS It's the database that contains all reservation and request information:

City Zones DBMS It's the database that contains all zones information and queues;

Taxi and Taxi driver DBMS It's the database that contains all taxi drivers information;

Passenger DBMS It's the database that contains the registered passengers information.

2.3 Lower level modules view

After the initial architectural structure was in place, the analysis was refined to identify, for each component, several lower-level modules that compose it.

2.3.1 Client Tier

Central Messenger

- A module that interfaces with GUI;
- A module that interfaces with GPS Reader;
- A module that interfaces with Server Central Messenger;
- A module that handles and forwards messages to modules.

GPS Reader

- A module that interfaces with Central Messenger in order to receive requests and send information;
- A module that reads GPS information from Mobile APIs.

Graphical User Interface

- A module that interfaces with Central Messenger in order to receive and send messages;
- A module that shows navigation pages.

2.3.2 Web Tier

Central Messenger

- A module that interfaces with Queue handler;
- A module that interfaces with Request Allocator;
- A module that interfaces with Request Gatherer;
- A module that interfaces with Coverage Manager;
- A module that interfaces with Sign Up/Log In manager;
- A module that interfaces with Client Central Messenger;
- A module that interfaces with Client Browser;
- A module that handles and forward messages throw modules.

2.3.3 Business Tier

Sign Up Manager

- A *Communication module* that interfaces with Central Messenger;
- An *Analysis module* that, reading form *Data module*, analyses the correctness of data and it eventually confirms or rejects;
- A *Data module* that stores data in the correct database and read needed information.

Log In Manager

- A *Communication module* that interfaces with Central Messenger:
- An *Analysis module* that, reading form *Data module*, analyses the correctness of data and it eventually confirms or rejects;
- A *Data module* that stores data in the correct database and read needed information.

Coverage Manager

- A *Data module* that interfaces with databases making DBMS queries;
- An *Analysis module* that analyses information received from *Data module* and calculates the best coverage;
- A *Communication module* that interfaces with Central Messenger in order to send messages to Taxi Drivers selected by *Analysis module.*

Request Gatherer

- A *Communication module* that receives requests messages from Central Messenger;
- An *Analysis Module* that analyses data integrity and requests information received from *Communication module* and eventually it sends to *Data Module*;
- A *Data Module* that stores information interfacing with databases.

Request Allocator

- A *Data Module* that, periodically reading from databases, controls if there are requests to fulfil and updates taxi drivers information;
- An *Analysis module* that choses taxi diver to allocate to the requests;
- A *Communication module* that interfaces with Central Messenger in order to send notifications to taxi drivers.

Queue Handler

- A *Communication module* that receives availability messages from Central Messenger;
- A *Data module* that store and updates information in database.

2.4 Deployment view

Two artifacts are going to be deployed, each with a different target.

The first will be the *server* artifact, which will operate on a cluster of servers and will be composed of all components from the, as mentioned, Business and Web tiers. These will form the server side of the whole system, and will interface with their respectived database components from the Data tier.

The second will be the *client* artifact, in the form of the My Taxi Service mobile app, which will run on smartphones and interface with the *server* artifact with the discussed methodologies, and will be made up of all previously highlighted components belonging to the Client tier.

2.5 Runtime view

We have produced some run-time sequence diagrams of the main operations that can be made with the software, namely:

- Receive and register a new reservation;
- Coverage cyclic checks and uptates;
- Taxi driver queue updates;
- Reservation allocations.

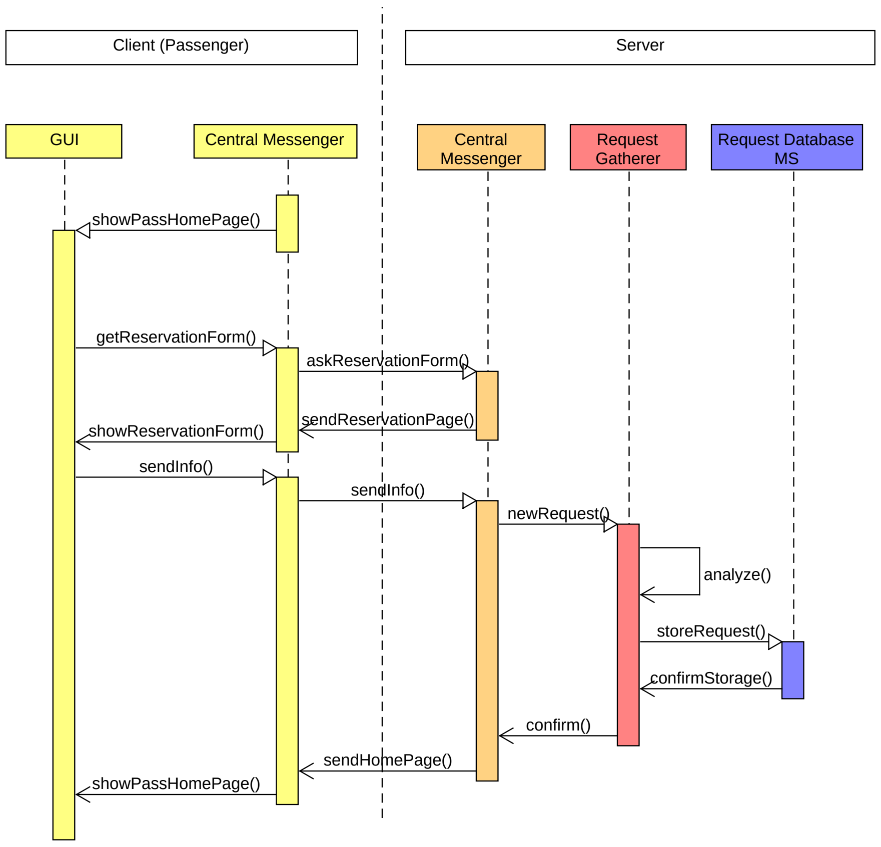


Figure 2.2: New reservation registration

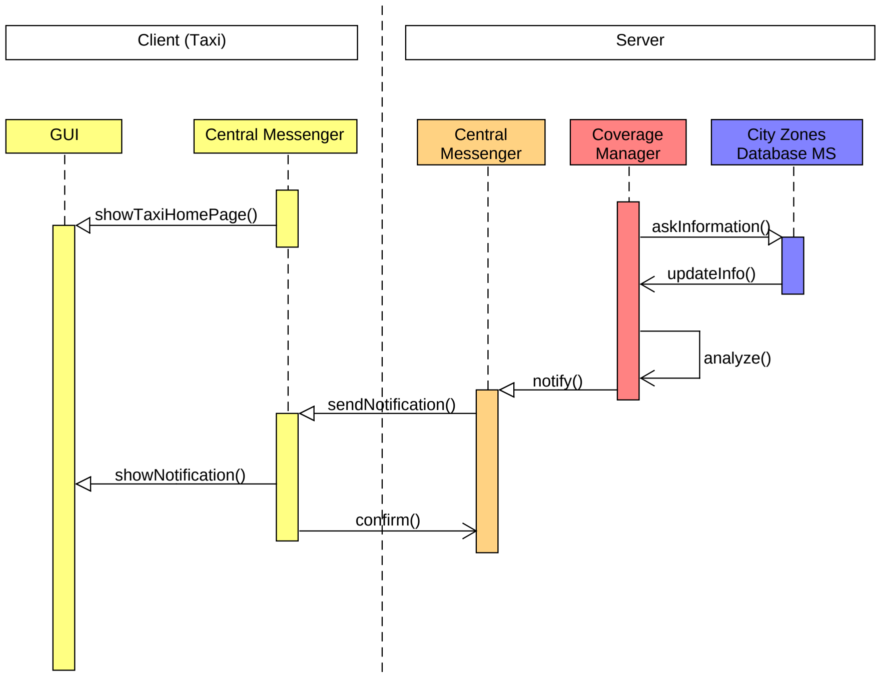


Figure 2.3: Coverage updates

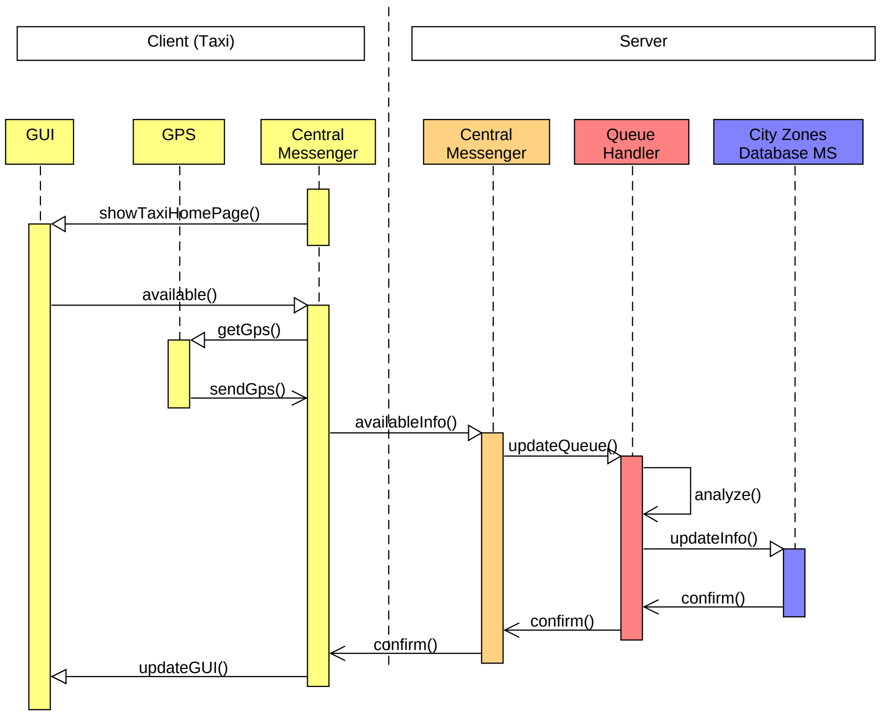


Figure 2.4: Taxi driver queues updates

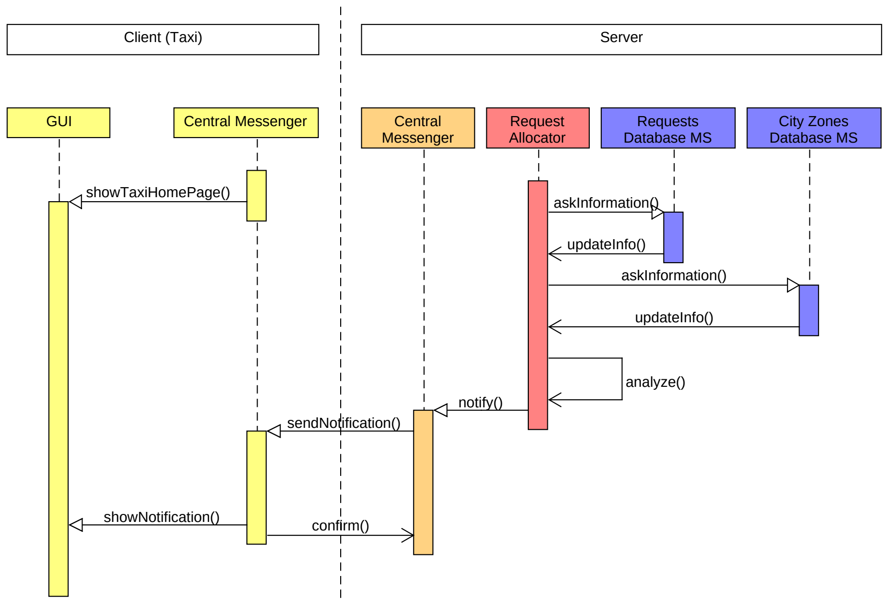


Figure 2.5: Reservation allocations

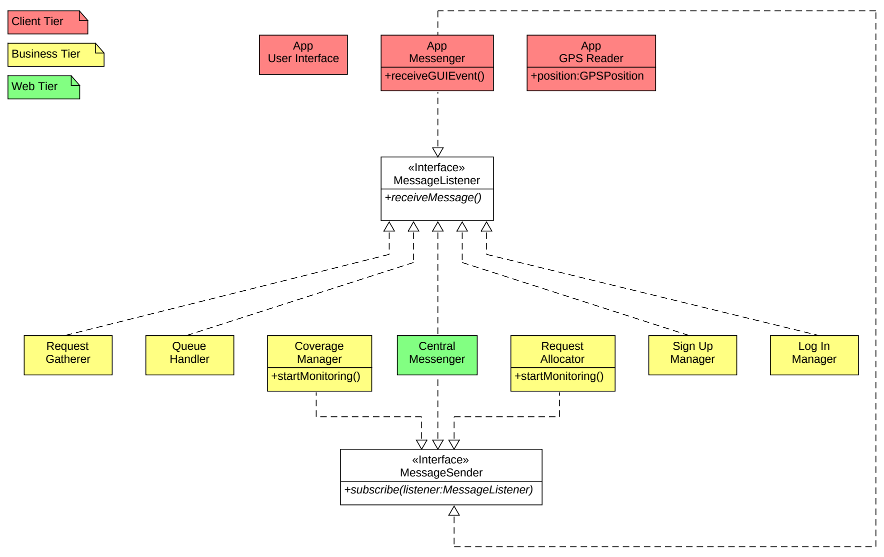


Figure 2.6: Component interfaces

2.6 Component interfaces

An Event-based style being used, components have little to no public interfaces to show to the outside world. Every information exchange taking place in the system, any order being given and received, everything is passed along as a *message* event, to which each component, upon inspecting its contents, will react according to what it knows it needs done. As such, every representation of such interfaces is by definition barebone, but a diagram is included for the sake of clarity.

2.7 Architectural styles and patterns

With regards to the general architecture, we found the problem at hand to be *intrinsically* made to be solved by a client-server approach, and moreso we developed it in the particolar flavor of *fat server* (since the application server runs the data tier, web tier, business tier) and *thin client*. This architecture is exceedingly well supported and ready to be developed with JEE.

Additionally, we found most natural to design an event based architecture through an intense flow of messages among components. We choose this style because our components are very independent and in most of cases they interact asynchronously with each other.

2.8 Other design decisions

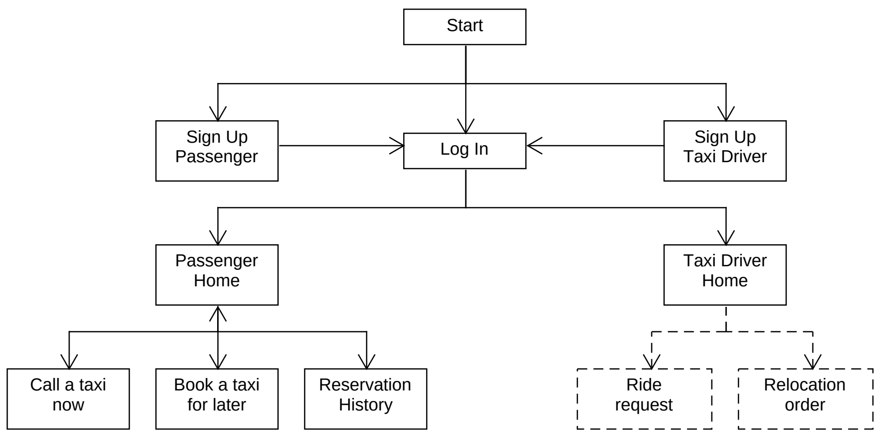
We have paid particular attention in designing a very modular architecture favoured by having chosen a top-down approach. This disposition has been selected in order to facilitate future extensions.

Chapter 3

User interface design

The following are simple mockups of how the My Taxi Service app should appear. In the case of the webapp, the required design is very much the same.

First, is a general overview of how the different screens are navigated by the user.



In the following pages, several mockups of the main screens.

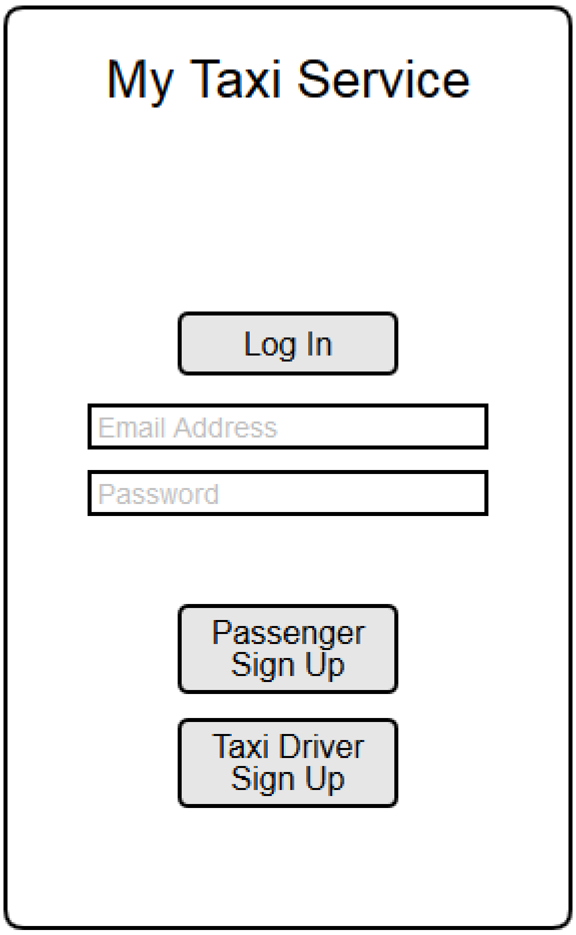


Figure 3.1: Sign Up / Log in view

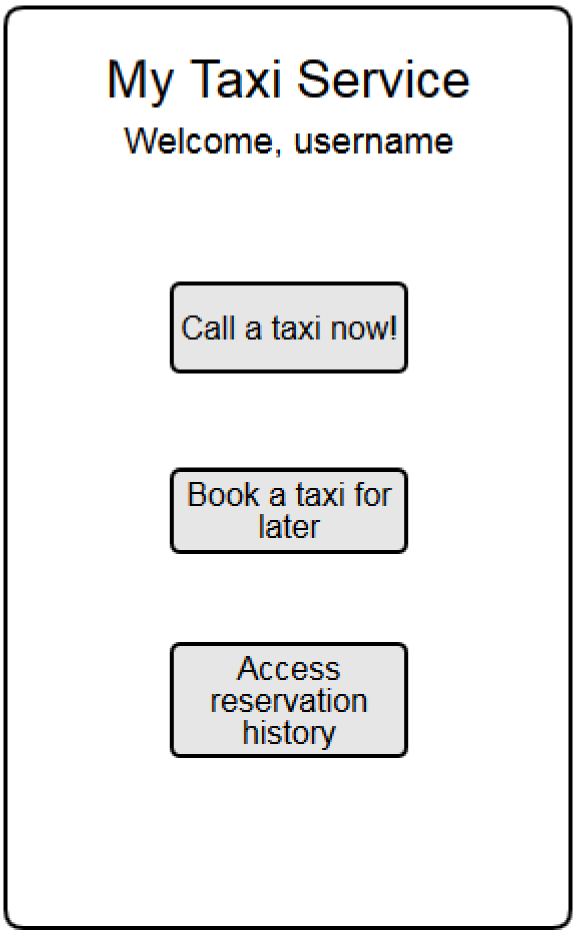


Figure 3.2: Passenger home

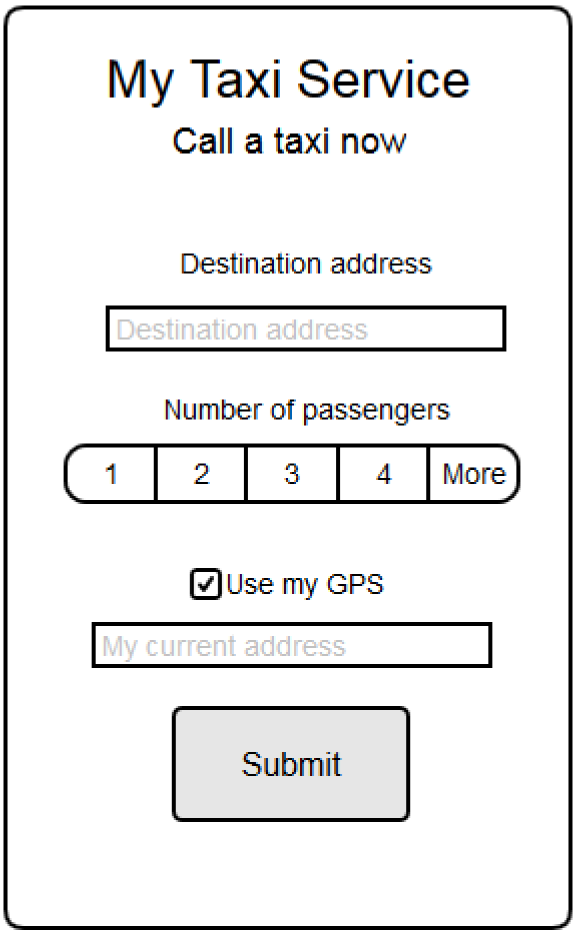


Figure 3.3: Call a taxi now function view

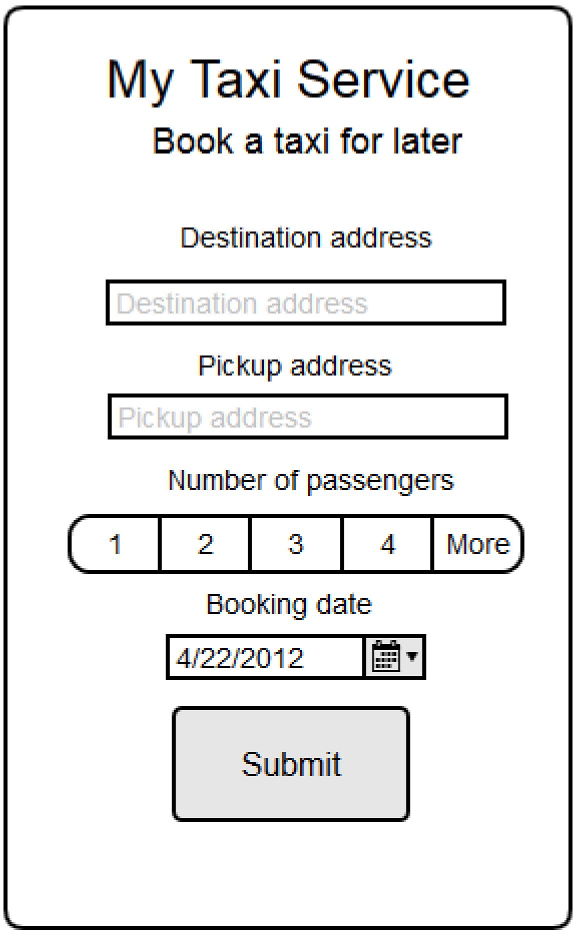


Figure 3.4: Booking function view

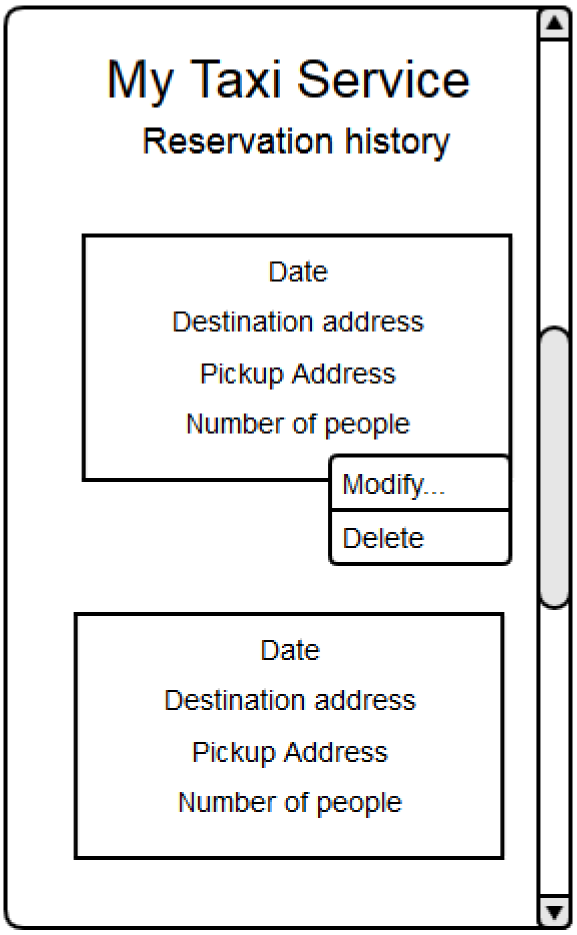


Figure 3.5: Reservation history view

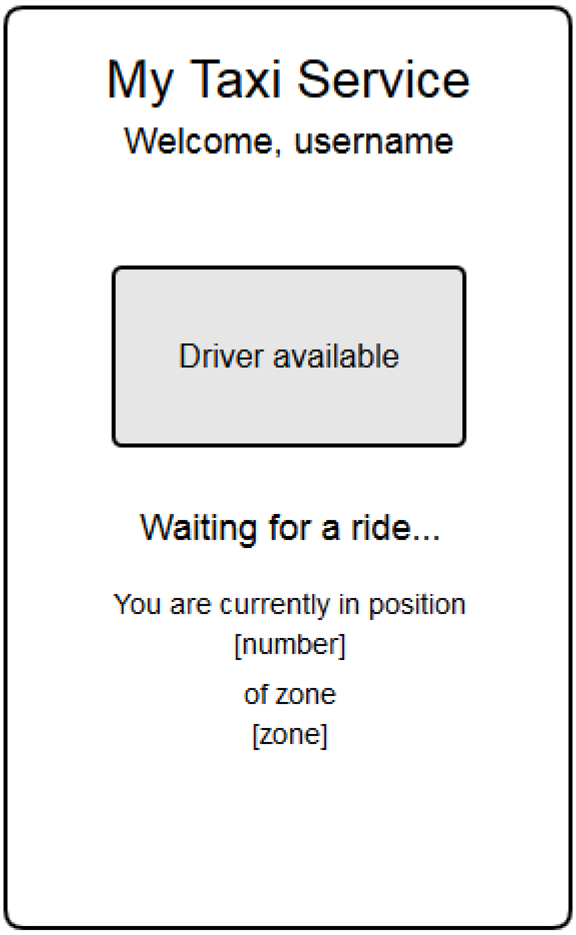


Figure 3.6: Taxi driver home

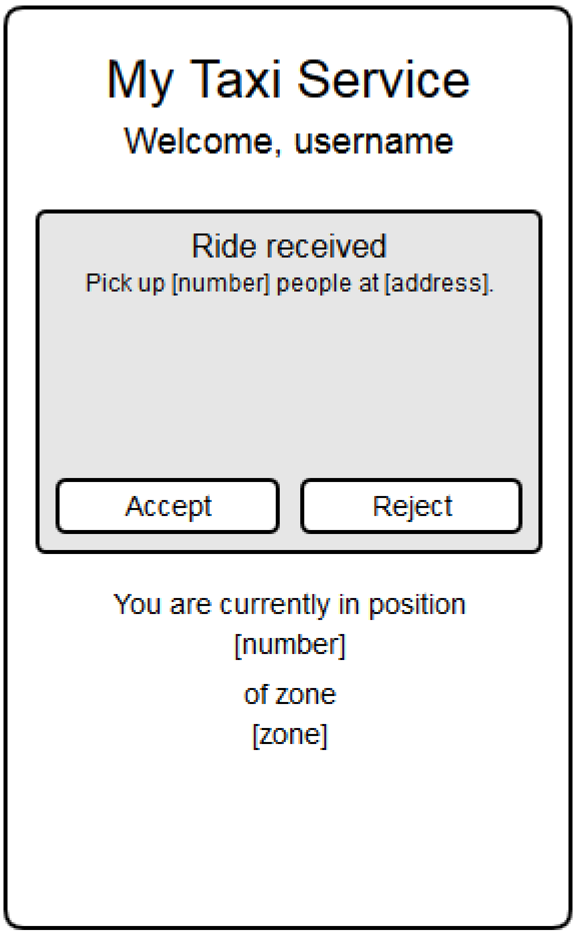
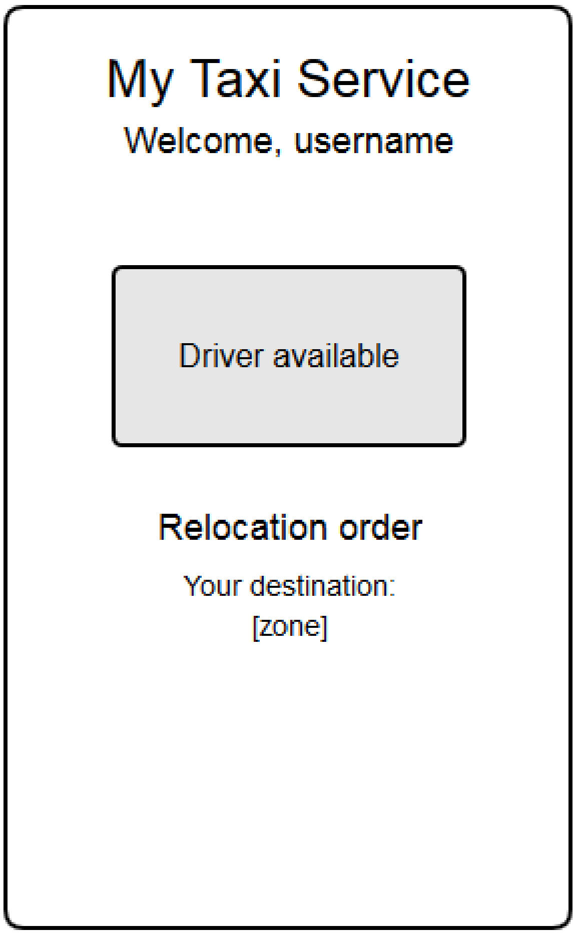


Figure 3.7: Taxi driver receives a ride request



My Taxi Service

Welcome, username

Relocation order

Your destination:
[zone]

Figure 3.8: Taxi driver receives a relocation request

Chapter 4

Algorithm design

The algorithms provided here are written in a Java-like language, with classes and methods, to provide samples of the core functionalities.

4.1 Main controller

Core functionalities that bind together the system.

```
public class Controller {
  public static Zone[] map; // list of zones
  public static int zones; // number of zones
  public Taxi[] toMove; // list of Taxi to be moved
     from a zone to an other
  public int toMovePointer; // pointer of toMove list

  public int contaDisponibili(){

    //it counts every available taxi in the city and
       returns that number

    int a = 0;
    for(int i=0; i<zones; i++){
      a=a+map[i].contaTaxi();
    }
    return a;
  }



  public int totCoeff() {

    // it calculate the sum of the coefficients of
       every zone and returns that number

    int a = 0;
    for (int i=0; i<zones; i++){
      a = a + map[i].coefficient;
    }
    return a;
  }


  public int allocTaxi(int zone) {

    // it calculate how many taxis should be available
        in the zone given as parameter

    int a = 0;
    a = (contaDisponibili() / totCoeff()) * map[zone].
       coefficient;
    return a;
  }

  public int taxiToMove(int zone) {

    // it adds taxis that have to be moved from the
       zone given as parameter to the list "toMove"
       and return number of taxis added

    int a = 0;
    if (allocTaxi(zone) < map[zone].contaTaxi()){
      a = map[zone].contaTaxi() - allocTaxi(zone);
    }

    for (int i=0; i<a;){
      if (map[zone].list[i].available){
        toMove[toMovePointer] = map[zone].list[i];
        i++;
        toMovePointer++;
      }
    }
    return a;
  }

  public int reqZones(int zone) {

    // it allocates taxis to the new zone that
       requires them and return number of taxis
       allocated

    int a = 0;
    if (allocTaxi(zone) > map[zone].contaTaxi()){
      a = allocTaxi(zone) - map[zone].contaTaxi();
    }
    for (int i=0; i<a;){
      toMove[toMovePointer].zoneToGo = zone;
      toMovePointer--;
    }

    return a;
  }



  public void cover(){

    // it moves taxis to ensure the established
       coverage

    for(int i=0; i<zones; i++){
      taxiToMove(i);
    }
    for(int i=0; i<zones; i++){
      reqZones(i);
    }
  }




  public static void move(Taxi t, int z1, int z2){

    // it moves a taxi from a queue to an other

    for (int i=0; i<map[z1].pointer; i++){
      if (t == map[z1].list[i]){

        map[z2].list[map[z2].pointer] = map[z1].list[i
           ];
        map[z2].pointer++;
        map[z1].list[i] = map[z1].list[map[z1].pointer
           ];
        map[z1].pointer--;
      }
    }
  }


}
```

4.2 Reservations queues

How taxis are allocated to requests.

```
public class ReservationQueue {
  public Reservation[] queue; // list of reservations
  public int pointer; // pointer of the queue


  public void fulfill(Time t){

    // it allocates a taxi to the reservation that has
        to be fulfilled in that time

    for (int i=0; i<pointer; i++){
      if (queue[i].startingT == t){
        for (int j=0; j< Controller.map[queue[i].zone
            ].pointer; j++){
          if (Controller.map[queue[i].zone].list[j].
              available){

            queue[i].server = Controller.map[queue[i].
                zone].list[j];
            Controller.map[queue[i].zone].list[j].
                toServe = queue[i];
            Controller.map[queue[i].zone].list[j].
                available = false;
            queue[i] = queue [pointer];
            pointer--;
          }
        }
      }
    }

  }

}
```

4.3 Taxis as entities

How taxis are treated by the system.

```
public class Taxi {
  public int id; // identification number of the taxi
  public boolean available; // if the taxi is
      available
  public int myZone; // identification number of the
      zone in which the taxi is
  public int zoneToGo; //identification number of the
      zone in which the taxi has to move
  public Reservation toServe; // the reservation that
     he has to be fulfilled


  public void move() {
    Controller.move(this, myZone, zoneToGo);
    this.myZone=zoneToGo;
  }
}
```

4.4 Zones as entities

How city zones are treated by the system.

```
public class Zone {
  public Taxi[] list; // list of taxis
  public int pointer; //pointer of taxis list
  public int coefficient; // coefficient of density of
      the zone (to proportion the distribution)

  public int contaTaxi(){

    // it counts the number of available taxis in the
       zone and returns that number

    int a = 0;
    for(int i=0; i<pointer; i++){
      if (list[i].available){
        a++;
      }
    }
    return a;
  }
}
```

Chapter 5

Requirements traceability

To ensure requirements traceability we pointed out, for each of the requirements identified in the Requirement Analysis and Specification Document for this project, which architectural components (defined in section 2.2) are involved in their satisfaction and implementation. We omitted to explicitly include the *Central Messenger* because it is involved in every functional requirement and we also omitted to explicitly include databases because we have already discussed their strict connection with components of the *Business tier*.

1. The system must balance taxi drivers workload by assigning each one to a *first-in, first-out* queque in a city zone – **Queque handler**;

2. The system must provide a sign up function for passengers – **Sign-Up Manager, Graphical User Interface**;

3. The system must store all required passenger information – **Sign-Up Manager**;

4. The system must provide a log in function to access all passenger features – **Log-In Manager, Graphical User Interface**;

5. The system must provide a function that allows logged passengers to require a taxi – **Request Gatherer, Graphical User Interface**;

6. The system must provide a form in which the logged passenger will be able to add trip information for a request for service (starting point, destination point, number of passengers) – **Request Gatherer, Graphical User Interface**;

7. The system must retrieve GPS information from the logged passenger's mobile phone – **GPS Reader**;

8. The system must provide a reservation function – **Request Gatherer, Graphical User Interface**;

9. The system must provide a form in which the logged passenger will be able to add trip information for a reservation (starting point, destination point, number of passengers, leaving time) – **Request Gatherer, Graphical User Interface**;

10. The system must finalise a reservation two hours before its requested time, making it unchangeable – **Request Allocator**;
11. The system must store a reservation history for each passenger – **Request Gatherer**;
12. The system must provide a function which shows reservation history – **Request Gatherer, Graphical User Interface**;
13. The system must provide a function to allow logged passengers to modify a reservation up to two hours before the requested time – **Request Gatherer, Graphical User Interface**;
14. The system must provide a function to allow logged passengers to delete a reservation up to two hours before the requested time – **Request Gatherer, Graphical User Interface**;
15. The system must analyse a newly-made real time request and send a confirmation with the estimated waiting time if it can be fulfilled – **Request Gatherer, Graphical User Interface**;
16. The system must analyse a newly-made reservation request and send a confirmation if it can be fulfilled – **Request Gatherer**;
17. The system must notify passenger if his reservation cannot be met – **Request Allocator, Graphical User Interface**;
18. The system must delete all reservations in a passenger's history that cannot be met after sending him notice – **Request Gatherer**;
19. The system must provide a sign up function for taxi drivers – **Sign-Up Manager, Graphical User Interface**;
20. The system must store all required taxi driver information – **Sign-Up Manager**;
21. The system must check that all taxi drivers trying to sign up own both a valid taxi driver's license and a proper driver's license – **Sign-Up Manager**;
22. The system must check that the information needed for a taxi driver sign up points to a specific person in the company that has not already registered before confirming – **Sign-Up Manager**;
23. The system has to provide a log in function to access all taxi drivers features – **Log-In Manager, Graphical User Interface**;
24. The system must provide a function to allow logged taxi drivers to inform the system of their availability – **Queue Handler, Graphical User Interface**;
25. The system must notify taxi drivers of an incoming request for service – **Request Allocator, Graphical User Interface**;
26. The system must provide all necessary information for taxi drivers to carry out requests – **Request Allocator, Graphical User Interface**;

27. The system must provide a function to allow taxi drivers to confirm that they are going to take care of an assigned request for service – **Request Allocator, Graphical User Interface**;
28. The system must provide an explicit *decline* function to allow taxi drivers to notify that they are not going to take care of an assigned request for service – **Request Allocator, Graphical User Interface**;
29. The system must flag a taxi driver as having declined an assignment if no answer is received within 30 seconds – **Request Allocator**;
30. The system must move a taxi driver that has declined an assignment at the end of their queue – **Queue Handler**;
31. The system must analyse taxi locations and calculate their best possible distribution – **Coverage Manager**;
32. The system must choose which taxi drivers need to be moved to ensure total coverage of the city – **Coverage Manager**;
33. The system must notify taxi drivers in which city zone they have to move, as needed – **Coverage Manager, Graphical User Interface**.

Appendix A

Document and work information

A.1 Revisions

This is the first version of this document. There are currently no revisions.

A.2 Tools used

TeXworks editor With PDFLaTeX, for composing and editing this document;

UMLet For drawing all kinds of diagrams;

Moqups For drawing mockups.

A.3 Overall time spent

The authors spent about 30 hours of their time, equally divided among them, working on this document.