Design Document
myTaxiService

Luca Nanni (850113) Giacomo Servadei (854819)

December 4, 2015

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Purpose

Here we present the Design Document of the application *myTaxiService*. The aim of this document is to show our design choices and the rationale behind them. We will focus mainly on architectural choices such as the subdivision into modules of our software, the deployment of this modules and the communication between them.

1.2 Scope

This document is intended to be a description of the architecture of the software system *myTaxiService*, commissioned by the government of a big city which wants to improve and automatize the managing of taxis and the calling system for the users.

In particular we will present different *views* of the system representing different levels of abstraction: the user's point of view, an internal sight of the subsystems and their high level interaction and the communication interfaces that they use to interact. We will describe the chosen architectural styles and pattern, some fundamental algorithms and how our choices are mapped on the requirements elicited in the RASD (Requirement Analysis and Specification Document).

1.3 Definitions

For the definitions and the glossary we refer to the Requirement Analysis and Specification Document.

1.4 Referenced documents

- *Requirements Analysis and Specification Document - myTaxiService*
- *Assignment 2 document*

1.5 Document structure

The following document is structured in three important sections:

2. The architectural description (chapter 2)
3. The algorithm description (chapter 3)
4. The user interface design (chapter 4)
5. The mapping between this design and the requirements (chapter 5)

Chapter 2

Architectural Design

2.1 Overview

Our first sight of the system is from the highest point of view. Here we describe the architecture of the system from a *layer* view.

2.1.1 An overview of our system

In the specific case of *myTaxiService* we mapped the previous architecture in this way:

- **Presentation tier**:
  - *Passenger*:
    - Dedicated Mobile application
    - Browser
  - *Taxi Driver*:
    - Dedicated Mobile application
- **Application tier**: main server(s) located in some office of the government of the city or (better), a virtual server hosted by some expert company and accessed through the Internet.
- **Data tier**: main database(s) located in some office of the government of the city or (better), a virtual database hosted by some expert company and accessed through the Internet.

2.2 High level components and their interaction

We can sketch our system like in figure 2.1

Here we can see the different components, both hardware and software, and how they are interconnected from a very high point of view.

In particular the web browser will interface the business logic through a web server, which implements services like Servlets and JSP. In this case, so, the chosen protocol will be HTTPS. As far as mobile applications are concerned they will interface the system through remote calls.

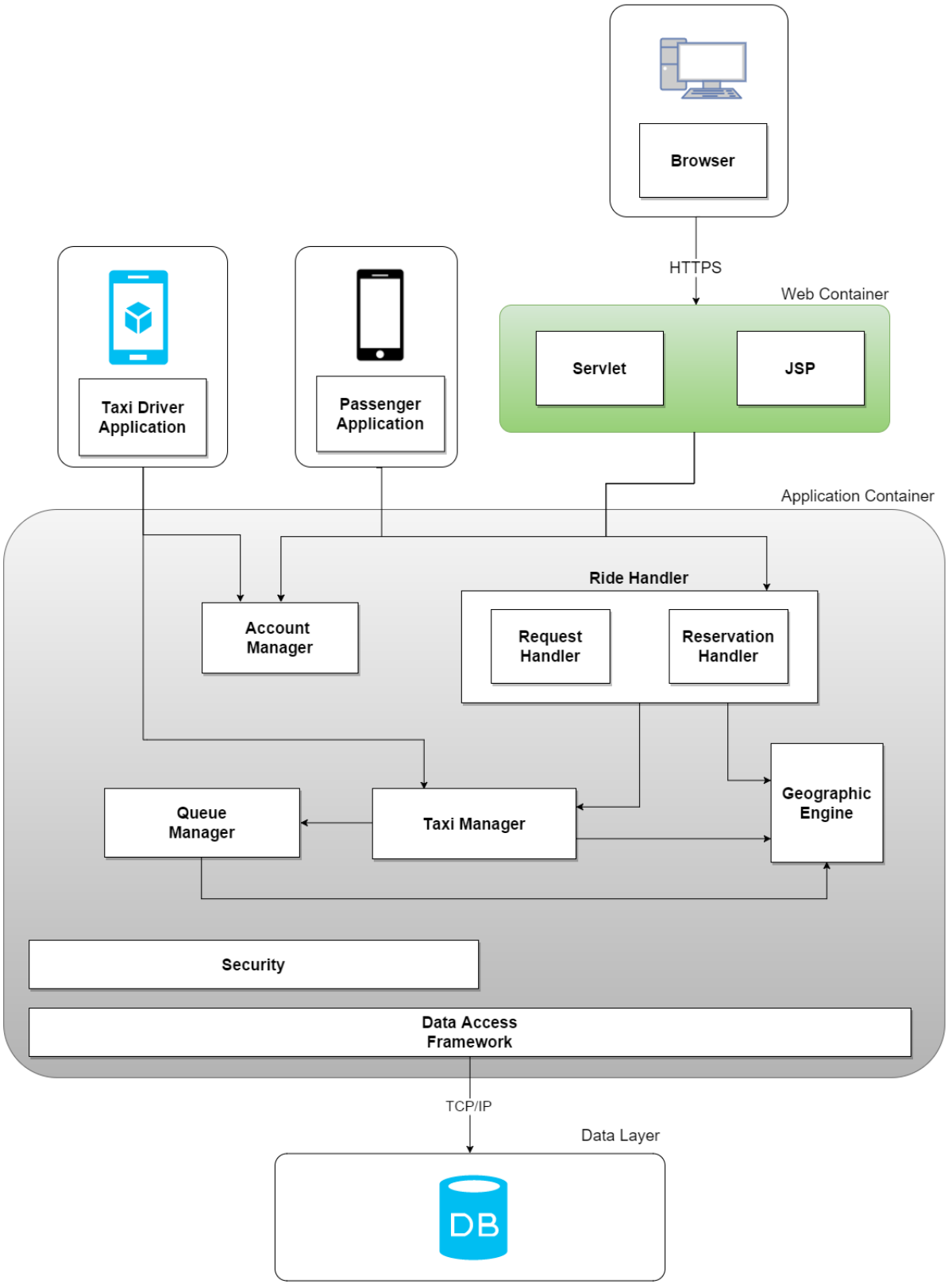


Figure 2.1: Architecture Overview

The application server will be divided in software components differentiated by their role in the implementation of the business logic. For data access and security will be used ready-made components provided by the JEE infrastructure.

The application server and the data layer will communicate through a TCP/IP connection.

2.2.1 Business logic components

In figure 2.2 we find an overview of the business logic components of our application.

1. **Account Manager**: It manages the accounts, both of Passengers and Taxi Drivers. It also handles the login and logout phases of both users and the registration for the Passenger and the relative account deleting. It exposes the following interfaces:
   - `Passenger`: for the interaction with the Passengers
   - `Taxi Driver`: for the interaction with the Taxi Drivers
2. **Ride Manager**: this component is delegated to the management of the life cycle of the incoming requests and reservations from the passengers. From this point of view it can be divided in two main subcomponents:
   - *Request Handler*
   - *Reservation Handler*

   It exposes the following interfaces:
   - `RideHandler`: which can be used by the Passenger mobile application and the web browser (web server)
3. **Taxi Manager**: manages the life cycle of the taxi drivers, starting from the availability status and the retrieving of their location. It is in charge of forwarding to the taxi drivers the calls from the passengers.
   It exposes the following interfaces:
   - `StatusHandler`: to communicate with the Taxi driver
   - `TaxiProvider`: to communicate with the RideManager
4. **Queue Manager**: it is in charge of the management of the taxi queues in the different zones of the city, applying the decided policy.
   It exposes the following interfaces:
   - `TaxiManagement`: to communicate with the TaxiManager
5. **Geographic Engine**: the roles of this component are to implement the algorithms to calculate the zone corresponding to a particular location. It is also in charge of the geographic definition of the taxi zones.
   It exposes the following interfaces:
   - `ZoneLocator`: which provides caulculations methods to the other components
   - `TimeCalculator`: which provides approximative traveling time calculation methods
6. **Security**: a ready-made software component to ensure the safety of tcp/ip connections from the mobile devices and from the web browser

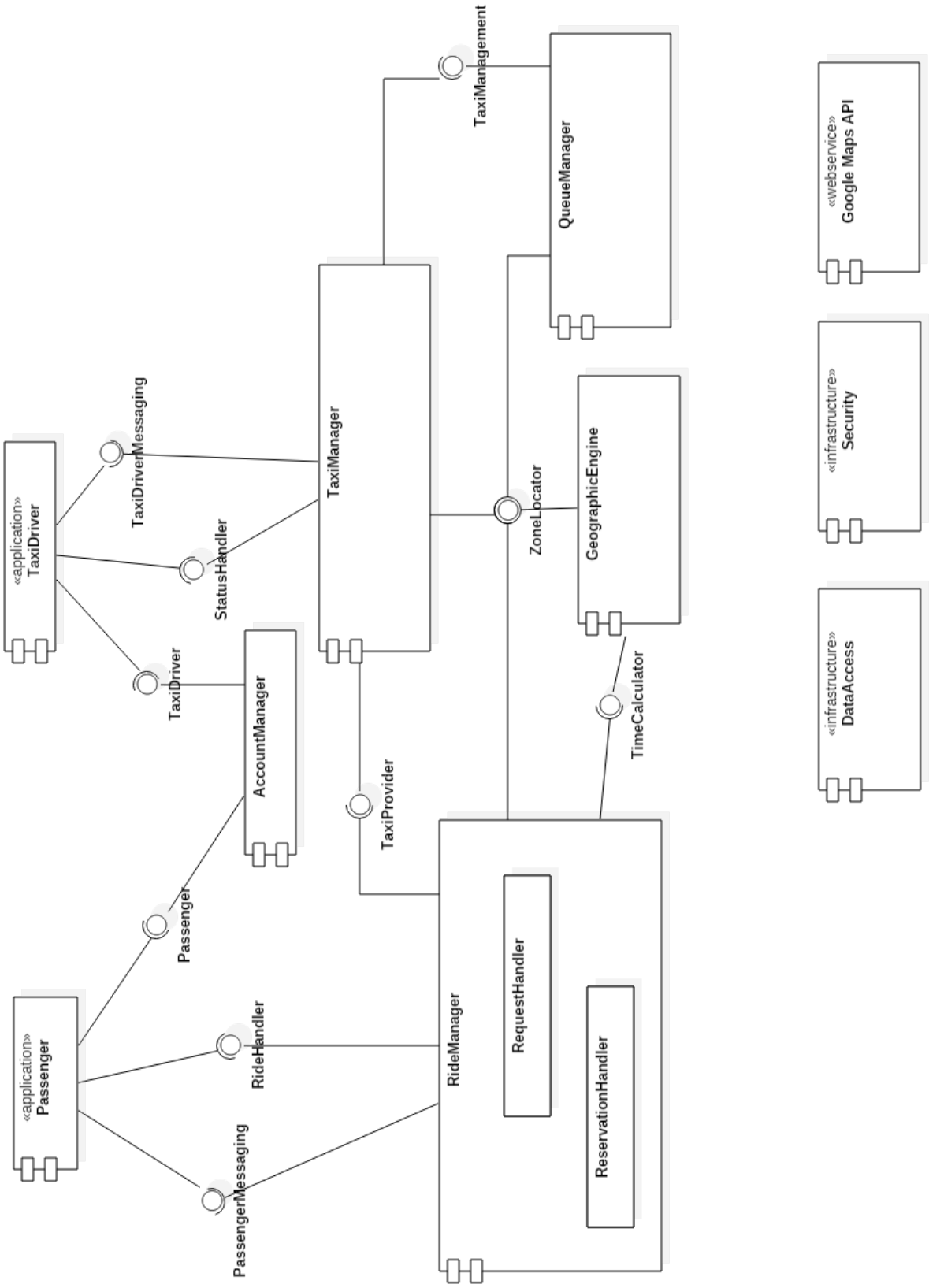


Figure 2.2: High level components of the business logic

7. **Data Access**: framework that abstracts the underlying database logic and enables to map the business logic entities into tables.

2.3 Component view

Now we refine the description of the components of the system.

2.3.1 Account Manager

Internal Components

The Account Manager is responsible for all the actions related to users and their account, such as login, logout registration and deletion of an account. Internally it is composed of the following components each with a precise job:

- *AccountController*: it exposes the interfaces for external calls and manages them. It has access to the Data Access interface in order to add or remove accounts, and to check the validity of credentials during the login phase.
- *SessionManager*: during the login phase, if the credentials has been recognized by the AccountController, the session manager, through the Security layer, provides a valid session to return to the passenger application for future interactions.

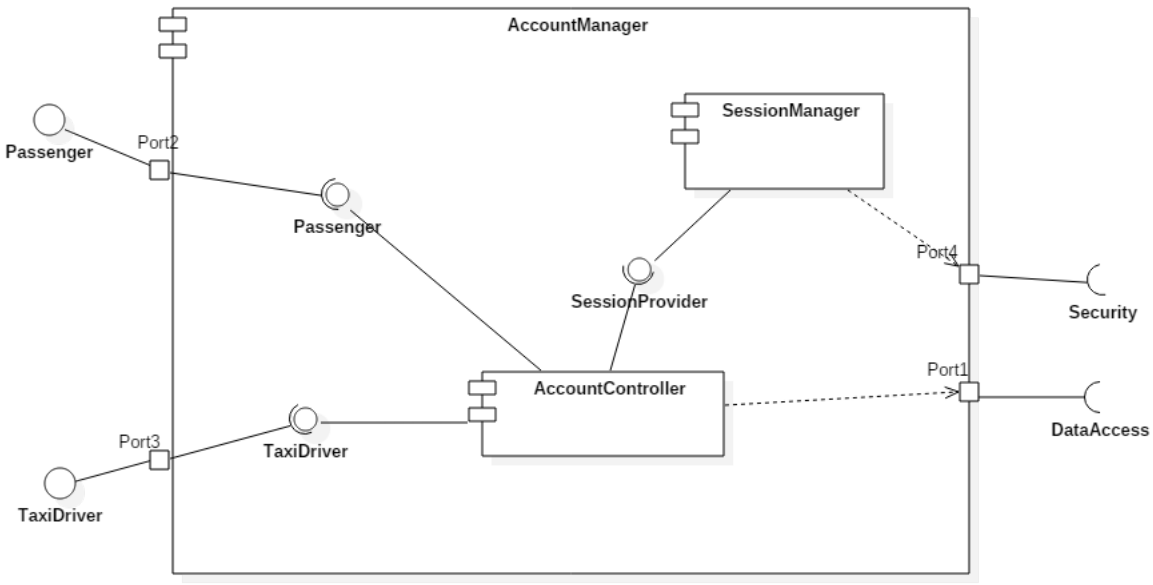


Figure 2.3: Account Manager

Provided interfaces

Provided Interface	Dedicated user	Description
Passenger	Passenger's application and relative web application	Login, logout, registration and deletion of the account
TaxiDriver	Taxi driver's application	Login, logout

Table 2.2: Account Manager: provided interfaces

Required interfaces

Required Interface	Description and usage
DataAccess	Access to the data layer in order to retrieve the account information of the Passengers and the Taxi Drivers
Security	It asks the security layer to provide for a session during a log in request

Table 2.4: Account Manager: required interfaces

2.3.2 Ride Manager

Internal Components

The Ride Manager is responsible for the communication with passengers, and for the management of the life cycle of requests and reservations. Internally it is composed of the following components each with a precise job:

- *RideDispatcher*: it exposes the interfaces for external calls and dispatches them to the right subcomponent depending on the nature of the call.
- *RequestHandler*: for every request it receives, it first checks for the validity of the location, then if this is valid it stores the request through the Data Access. Right after that, it forwards the request to the requestQueueManager. If the requestQueueManager finds a taxi driver, the handler computes the waiting time through the Geographic Engine, and returns to the passenger an affirmative response with the info, otherwise a negative response. The method for making a request is synchronous, this means that the passenger gui will wait until it gets a response by the server.
- *ReservationHandler*: for every reservation it receives, it first check for the validity of the location, then if this is valid it stores the reservation through the Data Access. It then waits until 10 minutes before the meeting time and then it forwards the reservation (in the form of a request) to the requestQueueManager (for further details about the algorithm that manages the wait for every reservation see 3.2). It keeps forwarding the reservation until it gets a taxi driver, then it computes the waiting time through the Geographic Engine, and then sends an affirmative message to the passenger with the info. The method for making a reservation is carried out in two phases. The first is synchronous, this means that the passenger gui will wait for the affirmative response after submitting the reservation. The

second is synchronous, this means that the gui will not wait for a response from the server but the server will just send a message with the info when needed.

- *RequestQueueManager*: it is in charge of calling the provideTaxi method of the Taxi Manager component. It is separated from the two handlers because the idea behind seeking a taxi is the same between requests and reservations and in future possibly it can also add some priority policies.

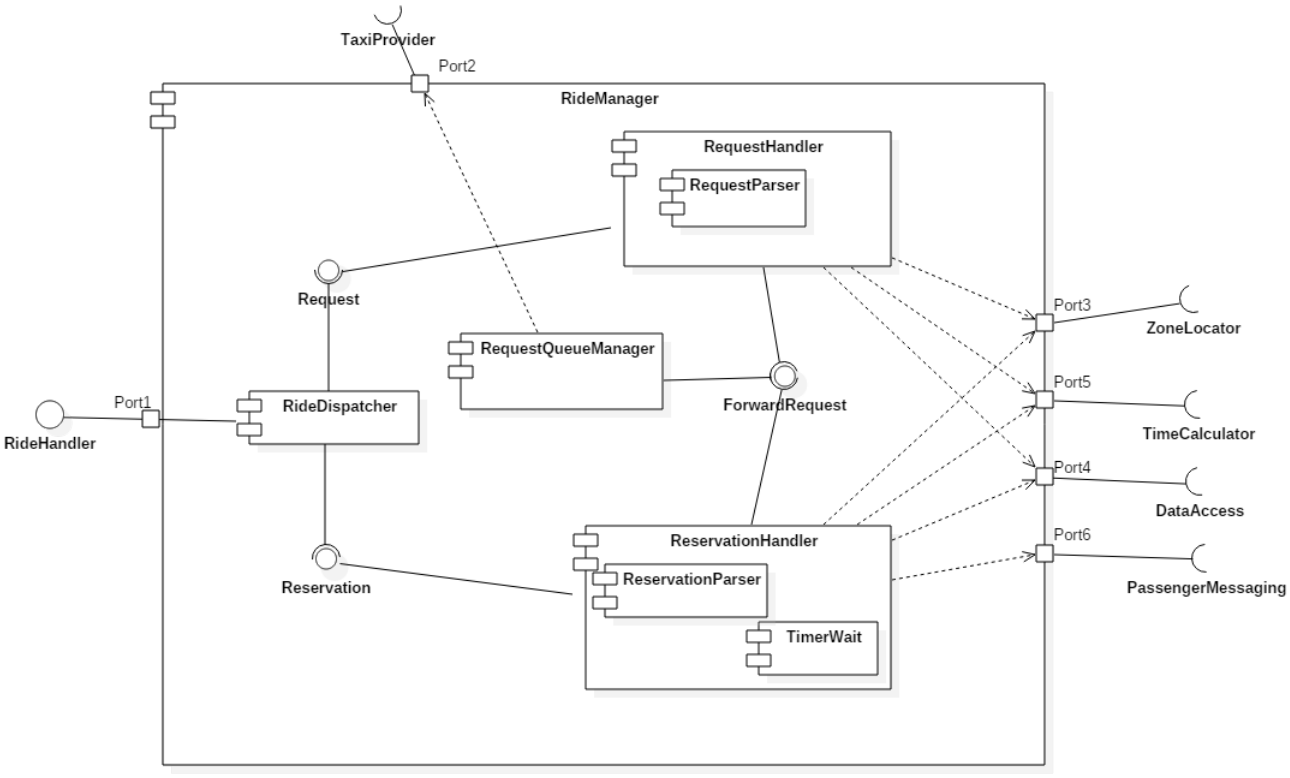


Figure 2.4: Ride Handler

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
RideHandler	The Passenger's application and the relative web application	Login, logout, registration and deletion of the account

Table 2.6: Ride Handler: provided interfaces

Required interfaces

Required Interface	Description and usage
DataAccess	Access to the data layer in order to • Store data of the request • Store data of the reservation • Retrieve reservations in order to forward them to the TaxiManager
Zone Locator	Check the correctness of the location data provided by the user
Time Calculator	Calculate the approximative waiting time
TaxiProvider	Retrieve a taxi who accepted the request/reservation or an error message saying the nature of the problem.
PassengerMessaging	Sends to the passenger the info for the taxi driver who is going to serve a reservation.

Table 2.8: Ride Handler: required interfaces

2.3.3 Taxi Manager

Internal Components

The Taxi Manager is responsible for the communication with taxi drivers, and for the management of their life cycle. Internally it is composed of the following components each with a precise job:

- *TaxiLifeCycleManager*: it manages the life cycle of taxi drivers. In order to do so it exposes an interface for setting the availability status, and for updating the location. It stores all the informations related to taxi drivers through the DataAccess. If a taxi driver sets himself as Available, the component will call the add method of the QueueManager. Oppositely if the taxi driver sets himself as NotAvailable, it will call the remove method. When a taxi driver updates his location, the component computes the zone through the GeographicEngine; if it notices that the zone has changed, it calls the changeZone method of the Taxi Manager. It also exposes an internal interface to the TaxiDispatcher, which is meant to be used to set the status of a taxi driver as Busy once he accepts a request.
- *TaxiDispatcher*: it exposes the interface for providing a taxi driver for a location. Once it has been called he first computes the corresponding zone for the location provided, then it starts a loop where it calls the QueueManager for getting a taxi. If there are no taxi available in that zone he returns a negative message, otherwise it forwards the request to the taxi driver it got as a response. If the taxi driver accepts, it set his status as Busy through the TaxiLifeCycleManager and return him to the caller. If the taxi driver does not accept, it starts the loop again.

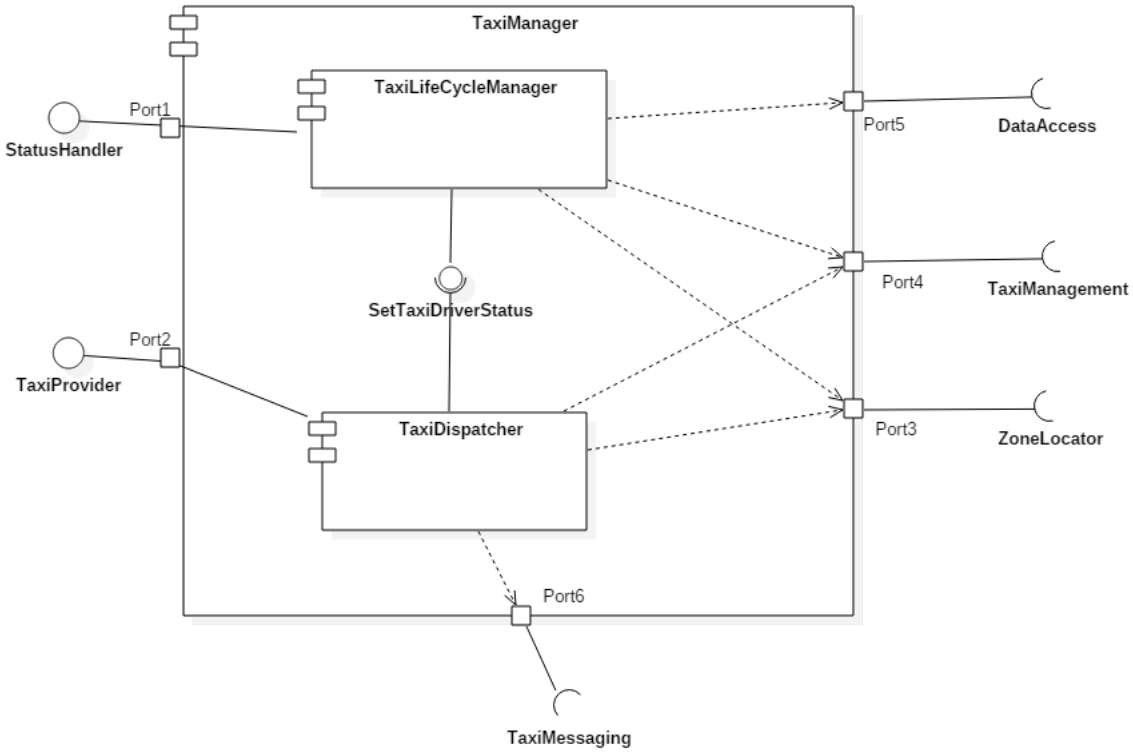


Figure 2.5: Taxi Manager internal structure

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
StatusHandler	The Taxi driver's mobile application	Allows the taxi driver to set himself as 'Available' and 'Not Available'. Plus, it is used from the application to communicate the current location
TaxiProvider	Ride Manager component	Given a location, it provides a taxi (if the correspondent queue is not empty and one taxi accepts)

Table 2.10: Taxi manager: provided interfaces

Required interfaces

Required Interface	Description and usage
ZoneLocator	Retrieves the correspondent zone of a location
DataAccess	Save the information of the taxi driver status
TaxiManagement	It communicates to the data structure containing the taxi drivers, whether to add, remove or retrieve them.
TaxiMessaging	It sends requests to taxi drivers.

Table 2.12: Taxi manager: required interfaces

2.3.4 Queue Manager

Internal Components

The Queue Manager is responsible for the management of available taxi drivers, to store and retrieve them when asked and apply the policy decided. Internally it is composed of the following components each with a precise job:

- *ManagementLogic*: it is the component which apply the decided policy over available taxi drivers. It exposes general methods such as add remove and get in order to hide the internal management. Before any operation it retrieves the corresponding queue from the QueuesContainer and then it makes the necessary operation on the queue.
- *QueueContainer*: as the name suggests, it is the container of the queues. It keeps then in a map with the zone as the key. It provide the ManagementLogic the methods for creating a queue or retrieving it.

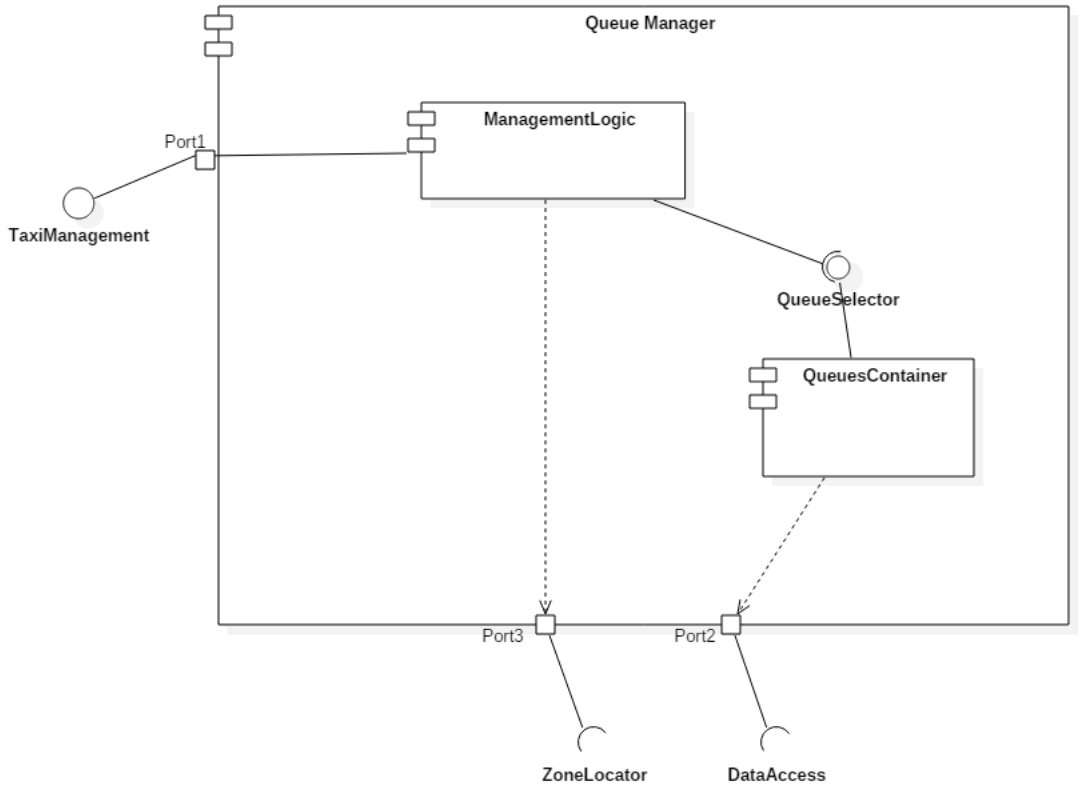


Figure 2.6: Queue Manager internal structure

Provided interfaces

Provided Interface	**Dedicated user**	**Description**
TaxiManagement	Taxi Manager component	Provides the methods for adding, removing and retrieving taxi drivers to the data structure (queue)

Table 2.14: Queue Manager: provided interfaces

Required interfaces

Required Interface	Description and usage
Zone Locator	Gets the list of the zones at the start up of the system in order to create the needed queues
Data Access	Maintains the queues in the database

Table 2.16: Queue Manager: required interfaces

2.3.5 Geographic Engine

The Geographic Engine is responsible for calculating the approximative waiting time for a passenger, and for computing the corresponding zone of a location. Internally it is composed of the following components each with a precise job:

- *WaitingTimeCalculator*: when it receives a call, it forwards the source and the destination to a Google Maps Web Service which computes the approximative traveling time.
- *ZoneCalculator*: it is responsible for storing the geographic definition of the zones and compute if a location is valid, and if it is in which zone is located. To do that it needs the Google Maps Web Service for the activity of geocoding, which means to get geographic coordinates (latitude, longitude) from an address.

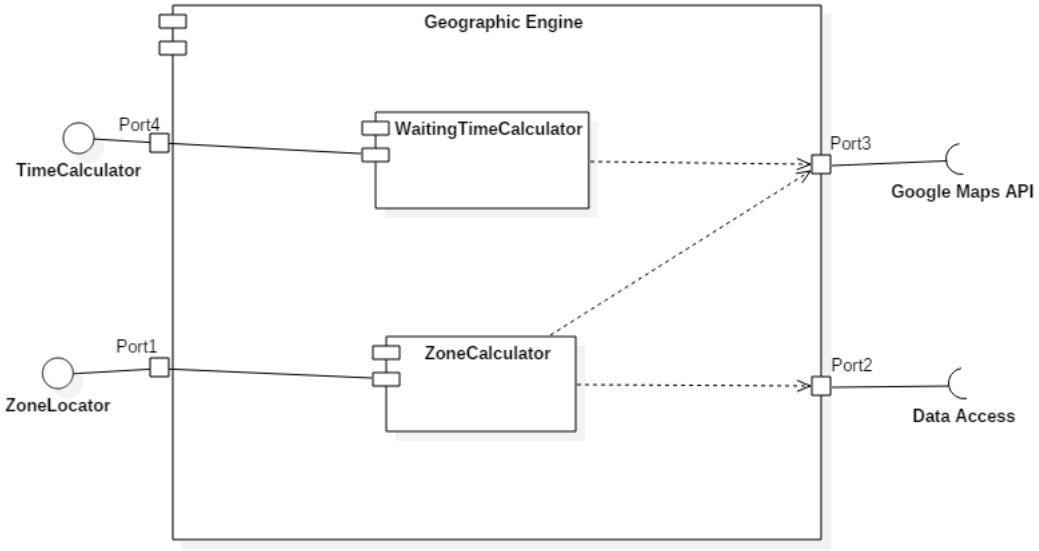


Figure 2.7: Geographic Engine internal structure

Provided interfaces

Provided Interface	Dedicated user	Description
TimeCalculator	Ride Manager component	Given a source location and a destination, it calculates an approximative waiting time
ZoneLocator	Taxi Manager, Queue Manager and Ride Manager components	It calculates the corresponding zone of a given location if it is valid (being valid means to be inside the city)

Table 2.18: Geographic Engine: provided interfaces

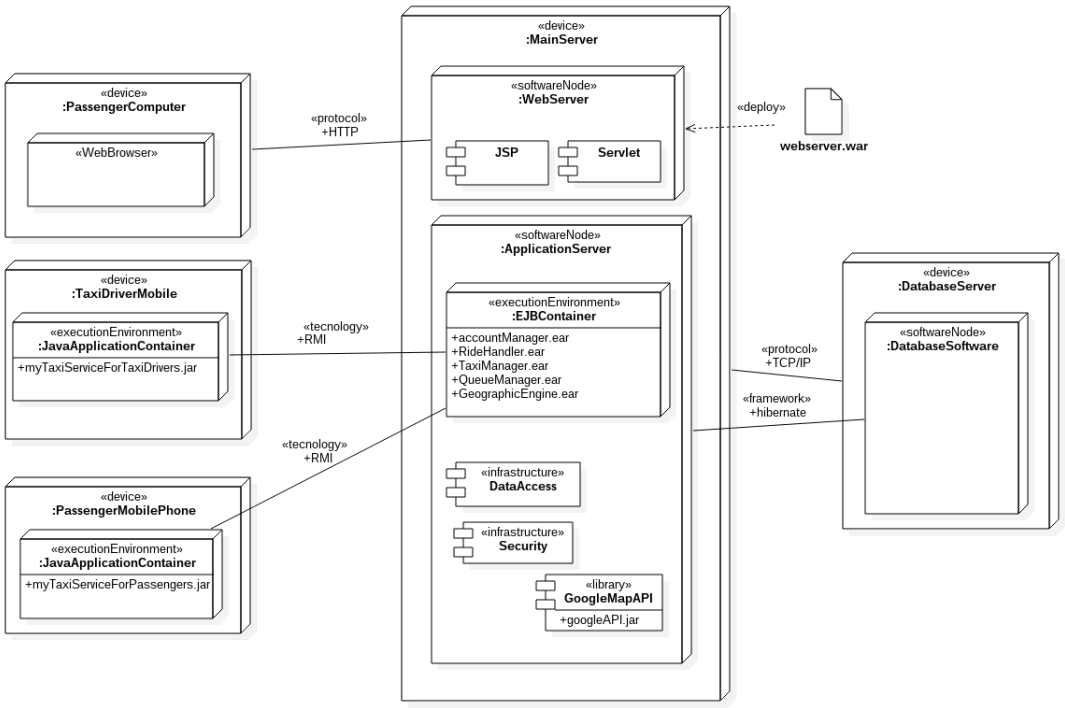
Required interfaces

Required Interface	Description and usage
Google Maps API	Used for the activity of geocoding and for calculating the approximative travel time
DataAccess	Retrieve the information about taxi zones

Table 2.20: Geographic Engine: required interfaces

2.4 Deployment view

In this section we show how our components are really deployed on hardware devices.



The UML diagram is self-explicative. We organize our deployed files like this:

- *Client side* we show the devices that can interact with our system: a mobile application and a browser for the passenger, and a mobile application for the taxi driver.
  - The passenger and taxi driver client applications interact with the system through RMI calls
  - The browser interact with the Web Server through HTTP protocol
- *Server side* both the web server and application server are deployed possibly in a cloud service, this will ensure the system to be reliable 24/24 7/7. The Application Server will keep all the components in a EJB pool so that it can handle the increased work load during specific time of the day, by deploying multiple instances of the stateless beans. It will also use the already specified infrastructural libraries for data access, security and the Google Maps API.
- The data are stored in multiple external *databases* accessed by the Main Server through the Hibernate framework. There will be more than one instance of the database in order to make the whole system more reliable. Hibernate will abstract the fact that more than one database is been used, so the system is not aware of this.

2.5 Runtime view

The components of the system interact with each other, in order to carry out the various activity that the system must has to accomplish. The activity are explained from different perspective as necessary.

2.5.1 Passenger makes a Request

Components involved and their role:

- **Passenger (Application)**: the activity starts when the passenger, from the mobile application, submits the request through the relative form. When the button is pressed, the client calls the method makeRequest of the Ride Handler component passing as parameters all the data of the form. He then expects as response either an affirmative message with the taxi driver info, or a negative message because there are no available taxi for the location provided.
- **Ride Handler**: once the Ride Handler as been called by the passenger, it firstly check whether the provided location is valid (using the Geographic Engine method) then he calls the Taxi Manager method provideTaxi, expecting a response with either a taxi or an negative message because there are no available taxi for the location provided.
- **Geographic Engine**: it checks if the location provided is valid or not.
- **Taxi Manager**: it provides a taxi for the specified location, if any available. For further details see 2.5.3.

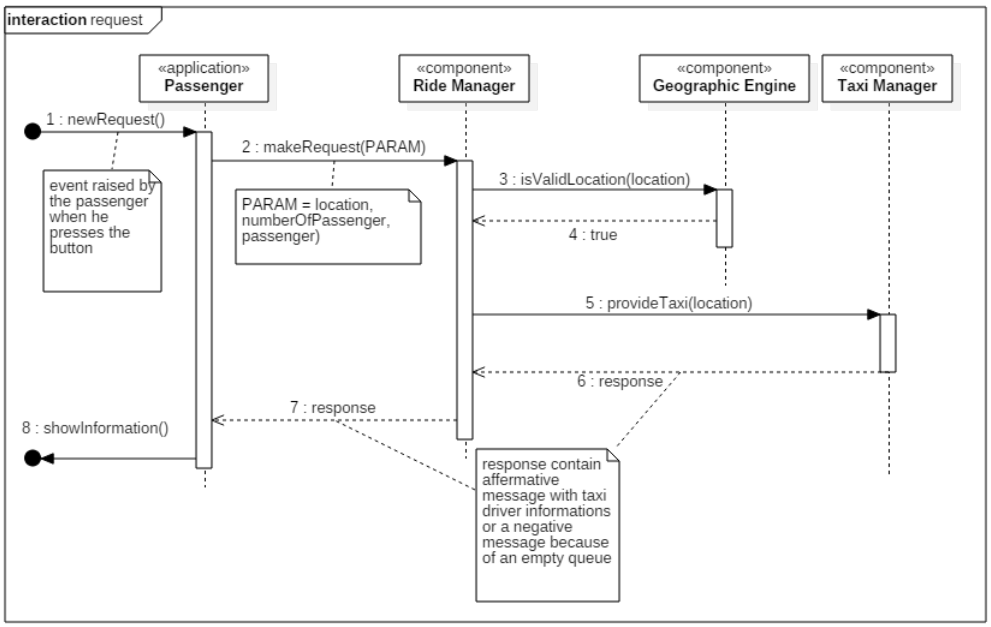


Figure 2.8: Sequence Diagram of the request

2.5.2 Passenger makes a Reservation

Components involved and their role:

- **Passenger (Application)**: the activity starts when the passenger, from the mobile application, submits the reservation through the relative form. When the button is pressed, the client calls the method makeReservation of the Ride Handler component passing as parameters all the data of the form. He expects as a response the outcome of the submission. Unless the location provided is not valid, the response is affirmative. The Passenger application now keeps behaving as usual and gets called by the system less then 10 minutes before the meeting time with the information of the taxi driver.
- **Ride Handler**: once the Ride Handler as been called by the passenger, it firstly check whether the provided location is valid (using the Geographic Engine method) then sends to the passenger the outcome of the reservation. After that, before looking for a taxi driver who will serve the reservation, he waits until 10 minutes before the meeting time. Once he wakes up, he calls the Taxi Manager method provideTaxi, expecting a response with either a taxi or a negative message because there are no available taxi for the location provided. In case the response is negative, he keeps calling the provideTaxi method until he finds an available taxi driver.
- **Geographic Engine**: it checks if the location provided is valid or not.
- **Taxi Manager**: it provides a taxi for the specified location, if any available. For further details see 2.5.3.

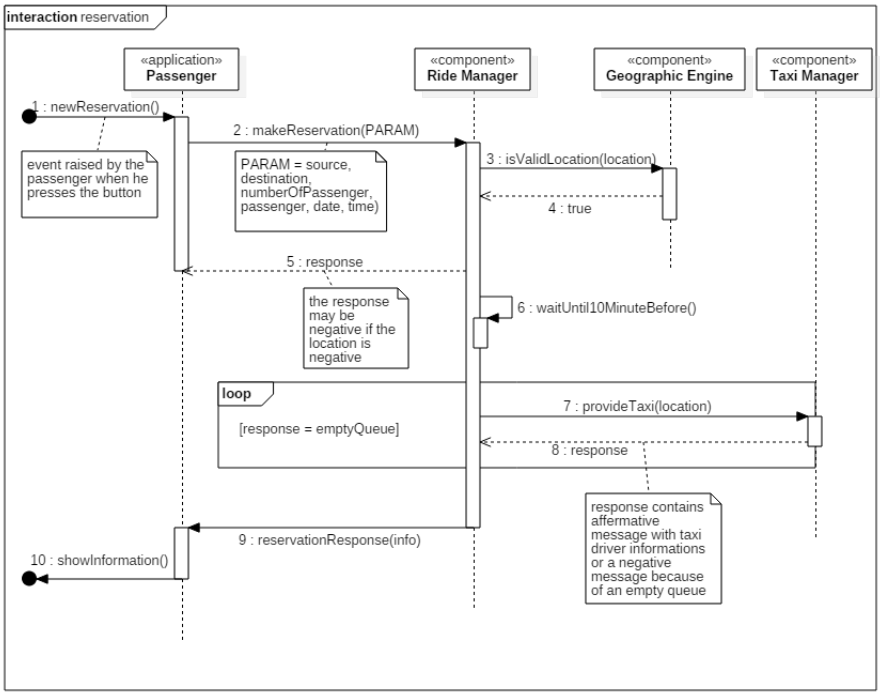


Figure 2.9: Sequence Diagram of the reservation

2.5.3 Provision of a taxi

Components involved and their role:

- **Taxi Manager**: it plays the most important role of the activity. First it gets the zone corresponding to the location from the Geographic Engine. Then he starts with a loop until either he finds a taxi driver who accepts the request, or the queue of the zone provided is empty. Every cycle it calls the method getTaxi of the Queue Manager to get a taxi driver. It then contacts the taxi driver by asking if he accepts to give a ride from a location. If the taxi driver accepts, the taxi manager returns to the caller the informations of the taxi driver. If the taxi driver does not accept, it first put them back to the queue (causing the taxi driver to be in the last position) and then it asks for a new taxi driver. Eventually, when the queue is empty, he returns a negative response.
- **Taxi Driver (Application)**: here the taxi driver is not a single application, but it refers to the current taxi driver being called by the taxi manager. When the application receive an offer of a request, it just shows a pop up to the user and let him decide whether to accept or not the request. Then it sends the response back to the taxi manager.
- **Geographic Engine**: it retrieves the corresponding zone of a location
- **Queue Manager**: after it is called by the taxi manager, he returns the first taxi driver in the corresponding queue of the zone provided. If the queue is empty it returns a negative response. Eventually it also put back in the queue a taxi driver who did not accept a request.

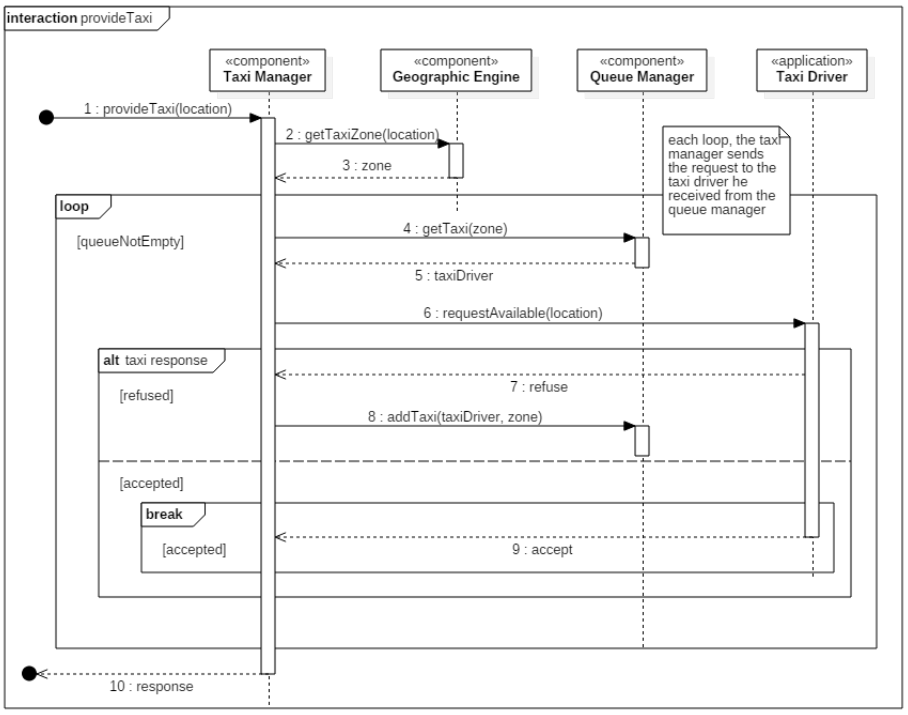


Figure 2.10: Sequence Diagram of the provision of a taxi

2.6 Component interfaces

Here we present the interfaces of our components: in particular which operations they offer, their meaning and their input/output parameters.

2.6.1 Account Manager

Passenger

- `Passenger login(String username, String password)`
  Given valid credentials, allows the passenger to receive a session and to login.
- `void logout(Passenger passenger)`
  Given a valid session, it deletes it.
- `boolean register(String username, String password)`
  It creates a new account for a passenger with the credentials provided.
- `boolean deleteAccount(String username, String password)`
  Given valid credentials, it deletes the account associated.

TaxiDriver

- `TaxiDriver login(String username, String password)`
  Given valid credentials, allows the taxi driver to receive a session and to login.
- `void logout(TaxiDriver taxiDriver)`
  Given a valid session, it deletes it.

2.6.2 Ride Manager

RideHandler

- `Response makeRequest(Location location, int numberOfPassengers, Passenger passenger)`
  It starts the activity of making a request. It returns either informations about the taxi driver who is coming, or an invitation to try later.
- `Response makeReservation(Location source, Location destination, int numberOfPassengers, Date date, Time time, Passenger passenger)`
  It starts the activity of making a reservation. If the time of the reservation is at least 2 hours later, it return a positive message.
- `List<Reservation> getReservations(Passenger passenger)`
  It returns all the reservations of a given passenger.
- `boolean deleteReservation(Passenger passenger, Reservation reservation)`
  It deletes a reservation of a passenger.

2.6.3 Taxi Manager

StatusHandler

- `boolean setAvailabilityStatus(TaxiDriver taxiDriver, AvailabilityStatus status)`
  It assigns a new availability status to a taxi driver.
- `void setLocation(TaxiDriver taxiDriver, Location location)`
  It updates the location of a taxi driver.

TaxiProvider

- `Response provideTaxi(Location location)`
  Given a valid location, if there is at least one available taxi driver, it returns it.

2.6.4 Geographic Engine

ZoneLocator

- `Zone getTaxiZone(Location location)`
  It computes the corresponding zone of a given location.
- `List<Zone> getTaxiZones()`
  It returns all the zones.
- `boolean isValidLocation(Location location)`
  Given a location, it checks whether it is valid or not.

TimeCalculator

- `Time getTimeEstimation(Location source, Location, destination)`
  Given valid source and valid location, it returns an approximative traveling time.

2.6.5 Queue Manager

TaxiManagement

- `void addTaxi(TaxiDriver taxi, Zone zone)`
  Given a taxi driver and a zone, it adds the taxi driver to the queue corresponding to the zone.
- `TaxiDriver getTaxi(Zone zone)`
  It returns the first available taxi from the queue corresponding of the given zone.
- `void removeTaxi(TaxiDriver taxi)`
  Given a taxi driver, it removes it from the queue he is in.
- `void changeZone(TaxiDriver taxi, Zone zone)`
  Given a taxi driver and a zone, it remove the taxi driver from the queue he is in, and add it in the queue corresponding to the zone.

2.7 Architectural Styles and Patterns

2.7.1 Client - Server

The application main architecture uses the *client-server* style.

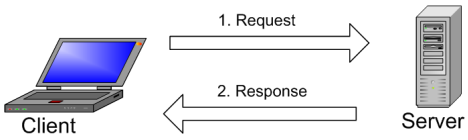


Figure 2.11: Interactions in a client-server architecture

The reasons for this choice are:

- The *centrality of the server* and the *sparsity of the clients*: we have different users that need some type of service provided by a particular organization
- The *absence of business logic client-side*: our user devices (web browser and mobile applications) must not need to know the back-end logic of the services they are invoking. This ensure scalability and the possibility to add new services or modify the already present with a minimum effort in the update of the applications client side.

The interaction of a client-server system has the client as initiator of the communication, which makes a *request* to the server. The server, received the request, elaborate a *response* using its internal business logic, and finally send it to the client.

We made the design choice to have *thin* clients, which leads to the following advantages:

- The client applications (and of course, the web pages) are easily modifiable
- The client applications are power consumption optimized
- Easy installation of the app and increased download speed of it (both mobile application and web pages).

2.7.2 The problem of the notification to the clients

A great design issue raised by the requirements of the application is *how the server can contact the clients* (both passengers and taxi drivers). A passenger is contacted by the server when a reservation of him is ready and it was assigned to it a taxi driver. A taxi driver is notified by the server when a request for him is available. We notice immediately that this interaction is basically asynchronous.

This model of interaction is in conflict with the concept of *client - server* style.
Anyway this conflict is only at the modeling level because we can *simulate* this interaction with a low level *polling* policy which uses the client-server paradigm with some optimization in order to minimize the number of requests carried out by the client to the server.

Anyway the high level architectural style for this kind of interaction is basically a variant of the *publish-subscribe* (event-based).

From the point of view of implementation there are lots of *messaging frameworks* that can be used: JMS maybe is the most natural choice if we would choose the JEE infrastructure.

These design choices have been chosen in order to *reduce* the number of request sent by the clients to the server, which are the main drawback of a client-server approach.

2.7.3 Three-tier-architecture

We decided to use a *three-tier-architecture* for our system.
This is simply a specialization of the client-server architecture in which we specify the different layers and components of our system.
A schematic representation of this responsibility distribution is at figure 2.12

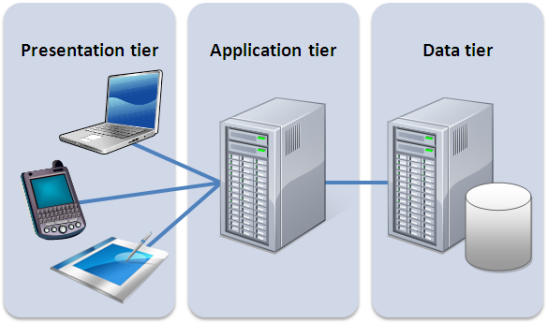


Figure 2.12: A representation of a three-tier-architecture

So we subdivide our system in three extremely separated layers (or *tiers*):

- A *presentation* layer for the graphic rendering of the data and events generated by the system, and from which the end user can interact with the system
- An *application* layer entitled to manage all the business logic of the system
- A *Data* layer responsible of the storage of informations to be used by the application and presentation layer

The three levels are completely independent and can be replaced easily. In particular, the presentation layer cannot communicate directly with the data tier, but it must forward its requests to the application one, which will use some data access framework to access the database.

The reasons of this architectural choice are:

- The possibility of use *well-defined interfaces* between the different layers. Each layer is dependent uniquely on the interfaces with the other elements of the system. This ensure scalability and, from the data layer side, the possibility of replication and clouding of the resources.
- This subdivision of roles makes the defining of the required *programmatic interface* easier. Each layer and each subcomponent will expose an interface which can be part of an API to be used by the future developers in order to improve the services and create new ones.

2.8 Other design decisions

Subdivision into components

From the previous sections is clear that all our architectural design is highly oriented to the subdivision of the entire system into submodules, having each of them a different role in the achieving of the requirements of our application. We can call this approach *component-oriented*. It is basically an application of the *divide and conquer* design principle.

Dependency inversion principle

We have taken care of always provide interfaces for the subcomponents applying so the *dependency inversion principle*. The various components of the system depend only on the interfaces of the others and never on the internal representation.

Deployment choices

As far as the deployment is concerned, we have already seen that our choice is to deploy both the business logic (application server) and the web management (web server) on the same machine.
We must say that our RASD did not specify any constraint about deployment and so we opted for the easiest and cheaper solution for our clients.
A future enhancement may be to delegate the data logic to a clouding infrastructure with a remote server. This possible choice will bring the following advantages:

- *Division of responsibility*: the data storage is delegated to a specialize provider which can manage it with advanced control systems. This feature brings to the following advantages.
- *Security*: to have the data stored far away from the Main Server enhance security
- *Easy scalability* of the database

Programmatic interface

For the satisfiability of the requirement to have a programmatic interface that can make the developers able to build new functionalities on the top of the already provided, we decided to expose all the component interfaces that we have described (figure 2.13).

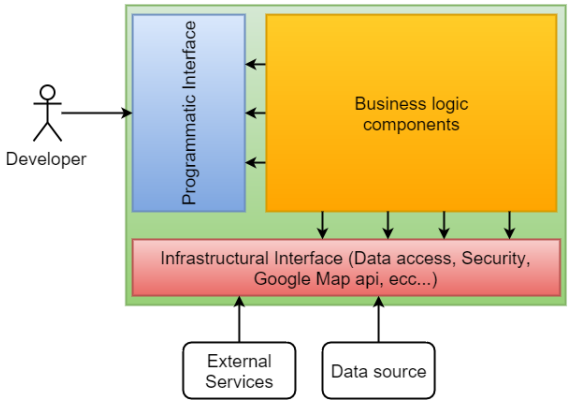


Figure 2.13: Schematic representation of the role of the programmatic interface in the system

Chapter 3

Algorithm Design

In this section we clarify some fundamental algorithm used by our system in order to achieve its objectives. We will deal basically with high level interactions between the components described in the previous sections.
All the algorithms are presented without any reference to a particular programming language but we are assuming an object-oriented paradigm for it.

Since all the architecture is designed for a client - server interaction an obvious choice of framework would be the *Java Enterprise Edition* framework and so the usage of the Java programming language.

3.1 Queue management

The following algorithm is in charge of the management of the different taxi queues. It will be used by the **ManagementLogic** unit, inside the **QueueManager** component.

Algorithm 1 Retrieval of a taxi from a queue

Require: *zone* : a correct taxi zone
Ensure: a provided taxi or an error message, if there is no taxi driver waiting in the queue

queues: map having as keys the taxi zones and as value the corresponding queue
queue := $queues.VALUE(zone)$ {Retrieval of the queue relative to the provided taxi zone}
$returnValue$: return value
if $queue.ISEMPTY()$ **then**
  $returnValue := error$
else
  $taxi := queue.GETTAXI()$ {Takes the first taxi in the queue}
  $returnValue := taxi$
end if
return $returnValue$

From the pseudo-code we understand the following:

- The **QueueContainer** stores the taxi queues in a map having has key the taxi zones

- Each queue is a particular data structure organized with a FIFO policy

The following algorithm describe the adding of a taxi in a queue, action performed by the **ManagementLogic** unit using the interface provided by the **QueueContainer**

Algorithm 2 Adding of a taxi to a queue

Require: Inputs:

- *taxi* : a taxi driver that must be inserted in a queue
- *zone* : the zone in which the taxi driver wants to operate

Ensure: the adding of the taxi driver to the queue corresponding to the zone provided

queues: map having as keys the taxi zones and as value the corresponding queue
$queue := queues.VALUE(zone)$ {Retrieval of the queue relative to the provided taxi zone}
$result = queue.ADDTAXI(taxi)$ {Adding the taxi driver at the bottom of the queue}
return

Lets analyze in more detail the data structure *queue* which has the role of storing the taxi drivers during their waiting of calls: it has a FIFO (*first-in-first-out*) policy for the management of its elements as it is visible in figure 3.1

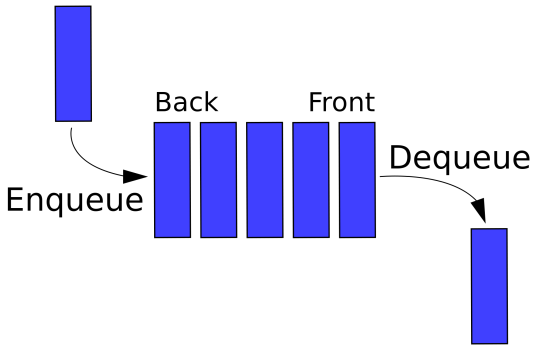


Figure 3.1: FIFO policy

In the Java programming language we could implement a class `TaxiQueue` which store an element of the class `LinkedList` which extends the interface `Queue`, which is already provided by the framework. Below you can see an example of usage of this class:

Listing 3.1: Example of usage of the Queue interface in Java

```
public class TaxiQueue {
        //Usage of the LinkedList implementation of the interface Queue
        //FIFO policy
        Queue<Taxi> queue = new LinkedList<Taxi>();

        public void addTaxi(Taxi taxi)
        {
```

```
                this.queue.add(taxi);
        }
        //....
}
```

3.2 Reservation Handler

The nature of the problem of managing the reservation waiting time before the submission of the request to a taxi driver requires a *multi-threaded* solution.
We design the following algorithm to be the conjunction of behaviors of 3 different threads:

- *Reservation Receiver*
- *Waiter*
- *Reservation Forwarder*

3.2.1 Reservation Receiver thread

Algorithm 3 *Reservation Receiver*: Adding of a reservation to the data structure

Require: Input:

- *reservation* (data structure containing: passenger data, meeting time and date, source and destination, number of passenger)

Already present data structure:

- *reservationList*: ordered list containing all the reservations waiting to be forwarded to a taxi driver. The order is done by meeting time.

Ensure: The *reservation* is added in the correct order in the *reservationList* and the timer of the *Waiter* is set correctly
$meetingTime = reservation.meetingTime$
$reservationList.PUT(reservation)$ {the reservation is put in order by meeting time}
$index = reservationList.GETINDEX(reservation)$
if *index* is the first **then**
  Sends a signal to the *Waiter* in order to stop is current timer (signal)
  $waitingTime = meetingTime - CURRENT\_TIME$
  Communicate to the *Waiter* thread the $waitingTime$
end if

3.2.2 Waiter thread

Algorithm 4 *Waiter*: setting of the timer and timer termination

Require: Already present data structure:

- *timer*: is a counter that can be set as ON/OFF and which can raise an signal when the set time expires.

Ensure: The correct setting of the timer

timer.count = 0 {First initialization of the timer}
while true **do**
  if a signal is received **then**
    waitingTime = the parameter communicated by *Reservation Receiver*
    $timer.SETTIME(waitingTime)$
    $timer.START()$
  end if
  if *timer* expires **then**
    Send a communication to the *Reservation Forwarder*
    $timer.STOP()$
  end if
end while

3.2.3 Reservation Forwarder thread

Algorithm 5 *Reservation Forwarder*: Forwarding of a reservation to the RequestQueueManager

Require: Already present data structure:

- *reservationList*: ordered list containing all the reservations waiting to be forwarded to a taxi driver. The order is done by meeting time.

while true **do**
  Wait for a signal by the *Waiter* thread
  A signal is received
  $first$ = the first index of $reservationList$
  $firstReservation = reservationList.GET(first)$ {Next reservation is got from the list}
  $reservationList.REMOVE(firstReservation)$ {Next reservation is removed from it}
  $nextReservation = reservationList.GET(first)$
  $waitingTime = nextReservation.meetingTime - CURRENT\_TIME$
  Communicate to the *Waiter* thread the $waitingTime$
  Forwards the $firstReservation$ to the RequestQueueManager
end while

Here we presented a pseudo-code of the behavior of this three threads. Each language, in particular Java, has its way to implement threads and waiting conditions. In this document we don't care about implementation.

Chapter 4

User interface design

We have already dealt with the interface design in the Requirements Analysis and Specification Document, where we have shown some mock-ups of the screens of our applications.
In this section we will refine the user interface basically from the point of view of *interaction* with the end-user (mapping between a sequence of actions and a screen flow).

We will use the class diagram for the specification of screen flows. This approach need some clarification about the used notation:

Symbol	**Meaning**
Directed arrow with *function()* over it	It defines a transition from a particular user interface element to another. This transition is triggered by the invoking of a particular method *function()* in the source interface element.
Class symbol	It determines a specific user interface element from the set {*Screen*,*Form*,*Element*,*Pop-up*} or a specific event (stereotype *event*). In both cases the type of element is specified in the stereotype of the class symbol.
Composition symbol	Means that a specific user interface element is contained in another. Typically is used when a form or a list of elements is contained in a particular screen.
Directed arrow with the stereotype *event* over it and a *text*	Means that it is not a user induced function to trigger the transition, but an event from the software, in particular an event sent by the server to the client. The *text* describe the event.

Table 4.2: Notation for the user interface flow diagram

4.1 Passenger App

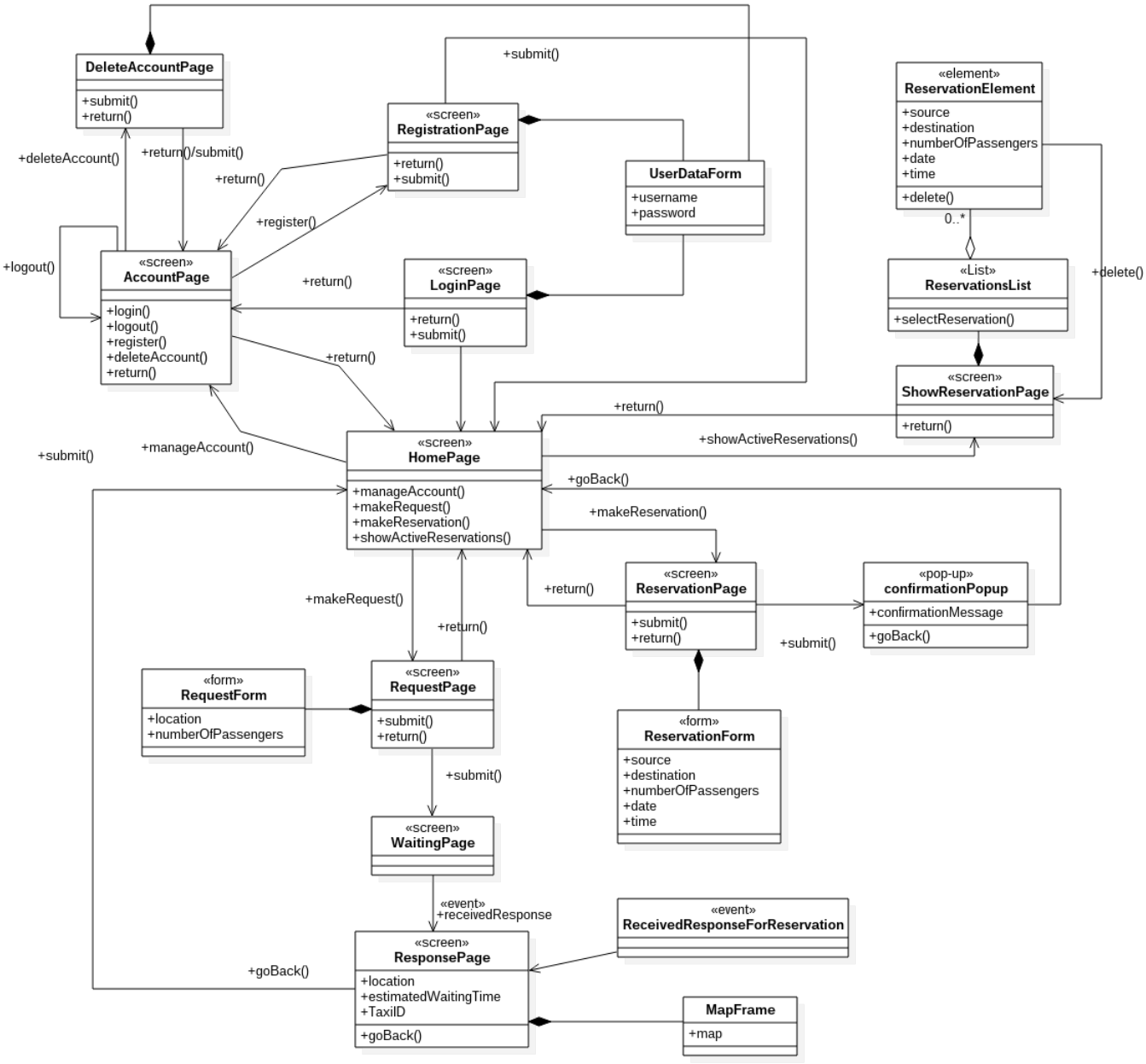


Figure 4.1: Screen flow of the Passenger mobile application

4.2 Taxi Driver App

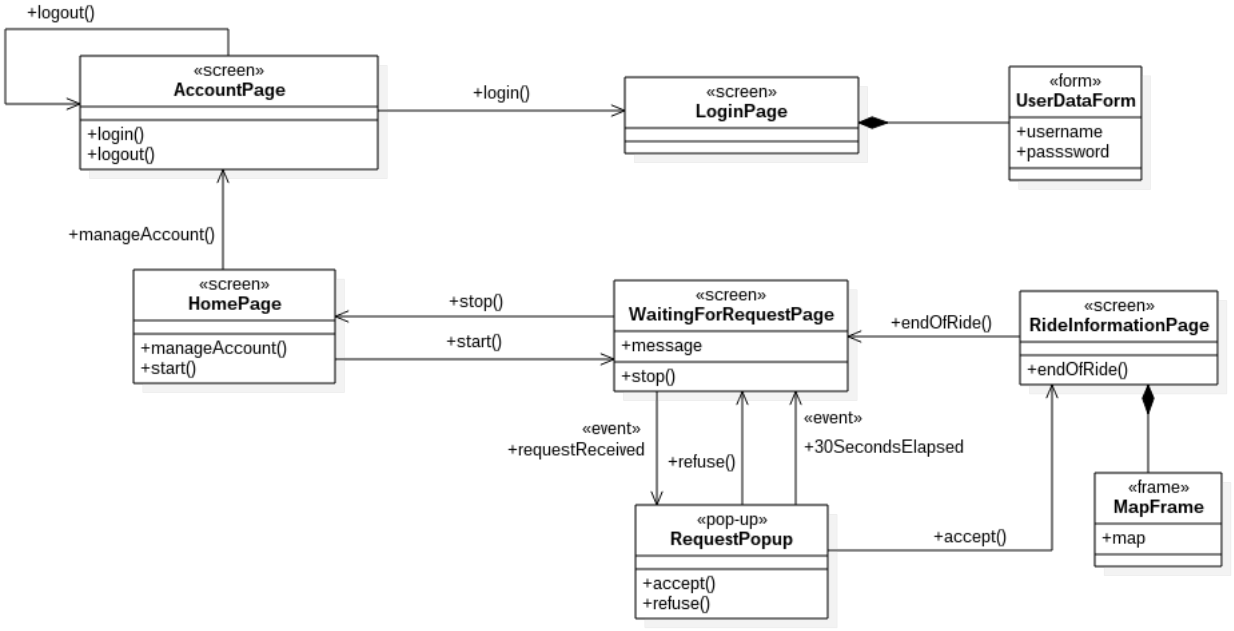


Figure 4.2: Screen flow of the taxi driver mobile application

Chapter 5

Requirements Traceability

Requirement		Design Solution	
R.P.1	Passengers can't register twice with same username	CMP: Account Manager	The AccountManager manages the login and registration phase, avoiding this to happen
R.P.2	Username provided by passenger in registration phase must not be empty	CMP: Account Manager	It makes this check
R.P.3	The password provided in the registration phase must not be empty	CMP: Account Manager	It makes this check
R.P.4	The system must notify and abort the registration procedure in the previous 3 cases	CMP: Account Manager	The $register(username, password)$ method provided by the Account Manager will return an error in any of the cases and the passenger GUI will show an appropriate pop-up.
R.P.5	The system must provide for the Passenger with a way to abort the registration procedure	UX: Passenger App	Will be provided, in the user interface, a specific button for stopping the registration procedure
R.P.6	If a passenger makes a request and the corresponding taxi queue is not empty, then the system must sooner or later respond positively to the Passenger	SD: Passenger request Provision of a taxi	The TaxiManager will return a taxi to who wants it if the corresponding queue is not empty, and an error message if empty

R.P.7	A passenger request must be refused if and only if there are no taxi driver available in the corresponding taxi queue or the number of passengers of the ride is greater than 3	CMP: RideManager	The RequestHandler inside the RideManager will use its parser and will return an error in case the specified number of passengers is greater than 3. For what the emptiness of the queue refer to R.P.6
R.P.8	A reservation must be refused if: • Origin and destination are the same location • The number of passengers of the ride is greater than 3 • time(meeting time) - time(reservation) $<$ 2 hours	CMP: RideManager SD: Reservation	The ReservationHandler inside the RideManager will use its parser and will return an error in case the specified number of passengers is greater than 3 or the source and destination are the same. It will cover the possibility to retry, in case the corresponding queue is empty, to forward the request. This is done until a taxi is found
R.P.9	When the system looks for a taxi driver for serving a reservation (10 minutes before the meeting time), if the corresponding taxi queue is empty, it will wait until there is at least one taxi driver in the queue.	CMP: RideManager SD: Reservation	Refer to R.P.8
R.T.1	A Taxi Driver, when he sets himself as available, must be put at the bottom of the taxi queue relative to his taxi zone.	CMP: QueueManager TaxiManager ALG: Adding of a taxi	The queues inside QueueContainer subcomponent of the QueueManager are managed as a FIFO structure
R.T.2	A Taxi Driver, when he is available, must be in one and exactly one taxi queue.	CMP: QueueManager TaxiManager	The TaxiManager checks the availability status of the taxi driver and puts him in exactly one queue, if he is available
R.T.3	A Taxi Driver, when he is not available, must not be in any taxi queue.	CMP: QueueManager TaxiManager	Refer to R.T.2
R.T.4	At each position of the Taxi Driver retrieved from the GPS data must correspond exactly one and only one taxi zone	CMP: GeographicEngine	This component has the functionality to associate to each location exactly one taxi zone

R.T.5	A taxi queue must have a number n of taxi driver waiting in the range $n \in [0, +\infty[$	CMP: QueueManager TaxiManager ALG: Adding of a taxi	The structure of the queue object is a FIFO dynamic structure
R.T.6	A Taxi driver must be always in one of this three states, which are mutually exclusive: available, busy, not available	CMP: TaxiManager	The TaxiManager manages correctly the states of each taxi driver
R.T.7	The system must put the taxi driver at the bottom of the queue if: he refuses a request, he does not respond to a request within a certain time from the reception of it (10 seconds)	CMP: TaxiManager SD: Provision of a taxi	The TaxiManager component manages the message sending and receiving form the taxi driver.
R.T.8	A taxi driver can receive requests if and only if he is at the top of a taxi queue	CMP: QueueManager ALG: Retrieving of a taxi	The FIFO structure of the queue object and the usage of this structure by the QueueManager makes this happen
R.T.9	A taxi driver waiting (available) in a taxi queue must receive requests only from passengers which specified a meeting point inside the correspondent taxi zone	CMP: GeographicEngine TaxiManager QueueManager ALG: Retrieving of a taxi	Refer to R.T.4 for the GeographicEngine. The TaxiManager will use the GeographicEngine in order to retrieve the correct taxi zone. The Queue manager will make the link between the taxi zone and the taxi queue
R.T.10	When a Taxi driver accepts a request from a Passenger, he must be removed from the corresponding taxi queue and set as busy.	CMP: TaxiManager QueueManager	The TaxiManager, received the positive message from the taxi driver, will set his state as busy (it was already removed from the queue)

Table 5.1: Requirements traceability table

As far as concern the requirement of the programmatic interface, it was already explained at the relative section.

Chapter 6

Appendix

6.1 Working hours

- Luca Nanni: 25 hours
- Giacomo Servadei: 25 hours

References

- ISO/IEC/IEEE 42010:2011(E), Systems and software engineering - Architecture Description
- IEEE Std 1016-2009 (Revision of IEEE Std 1016-1998) IEEE Standard for Information Technology - Systems Design - Software Design Descriptions