POLITECNICO
MILANO 1863

Design Document of Software Engineering 2 Project

Andrea Gussoni Federico Amedeo Izzo Niccolò Izzo

February 15, 2016

Contents

Chapter 1

Introduction

1.1 Purpose

This document has the objective of defining in a clear and unambiguous way the structural design of myTaxiService. The Service we are considering is inherently modular and heterogeneous, in fact its structural design will be a blend of Hardware, Software and UX elements. Everyone of these elements interact with each other by a variety of communication technologies. This document will have to define the complete internal structure of the system, highlighting the interconnection between the various subsystems and describing with detail the system's behavior in some various scenarios. In particular, the document will describe in detail the Architectural Design of the System. For Architectural Design we consider the details of the various components of the system from three different views:

- The components view, in which we focus on the various components interaction.
- The deployment view, in which all the deployment details are explained thoroughly.
- The runtime view, in which the system's composite structure is analyzed in some example scenarios.

These three views will give a complete description of the Architectural Design, considering it by various perspectives, and giving a three-dimensional and detailed view on the future system's inner structure.
The Design Document will also focus on the Software structure from the algorithmic point of view, describing the most crucial algorithms that will be embedded in the system's structure. Those algorithms will be explained

in abstract languages such as pseudo-code or flowcharts, to accomplish two goals:

- Let non technical people understand the structure of the algorithms without distracting them with implementation details.
- Give the Developers the possibility of realizing the algorithms with the chosen programming language.

Another section of the Design Document will explain in more detail the characteristics of the User Interface design. This, in addition to the mock-ups in the Requirement Analysis and Specification Document will help the Mobile Application Designers and the Web Designers in realizing a UX that truly fits the various actors' needs. This document will be deeply connected with the RASD, and will specify how each of the requirements analyzed in that document map into the various design choices elaborated inside this Design Document.

1.2 Scope

This Design Document describes the development of myTaxiService. As stated before myTaxiService is a service which purpose is to improve the actual taxi infrastructure of the city in order to simplify and make more reliable the procedure of a taxi call. The main goals of the systems from the customer point of view are:

- Have access to a taxi more quickly.
- Reserve in advance a taxi ride with fixed origin and destination.
- Share the taxi fee with others passengers going in the same direction.

In this Design Document we will see how this will be possible. In particular:

- The section that explains how the access to the taxi infrastructure will be more efficient and simple is the section that covers the design of the user interface.
- The section that shows how the reservation and the fee-sharing will be possible is the Algorithm Design section.

The benefits that the taxi drivers will have from the use of myTaxiService are:

- Get an even distribution of the customers' service demand.
- Automatic route planning proposal in case of shared taxi ride.

We will see in the Algorithm Design section that all the algorithms necessary to the operation of the system will take in large account these two aspects, that we think essentials for the success of whole the platform. In fact we think that the presence of the taxi drivers on the platform is the key aspect for a large adoption of this new platform. Some realities that are already on the market show that that the customers will be attracted by this kind of platform, but because they offer a service that is parallel to the classic taxi service there are often disputes about the legality of them. myTaxiService instead aims to involve the taxi drivers, and we think that having the widest number of them using the platform will be the key aspect that can lead myTaxiService to success.

1.3 Definitions, Acronyms and Abbreviations

1.3.1 Definitions

- Customers: those who will require the ride through the web application or the mobile application.
- Taxi drivers: registered users of the mobile application. They will upload their position and their availability to take rides to the system.
- Standard ride: action that begin with the customer's ride request and end with the customer's payment at the end of the ride.
- Reserved ride: a ride that has been reserved at least two hours before the starting time. It begins from the reservations and ends with the customer's arrival at his destination.
- Shared ride: a different type of ride in which the customer give his availability to share the ride. A ride is considered shared when at least two customers are traveling in the same taxi cab.
- Smart-phone: a mobile device capable of connecting to the Internet and making and receiving calls and SMS.
- Geo-localization: the act of obtaining a user's geographic coordinates, eventually uploading them to an on-line service.
- Application: mobile or web app running on the user's device.
- System: refers to the part of the application logic that runs on the remote server.
- Taxi Zone: is an area of approximately $2km^2$ for which the taxi-queue is unique.
- Developers: all the people involved in the development of the Service.
- Stakeholders: all the people that may be affected by the Service activities.
- Scalable: used in describing the capability of adapting the resource usage in accordance to an increase/decrease of number of incoming requests.

1.3.2 Acronyms

- DD: Design Document.
- RASD: Requirement Analysis and Specification Document.
- API: Application Programming Interface.
- UI: User Interface.
- OS: Operating System.
- UX: User eXperience.
- SOA: Service Oriented Architecture.
- IaaS: Infrastructure as a Service.
- SaaS: Software as a Service.
- PaaS: Platform as a Service
- MVC: Model-View- Controller.
- OO: Object Oriented.
- Java EE: Java Enterprise Edition.

1.3.3 Abbreviations

- Web app: Web-based application.

1.4 Reference Documents

- Specification Document: *Structure of the Design Document.*

1.5 Document Structure

The structure of the following document will follow the guidelines described in the specification document. So this will be the basic structure:

- Section 1: *Introduction*
  Gives a general description of the document and underlines the main aspects necessary to understand the document.

- Section 2: *Architectural Design*
  In this section will be described all the main components of the architecture of the final system. All the requirements present in the RASD will be translated in components or functionalities of the components, that eventually might interact one with each other. All the choices will find a justification and all the patterns and styles followed in the design procedure.

- Section 3: *Algorithm Design*
  In this section we will present all the main and most important algorithms on which the functionalities of the System will be based. We will use natural language descriptions and pseudo-code when it will be necessary.

- Section 4: *User Interface Design*
  Due to the fact that we already covered this part in the RASD, we will refer to it and when necessary we will expand and better specify some details in accordance to any new needs.

- Section 5: *Requirements Traceability*
  This section will be devoted to explain how the requirements expressed in the RASD have been put in place in the System. We will try for all the major requirements to highlight how the requirements map in the components of the designed System.

- Section 6: *References*
  In this section we will put all the required references to previous documents.

Chapter 2

Architectural Design

2.1 Overview

In this paragraph we will provide a complete overview of all the system components, beginning from the logical level and discussing later about the physical level.
In the high level components section 2.2 we will describe the system from a high level point of view. This description will be extended in the components view section 2.3. In the deployment view section 2.4 we will describe the deployment of the logical components on the physical systems, showing the tier-based architecture. The deployment view section 2.5 will be devoted to describe dynamically the behavior of the system, showing how it responds to various stimuli and showing some examples of the execution of its operations. In the components interfaces section 2.6 we will describe the anatomy of the interfaces between the various components of the systems. Some attention will be also put in a description of the APIs that will give access to our system. In the architectural syles section 2.7we will discuss about the design styles we selected for building the architecture, also showing what patterns we thought to be appropriate to follow. Finally in the other design decisions section 2.8 we will explain some minor design choices that we took in the architectural design of the system.

2.2 High level components and their interaction

Here is a list of the main components of the system:

- Database: The layer's main function will be to store data to be used by other components of the system. It will be responsible for the correct data storage and retrieval. In this layer no type of application logic will be implemented.

- Back-end Server: In this layer we will have the business logic of the system, including all the algorithms. This layer is of SOA type (Service Oriented Architecture).
- Web Server: This is the layer that will serve as an interface with the external world for the interaction with the customers through web browsers. This layer will not contain any business logic.
- Mobile App Interface: This is the layer that represents the interface of the system on the mobile clients, for both taxi drivers and customers.
- App Proxy: This is the proxy layer between for the communications between the mobile apps and the application server.

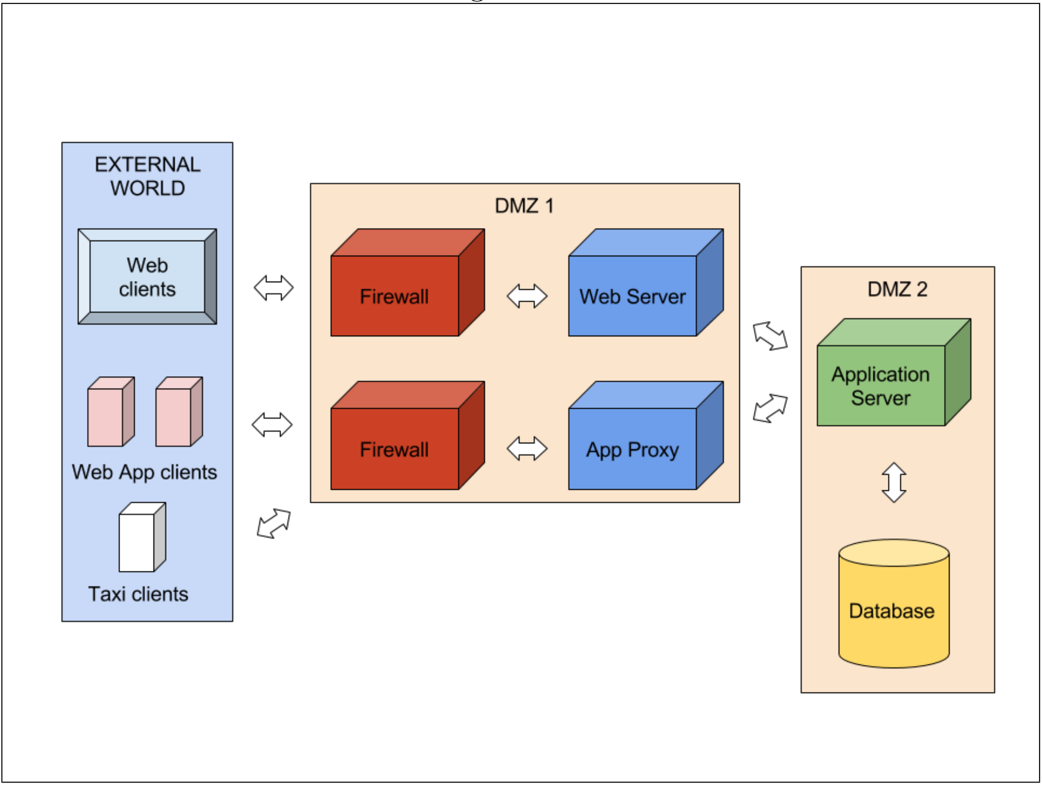
We made the design choice to structure the main components in a four-tiered architecture. This choice has the main goal to deploy web servers and application servers (the ones that will implement the business logic) on different tiers. This logical and physical separation will bring some advantages regarding the following aspects:

- Security: in this architecture the application server will never be exposed directly to the external world. In this way we can concentrate the main security checks and measures in the web server's tier, this tier is the most suitable location where to do them. In this way we can also mitigate the effects of an hypothetical DoS (Denial of Service) attack, deploying various web servers that communicate with others application servers. So that the core of the system is very difficult to flood and can always be responsive.
- Scalability: with this design-choice we avoid big restrictions to the future expansions of the system. Indeed we will be able to better evaluate the number of needed physical machines also during the beta test of the system. Also in this way we can benefit from the scalability properties of cloud systems (This part will be better detailed in the Deployment section 2.4).
- Performance: with this design choice we can make all the interactions between the components asynchronously (except for the interactions between the Back-end Server and the Database, that must be synchronous in order to guarantee the correct behavior of the system. We

will evaluate the option to make use of a distributed cache system in the Deployment section 2.4).

This is a diagram that represents the high level view of the architecture:

Figure 2.1:



2.3 Component view

The main components that will compose the system are the following:

- Customer Client: this component represents the mobile clients that will be used by the customers. It represents both the web clients and the Mobile App, at this level indeed there is no need to differentiate them. This is a presentation layer, and the interactions between the customers and the systems all traverse this layer.
- Taxi Client: also this is a presentation layer and as seen for the Customer Client it is the point of contact between a certain type of users (the taxi drivers) and the system.

- Account Manager: this is the component that provides support to all the registration, login and authentication functions needed in order to obtain access to the services provided by the system. This component is interfaced with the database, in particular with the sub-component responsible of the storage of the account informations.

- Queue Manager: this entity is responsible of the correct management of the Taxi Queues. As further specified in the algorithm chapter 3, the role of this entity is to adapt the queue distribution on the basis of various factors, such as the number of available taxis at a certain point of the day. It also provides interfaces to the entities that have the task of organizing the rides. This is due to the fact that given the dynamical nature of the queues, we need to have an entity that offers an abstraction of the queues. This is the reason behind the absence of the queues in the component diagram. This component is interfaced with the component of the database that store the information on the on-going rides.

- Taxi Dispatcher: this is one of the main components of the business logic of the application. As we will see it will live in the Back-end Server. Its role is to interact with the Queue Manager component in order to manage the correct dispatching of the taxi needed to serve the rides.

- Ride Dispatcher: this component is the one responsible to arrange the ride. It receives the position from the customer, interacts with the Queue Manager and also interacts with the customer's client in order to provide all the necessary informations. In case of a reserved ride it also manage the reservation, and at the correct time it will initialize the ride. In case of a shared ride the component activates the Shared Ride Manager component. This component is also responsible for the calculation of the quota of each passenger in the case of a shared ride.

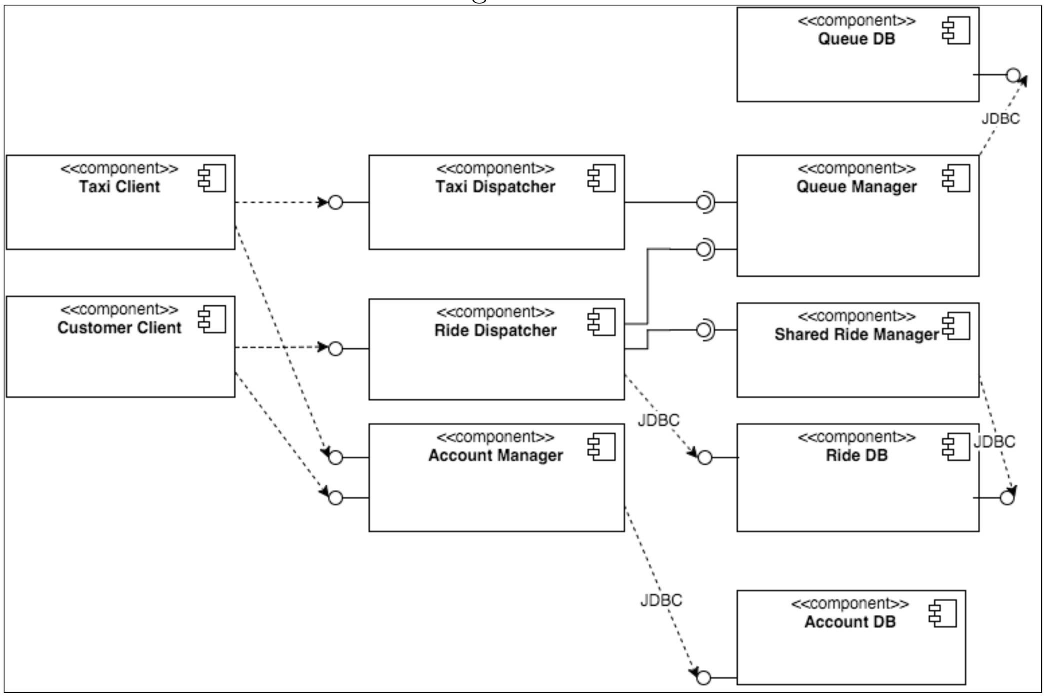
- Shared Ride Manager: this component is responsible of the correct instantiation of a shared ride. It doesn't interact directly with the clients, but it is activated only by the ride dispatcher in case of need. In the Algorithm Design (chapter 3) we will see in detail the behavior of this component.

- Database: the database is the component that manages all the information that needs to be stored for the correct operation of the whole system. In the component view diagram we split this entity in 3 sub-entity, each of them storing a different kind of information. As we will see in the deployment view section this sub-entities will be grouped in a single macro entity.

2.3.1 Component view diagram

In this subsection we provide a diagram that represents the main components present in the system. This diagram is called **Component View Diagram**:

Figure 2.2:



<<component>> Queue DB
JDBC
<<component>> Taxi Client
<<component>> Taxi Dispatcher
<<component>> Queue Manager
<<component>> Customer Client
<<component>> Ride Dispatcher
<<component>> Shared Ride Manager
JDBC
JDBC
<<component>> Account Manager
<<component>> Ride DB
JDBC
<<component>> Account DB

2.3.2 Implementation of the components

The business logic will be implemented using *Enterprise JavaBeans (EJB)* provided by the Java-EE architecture. We will make use of both stateless and stateful beans, even if we will prefer the use of stateless beans in order to avoid a too much resource-demand in the Back-end. The store of the state of users, rides etc. will be done using the Database and a suitable architecture.

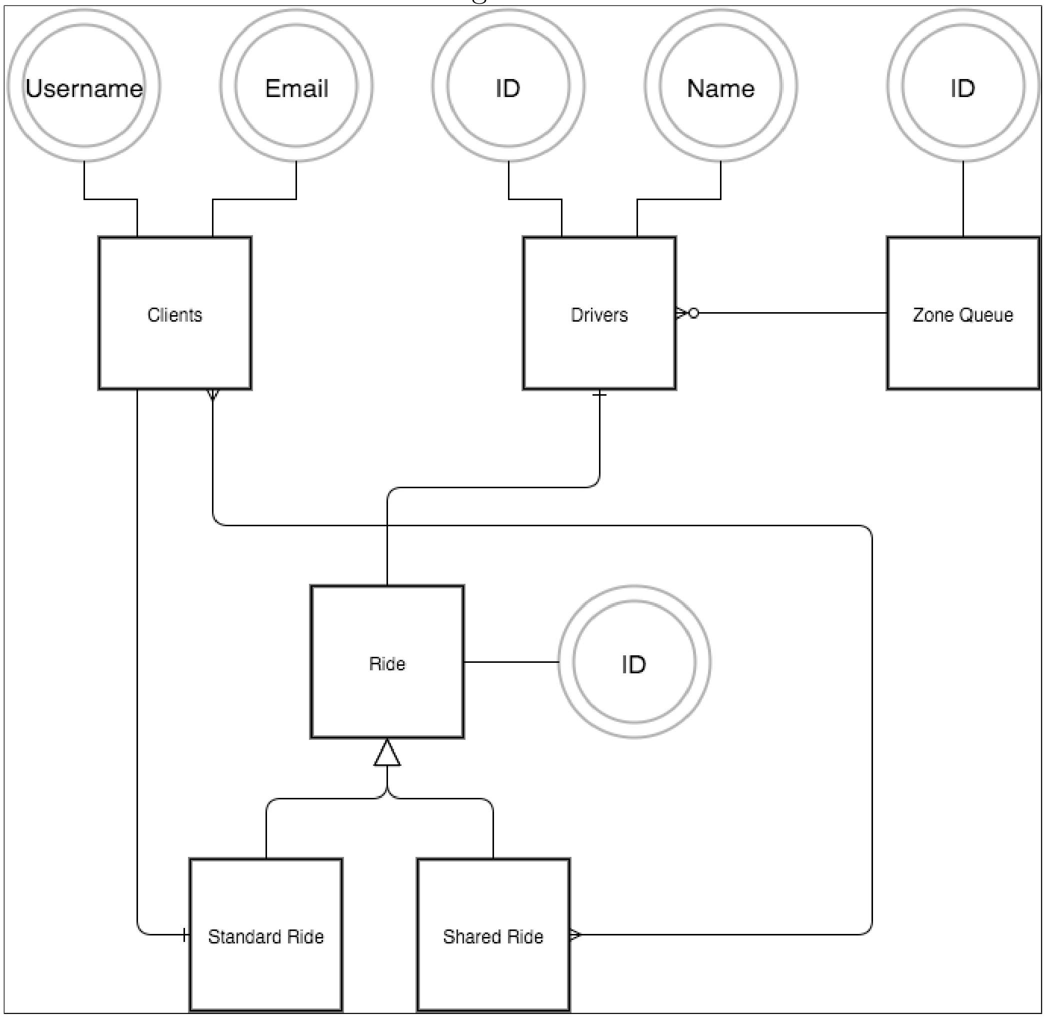
2.4 Deployment view

In the deployment of the systems we have to take in consideration the following elements:

2.4.1 Database

We selected MySQL as DBMS with the embedded InnoDB engine. The DBMS will guarantee the ACID properties for data stored in the system. We won't provide any further details about the inner design of the DBMS, in fact it will be used only through the interfaces that it provides, as a sort of black box.
The only aspect we have to take in consideration in the design of our system is the structure of the tables which will be present in the Database. Therefore a draft of a E-R schema of the tables is provided:

Figure 2.3:



As stated before in the high level components section 2.2 the only other component authorized to communicate with the Database will be the Back-end Server. In order to guarantee the security of the data involved in the transactions it will be necessary to grant only the minimum possible level of permissions to all the process that will retrieve or store data in the DB (This aspect will be better analyzed in the Back-end Server **??**).
In addition to this the security of the physical system will have to be taken in high consideration. Especially in the case the DB tier will be hosted in a cloud environment, it will be necessary that all the data and all the communications will be encrypted with strong security standards. As many episodes have shown in the recent past, the loss or worst the theft of the customers' precious data often means the immediate and irrevocable death

of the company involved in this kind of episodes.
In the component interfaces section 2.6, we will analyze in detail the interfaces that the database shows to the components that interact with him (in our case the Back-end Server).
Finally we plan to make large use of foreign key in the design of the system to guarantee the data integrity among the DB. For more advanced type of integrity we will rely only on the mechanism provided by the Java EE-based business logic that resides in the Back-end Server, not making use for example of the trigger function provided by the DBMS.

2.4.2 Back-end Server

The Back-end Server will represent the business logic tier of our system, and the application logic will be provided by the Java EE platform, which runs on the GlassFish Server.
The business logic will be implemented using *Enterprise JavaBeans (EJB)*. We will make use of both stateless and stateful beans, even if we will prefer the use of stateless beans in order to avoid a too much resource-demand in the Back-end. The store of the state of users, rides etc. will be done using the Database and a suitable architecture.
The Back-end Server exposes RESTful API to allow the Web tier and the App proxy to use the services it provides.

2.4.3 Web Server

The core component of the Web Server will be GlassFish Server, we will rely on the Java Web Components for managing the presentation layer (JSF), that implement the MVC (Model-View-Controller) architecture.
In this tier there is no application logic implemented, all the logic is situated in the Back-end tier. The Web uses the interface provided by the Application level, in particular the RESTful API.

2.4.4 Mobile App

Also this is a a presentation layer, and as the Web tier do not implement any application logic. The communication with the Back-end Server is done by an App Proxy, also implemented using Java EE technology.

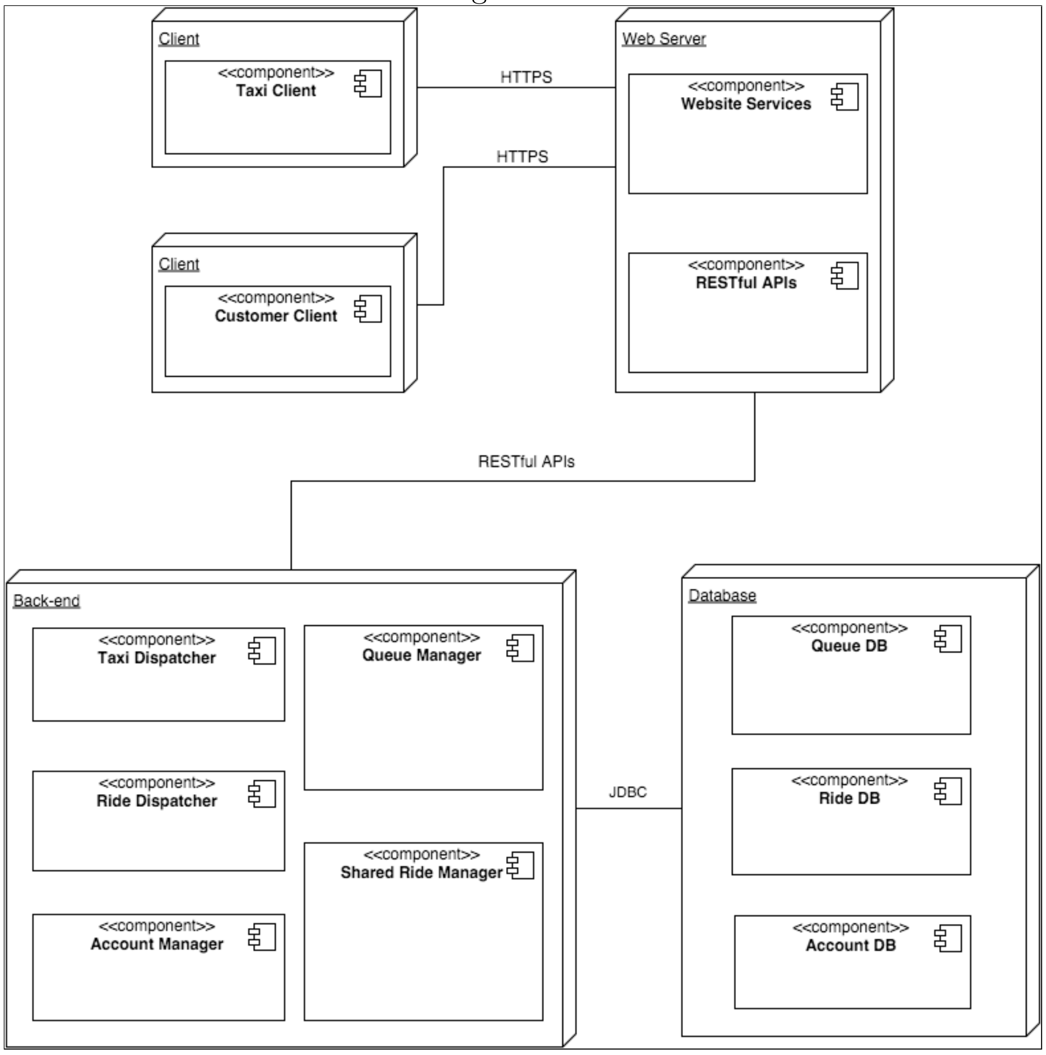
2.4.5 App Proxy

This tier allows the mobile clients to communicate with the Back-end Server, maintaining the nice properties regarding the security considerations made before. In this way we maintain a high level of separation between the external world and the Back-end, in order to avoid all the problems seen before.

2.4.6 Deployment View Diagram

With the design structure that we have made until now we have no big restrictions on how distributing the various tiers on physical machines.
Obviously a separation between for example the Database tier and the Back-end Server is highly recommended, but in principle (for example in the development environment) all the tiers can be physically deployed in the same machine.
Thanks to the use of standardized interfaces there wouldn't be particular big issues when the tiers will be distributed on various machines. For example all the communications between the Database and the Back-end server will be done through the standard network interface provided by the MySQL server.
Obviously some particular attention must be paid to hypothetical latency issues, especially between the Back-end Server and the Database. This two tiers must be deployed possibly on two private network segments that have an high bandwidth and low latency.
We now take advantage of a deployment view diagram in order to show how the components of the system identified in the previous section are mapped on the physical layers:

Figure 2.4:


Client
<<component>>
Taxi Client
HTTPS
Web Server
<<component>>
Website Services
HTTPS
Client
<<component>>
Customer Client
<<component>>
RESTful APIs
RESTful APIs
Back-end
<<component>>
Taxi Dispatcher
<<component>>
Queue Manager
<<component>>
Ride Dispatcher
<<component>>
Shared Ride Manager
<<component>>
Account Manager
JDBC
Database
<<component>>
Queue DB
<<component>>
Ride DB
<<component>>
Account DB

2.5 Runtime view

This section is devoted to describe the dynamic behavior of the operation of the system.
For this purpose we will use some sequence diagrams in which each line represent a component of the system. The interactions between the components that live in the Back-end Server and the Database are not represented, because they will be entirely managed by the Java EE platform (in particular by the Java Persistence API).

Figure 2.5: Sequence Diagram relative to the Taxi Call

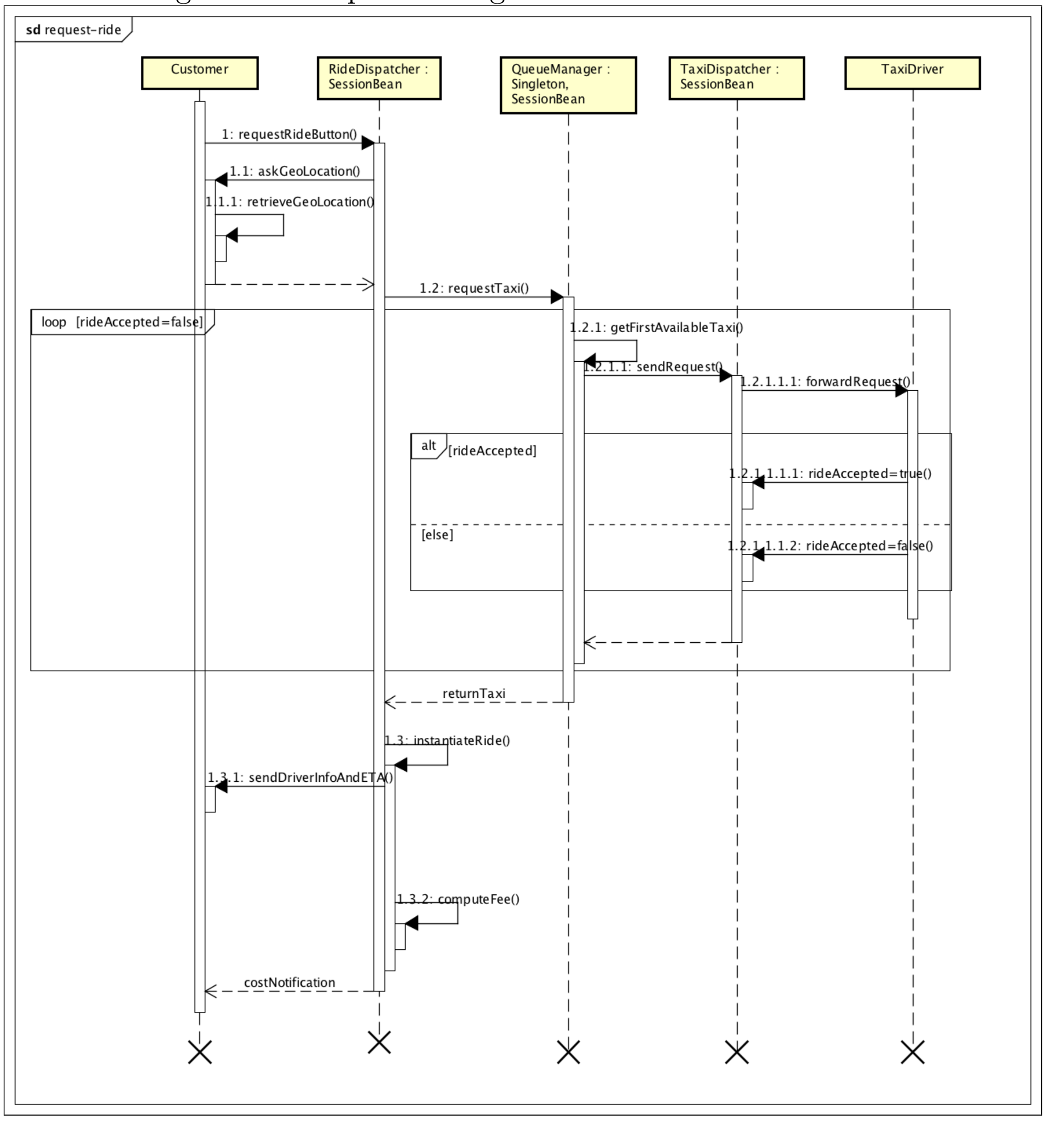

sd request-ride
Customer
RideDispatcher : SessionBean
QueueManager : Singleton, SessionBean
TaxiDispatcher : SessionBean
TaxiDriver
1: requestRideButton()
1.1: askGeoLocation()
1.1.1: retrieveGeoLocation()
1.2: requestTaxi()
loop [rideAccepted=false]
1.2.1: getFirstAvailableTaxi()
1.2.1.1: sendRequest()
1.2.1.1.1: forwardRequest()
alt [rideAccepted]
1.2.1.1.1.1: rideAccepted=true()
[else]
1.2.1.1.1.2: rideAccepted=false()
returnTaxi
1.3: instantiateRide()
1.3.1: sendDriverInfoAndETA()
1.3.2: computeFee()
costNotification

Figure 2.6: Sequence Diagram relative to the Driver Enqueueing

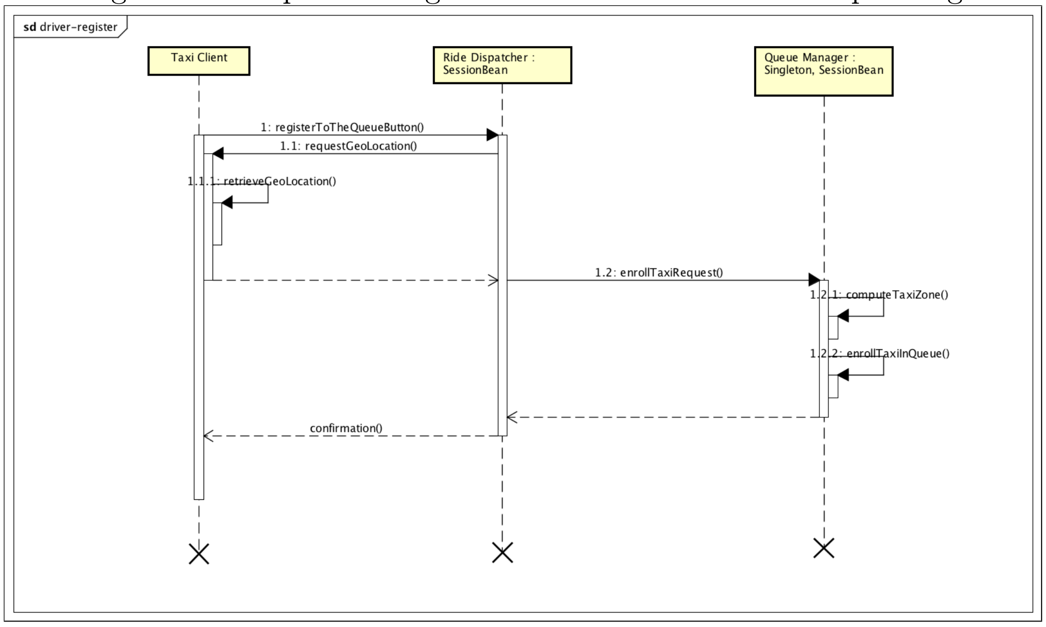
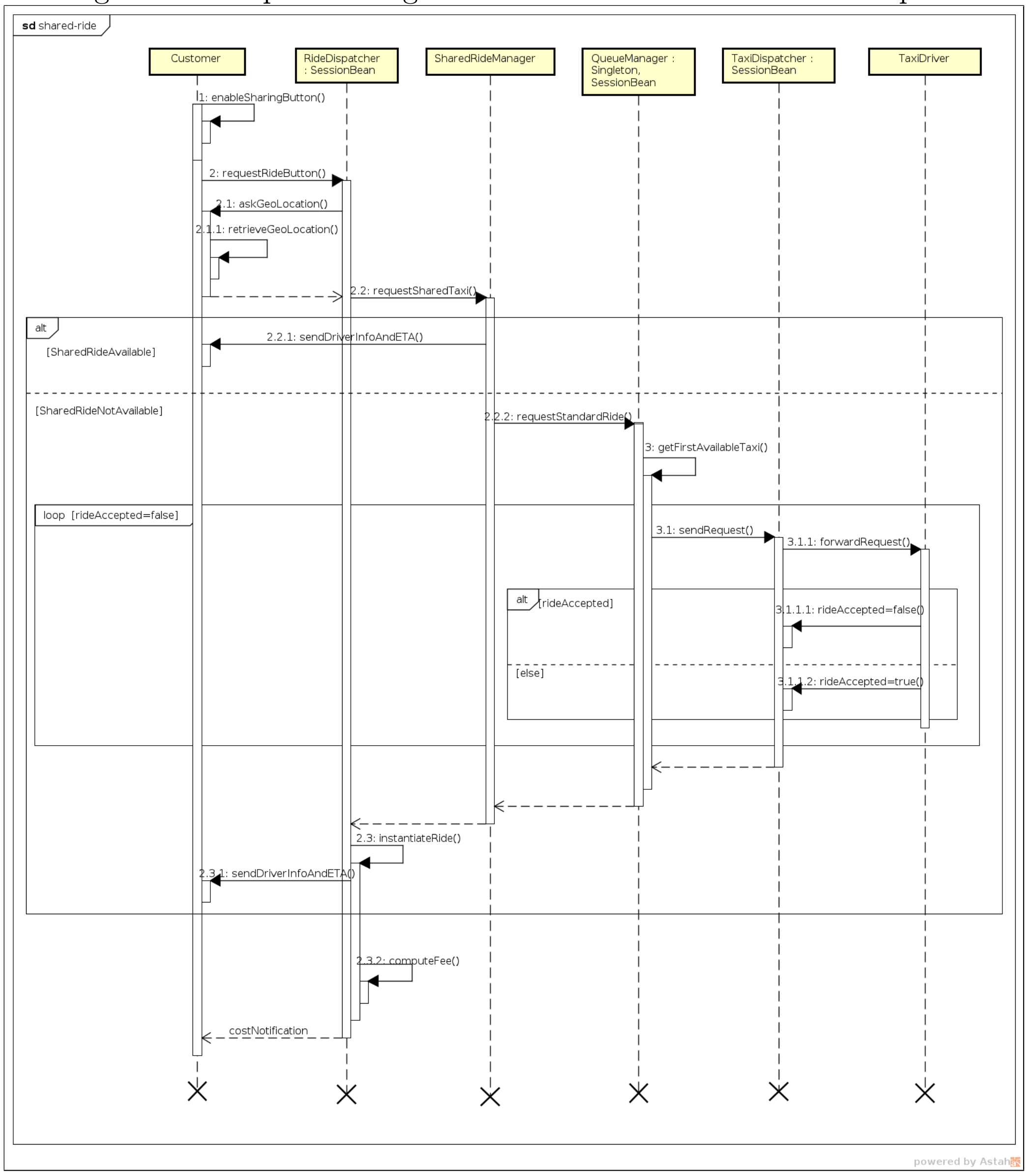

sd driver-register
Taxi Client
Ride Dispatcher : SessionBean
Queue Manager : Singleton, SessionBean
1: registerToTheQueueButton()
1.1: requestGeoLocation()
1.1.1: retrieveGeoLocation()
1.2: enrollTaxiRequest()
1.2.1: computeTaxiZone()
1.2.2: enrollTaxiInQueue()
confirmation()

Figure 2.7: Sequence Diagram relative to the Shared Ride Request



2.6 Component interfaces

2.6.1 Interface between the Database and the Back-end Server

The communications between the Database and the Back-end server take advantage of the JDBC interface provided by the Java EE platform. In particular, the implementation in the Back-end of the relational data management is done by using the Java Persistence API, that uses the standard network layer for all the communications. This fact allows us to abstract from the implementation-level details here, and also allows us to make no restrictions on the deployment of DB machines on the physical layer. For example the DB machine and the Back-end server can be deployed on the same machine, or on different ones without any design constraint.

2.6.2 Interface between the Back-end and the Web Server/-Mobile App Proxy

The Front-end layer of the system will communicate with the Back-end server through a RESTful API that uses HTTPS standard protocol as vector. The implementation of the API is done by the use of the JAX-RS framework, that is a Java-EE framework expressly designed for the implementation of RESTful Web Services. These APIs will be implemented during the coding of the components.
The use of this kind of APIs brings many advantages in our system. For example it leaves open the possibility for future extensions of the service. Obviously a high detailed documentation for the API must be developed in parallel with the system development, in order to make easier the realization of future extension of the system. Therefore this type of design for the interfaces can also simplify the interactions between the components themselves, because in this way we define in a specific way the functions and the parameters that a function call can use. This allows to treat the development of the various components independently from each other, once the interfaces are defined.

2.6.3 Interface between the presentation layer and the Front-End

The communications between the presentation layers (Mobile App and web browsers) will use the standard HTTP(S) protocol to communicate with the

Web Server (or in the case of the Mobile App with the App Proxy). We will use the GlassFish Server as Web Server. The use of HTTPS must be enabled by default and should not be optional. The communications necessary to the system will indeed regard very important and sensitive informations. The use of plain HTTP can not guarantee a minimum security level for the application and for the data involved in the standard operations of the system.
With the use of a standard interface such as HTTPS, one can also bring advantages regarding the development of the presentation layer. Indeed we do not require any type of dependency on the customer client in the case of the Web Interface (except obviously a modern Web-Browser). Also in the case of the Mobile App the use of HTTP(S) means that the Mobile App can simply be a wrapper for the Web Interface, making the development of itself for the various platforms (iOS and Android) very effortless.

2.7 Selected architectural styles and patterns

2.7.1 SOA

Our systems follows the SOA (Service Oriented Architecture) style. Indeed it exposes to the clients a service. At a high level we can see that the system follows the client-server architecture for the interaction between the external world and the system. The choice was by our point of view compulsory. For this kind of system a peer-to-peer architecture made absolutely no sense. The core operations that the system must provided need to be centralized in a single entity (this does not mean that the deploy of the system must be done on a single tier, these are two different aspects).

2.7.2 Cloud Computing

Regarding the deploy aspect, as stated before, we highly recommend the use of a cloud infrastructure, at least for the Database tier and for the Back-end Server. Through the use of an IaaS we can easily have the low latency and high bandwidth properties that we wanted for the communications between the Back-end tier and the Database tier. Using the high scalability nature of the cloud resources we can also adjust the resource dedicated to the system. For example during the Beta-Test phase we can gradually tune the resources dedicated to the various layers, for example dedicating more computational power to the Database or to the Back-end. This operations cannot be done on a physical architecture, because a little adjustment would require hours

or days of work. Besides the cost of renting a cloud infrastructure is often smaller than the cost spent in creating and maintaining a physical infrastructure.
Obviously the use of a Cloud Infrastructure introduces some issues that need to be taken in consideration. The first one is that we have to pay double attention in the security of the data and the communications, because we don't have direct control on them. But this isn't only a issue. In the case of an in-home infrastructure we also have to pay much attention to the security of the data. For example we have to make regular remote backups in order to protect ourself against accidental or malicious events. This means having an high bandwidth and an high amount of data consumed. Using instead a Cloud Infrastructure we can rent resource in order to optimize the exchange of data between the machines involved. In fact we can rent an IaaS infrastructure requiring that all the database machines and the back-end ones connected between themselves with an high speed network (for example an 10 G bps). The connection with the external world instead can be done with more standard connection (for example 1 G bps or 100 M bps).

2.7.3 Thin Client

During the design of the whole system we took for granted that the computational power of the clients was very reduced. Indeed all the logic and the data are processed and stored on the server side of the application. We made this choice in order to enable all kind of devices (in particular the mobile clients, that generally have a limited computational power) to offer acceptable performances during the usage of the system. In particular the majority of the computational load is assigned to the Server side of the systems, that has the required resources to manage the logic of the application.

2.7.4 MVC

The presentation layer uses a MVC (Model-View- Controller) as design pattern. We made this choice because MVC is the most used pattern in Object Oriented Architecture, and Java offers a lot of frameworks and support to the development done using this design pattern.

2.7.5 Client Server

The Client Server design pattern can be recognized in many aspects of our system.
The whole system can be viewed as a big architecture where the client of the

user contacts a server side part of the systems that offers a service. Also the Back-end can be viewed as a client that contacts the Database that acts as a server. Therefore the Mobile App or the Web App obviously are clients that contact the Web Server.

2.7.6 Plug-in

The structure of the architecture permits in the Application Logic the development of Plug-in for the Application Logic. We can see an example in the management of the shared ride. A plug-in component that's called "Shared Ride Manager" is "added" to the core application logic in order to introduce a new functionality.

2.8 Other design decisions

An important decision we took for the development of this project is to establish an alpha and beta test phases that have to be conducted before the final delivery of the product.

2.8.1 Alpha Test

In this phase we will provide a preview of the application to a restricted and selected group of people (that ideally must be involved in the project) and gave them access to a preview of the system. In this way we can easily find issues and problems (for example in the behavior of the user interface application) and correct them immediately in the development of the software. Obviously this kind of product can not be addressed to non-technical people, because we expect the User eXperience to be very poor, and too frustrating for a simulation of normal usage.

2.8.2 Beta Test

The beta phase instead is intended for a larger set of people, this time also non technical people. At this point all the features have already been implemented. Due to this fact the User eXperience must be almost production-ready, because for this phase we have to involve a significant number of taxi drivers and customers. Marketing strategies (that are beyond the purpose of this document) must be found in order to attract a selected audience for the beta phase. At this point we have to be sure that the back-end is ready for accepting at least some concurrent requests. This phase is crucial for the tuning phase of the resources allocated to the system. Here we have to find a basic configuration that can work in a standard environment. Then we have to develop the suitable scaling and provisioning techniques that enable the system to scale up the allocation of resources when it is needed.

2.8.3 External Maps Service

For what concerns the aspect of the geo-location we plan to use an existing service in order to avoid to re-implement all the functionalities that we need for the normal operation of our system.
In order to do this the system will us an external service, **Google Maps**, that we will use to achieve all the operations of distance calculation, visualization and geolocation.
The main reasons to do that are:

- We don't have to re-create all the map structure of the city. We can exploit the great work done by Google in the collection of data.
- We can use the services offered by the APIs both server side (calculations on distances and similar) and client side (We can use the map visualization features and the location APIs provided by the **Google Location Services**)
- We can rely on a very stable and well-know service instead of an handcrafted service.

  Obviously a further investigation about the policy that Google offers for the use of his services have to be done(Premium plans that have no limitations on query to the system, highest priority on the other traffic etc.)

Chapter 3

Algorithm Design

3.1 Zone queue size check and reallocation

For managing the taxi queues in a vast area, i.e. the city of Milan, having a fixed allocation for the taxi queues is not optimal. In fact, some problems may occur with this paradigm, among them there are:

- If the taxi queue covers a vast area where there is a large amount of taxis, due to the fact that the queue is managed with a FIFO policy (necessary for a fair management of the queue) there is the risk that a ride request is assigned to a taxi that is far from the pick-up location. If we allocate instead more taxi queues and each of them covers a smaller area, we can assume that in average the distance between the current taxi rider location and the pick-up location is reduced, and accordingly the estimated time of arrival of the taxi is diminished.

- In the opposite case, if the system observes a lack of taxis in some zones (for example during nighttime), it can rearrange the taxi queues in order to join some of them and find a situation where there is at least a minimum number of taxis in each zone. This solution obviously in average increases the estimated time of arrival of the taxi, but allows customers to find a taxi also during time slots (as nighttime), when usually it won't be a problem to wait a little more in exchange of a sure availability of the service.

In case of zone join, some particular care is devoted to the fairness:
If the taxi drivers are added in the new zone in random order, the first taxi of a queue could end up at being in the middle of the newly created queue. To avoid this kind of scenario we implemented the join with an alternate picking from the two merging zones. Likewise, in the case of zone split, if we

assign the taxi drivers randomly to the two new queues, some bad scenarios may happen:
A taxi that is geographically located at the border the zone to be split may end up in a zone which no more corresponds to its position. To avoid this kind of situation we are checking the saved position of every driver before choosing which queue to enqueue it in.

Syntax: Python flavored pseudo-code

```
periodicZoneCheck():
  foreach zone in zones:
    # Split the zone in two new zones
    if zone.size < minimumTargetSize:
      QueueSystem.pause()
      toMerge = zone.selectForMerge()
      newZone = new Zone()
      newZone.fairExtend(zone, toMerge)
      zones.remove(zone)
      zones.remove(toMerge)
      del(zone)
      del(toMerge)
      zones.add(newZone)
      QueueSystem.resume()
    elif zone.size > maximumTargetSize:
      QueueSystem.pause()
      firstZone = new Zone()
      secondZone = new Zone()
      zone.locationSplit(firstZone, secondZone)
      zones.remove(zone)
      del(zone)
      zones.add(firstZone)
      zones.add(secondZone)
      QueueSystem.resume()


selectForMerge(self):
  minSize = MAX_ZONE_SIZE
  minZone = null
  foreach zone in self.adjacentZones:
    if zone.size < minSize:
      minSize = zone.size
      minZone = zone
  return minZone


fairExtend(self, firstZone, secondZone):
  while not (firstZone.isEmpty() or secondZone.isEmpty
   ()):
    self.extend(firstZone.pop())
    self.extend(secondZone.pop())
  if firstZone.isEmpty():
    zone.extend(secondZone)
  else
    zone.extend(firstZone)
  self.extendArea(firstZone)
  self.extendArea(secondZone)


locationSplit(self, firstZone, secondZone):
  foreach taxi in self.taxis:
    self.remove(taxi)
    if firstZone.covers(taxi.actualPosition):
      firstZone.add(taxi)
    elif secondZone.covers(taxi.actualPosition) :
      secondZone.add(taxi)
    self.area.split(firstPart, secondPart)
    firstZone.area = firstPart
    secondZone.area = secondPart
```

code/zonequeuecheck.py

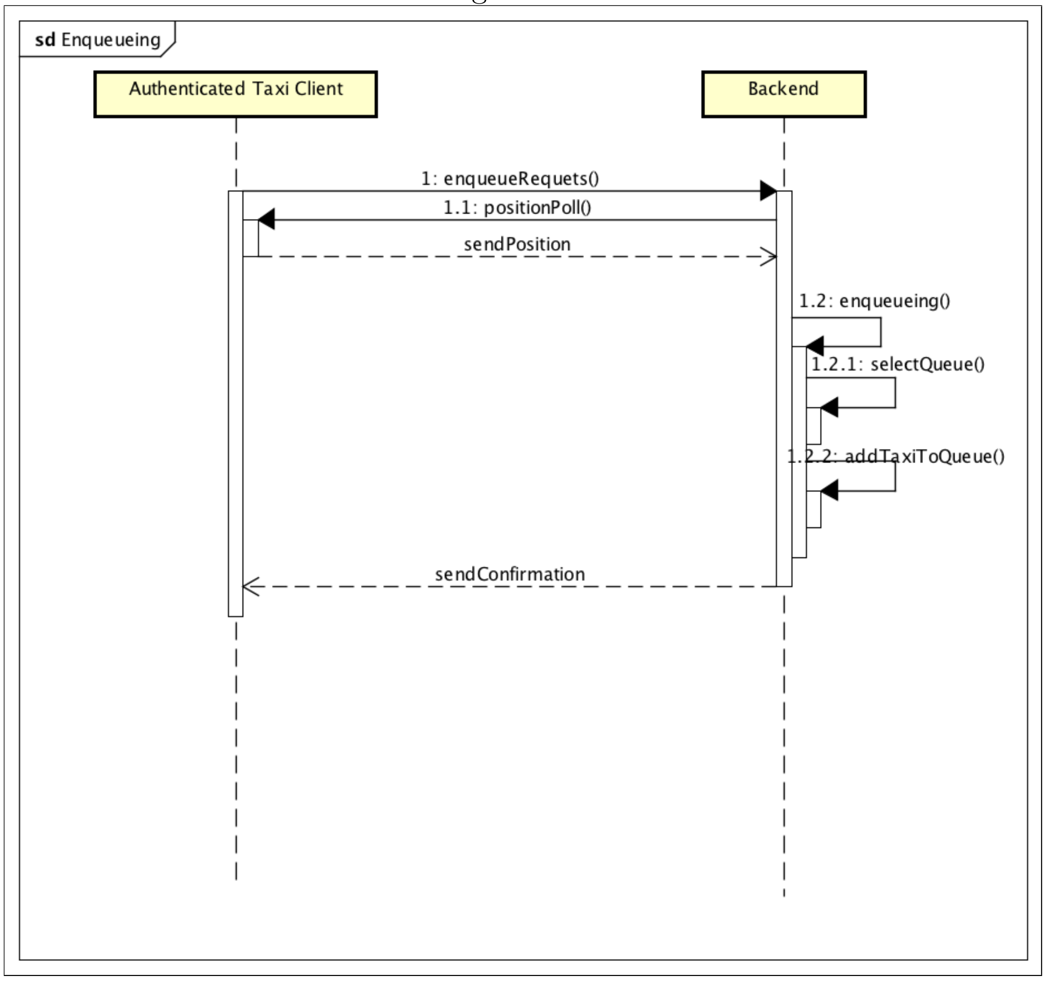
3.2 Taxi enqueuing

In this section we present a series of Flowcharts with the aim of better clarifying the structure of the enqueuing procedure from the perspective of the taxi driver. Let's analyze in details this operation:

- The taxi driver at any time during his service can open the myTaxiService and, after the authentication procedure, ask for the enqueuing. At this point the mobile application located on the device of the taxi driver activates the procedure to retrieve the current location of the cab. After this the client application sends the information about the location to the back-end system, that now activates the procedure responsible of the correct enqueuing of the taxi driver. This functionality heavily relies on the previously analyzed zone check and reallocation function. After the system has elected the right taxi queue, it sends a confirmation to the client side of the application about the enqueuing.

Sequence diagram

Figure 3.1:



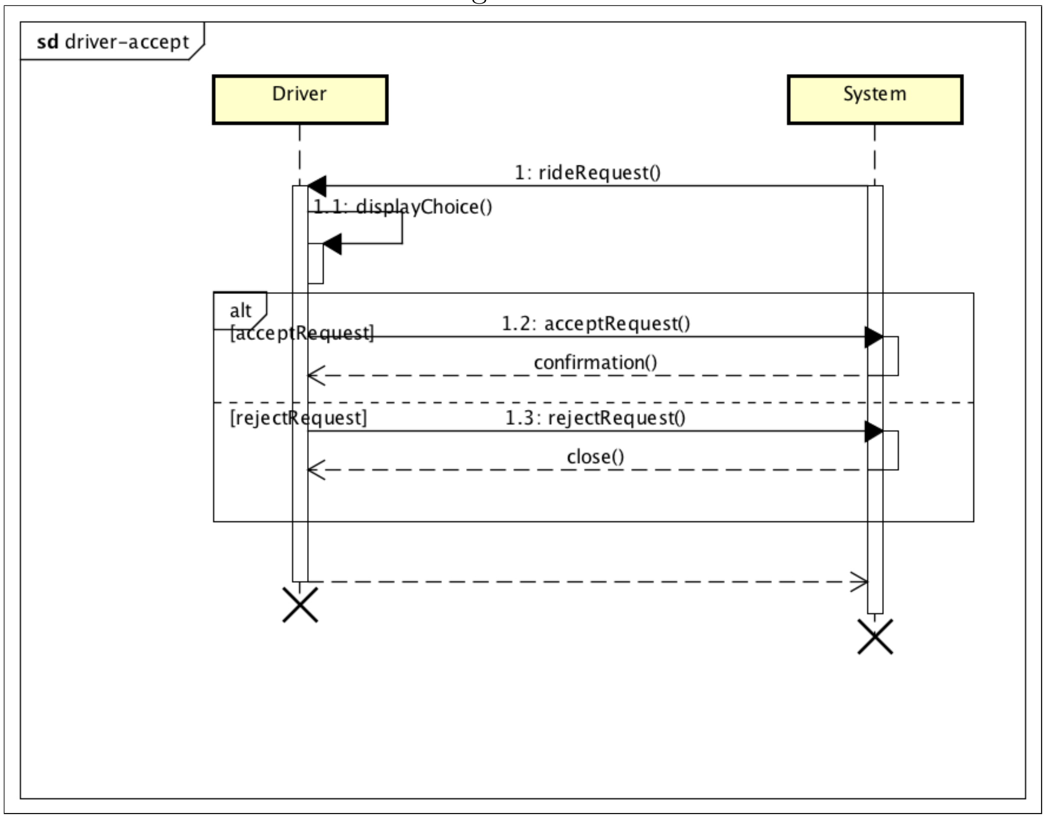
3.3 Ride Start

In this section we present a series of Flowcharts with the aim of better clarifying the structure of the acceptance/rejection of a ride from the perspective of the taxi driver. Let's analyze in detail this operation:

- After the taxi driver enters in a local queue he is eligible for a ride request. The algorithm that is specified in the next paragraph works in the following way: after the system receives a ride request from a customer, it will select a candidate taxi driver. At this point the back-end sends the request to the smart-phone associated with the selected taxi driver. Then the taxi driver will have to decide to accept or reject the ride request in a reasonable amount of time (that will be tuned on the base of various factors, in order to minimize the time that the customer will have to wait between the ride request and the actual confirmation by the system).

Sequence diagram

Figure 3.2:



3.4 Taxi queue management

In this part we will explain how the algorithms that manage the queue of the taxi works. The main idea is to use a deque which is a double ended queue. A structure of this kind allows us to insert and remove element from the top or from the bottom of the queue, according to our choice. Deques are very lightweight to implement, in fact they are just a particular kind of double-linked lists in which you can only insert an element on the two ends of the list. This simple implementation has the advantage to effectively maintain the fairness towards the taxi driver. A policy that we think is appropriate to guarantee the fairness is to put a taxi driver that has rejected a ride request at the end of the queue, in order to allow the highest number possible of taxi driver to receive a ride request. In this way we also guarantee that an unfair behavior as registering in the queue without being ready to accept a ride is penalized by the system and thus discouraged. So the queue will be managed in this simple way:

- When a new taxi rider registers to a taxi queue it is placed at the bottom of the deque.
- When the systems needs a taxi driver to satisfy a ride request it simply pops the first element from the deque. In case the driver accepts the request it is finally removed from the deque, otherwise it is put at the bottom of the deque.
- By applying the previously describedBelie zone management policies we virtually never have an empty deque, unless we are in a limit situation when we have one or no taxi driver in the deque, that is a situation that should never occur, see the section about the queues reallocation for further informations.

Syntax: Python flavored pseudo-code

```
class Zone():

  def add(self, taxiDriver):
    self.zoneDeque.appendleft(taxiDriver)

  def pop(self):
    return self.zoneDeque.pop()

  def extend(self, secondZone):
    foreach driver in secondZone:
      self.zoneDeque.appendleft(driver)

  def pushback(self):
    topDriver = self.zoneDeque.pop()
    self.zoneDeque.appendleft(topDriver)

  def remove(self, taxiDriver):
    self.zoneDeque.remove(taxiDriver):

  def covers(self, position):
    return self.area.covers(position)

  def extendArea(self, zone):
    self.area.add(zone.area)

def rideRequest(zoneQueue):
  firstDriver = zoneQueue.pop()
  if firstDriver.confirmRide():
    return
  else
    zoneQueue.add(firstDriver)
```

code/taxiqueuemanagement.py

3.5 Taxi position update

In order to avoid an excessive load on the back-end of the system, we think that the best way to update the position of the cabs is to periodically make the clients inform the back-end of their new position, taking in account the movement of the cab and other factors. We also analyzed the opposite situation, where the back-end polls each client to be informed on any update of the position. But in this way we can't exploit an important information that only the client has, the information about the movements of the cab. In fact it is useless to update the position let's say every 10 seconds if the cab is steady in a parking spot. In this situation the update should be reduced in order to minimize the load on the back-end and also the bandwidth and battery consumed by the devices. Obviously the back-end should be able to force an update of the position, for example if it suspects that the connection has gone down for some reasons, for example after a timeout has expired. In this case the choice of a right timeout will be crucial to avoid that a taxi driver is removed from the queue due to a temporary loss of the connectivity of his terminal (e.g. the cab entered a tunnel). We think that only in the feedback of a beta program this parameter (as other ones of the system) can be correctly tuned.

Syntax: Python flavored pseudo-code

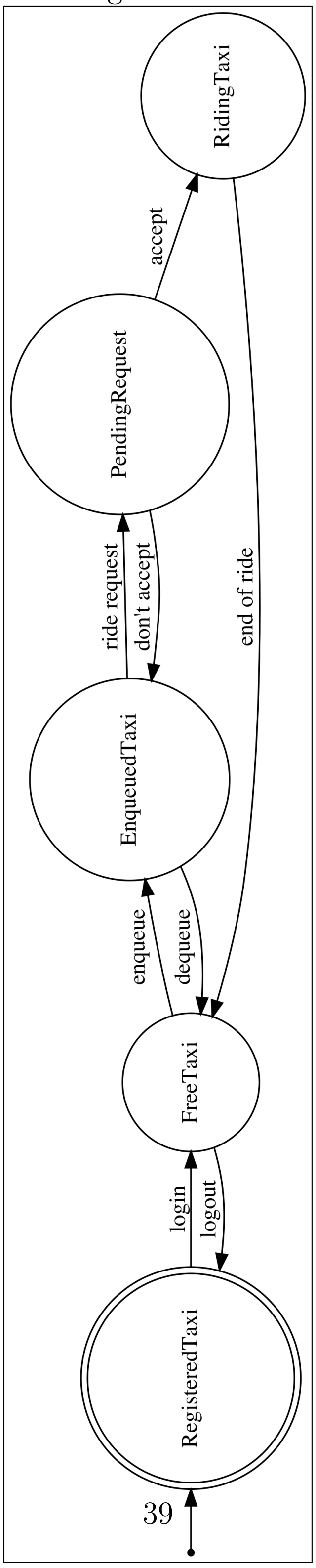
```
scheduledLoop(300, checkPosition)

def checkPosition() (
  actualPosition = System.getCoordinates()
  distance = computeDistance(actualPosition,
   previousPosition)
  if distance > 500:
    uploadPosition(actualPosition)
```

code/taxipositionupdate.py

3.6 Taxi status sequence

In this paragraph we will use a Finite State Automata model to explain in a clearer way the possible states in which a taxi cab can be. This would help in finding every possible issue in the formulation and in the implementation of the system. This representation should also help in formulating and implementing the transitions between various states, which is a critical phase of the development of this system. Indeed in the transitions we have to manage a lot of changes and variables in the system, and we should also pay attention to build the system in a way that makes it robust to any error and problem that may occur (also here we can have a drop of the connectivity, a delay in the propagation of the information through the various layers of the systems etc...).

Figure 3.3:

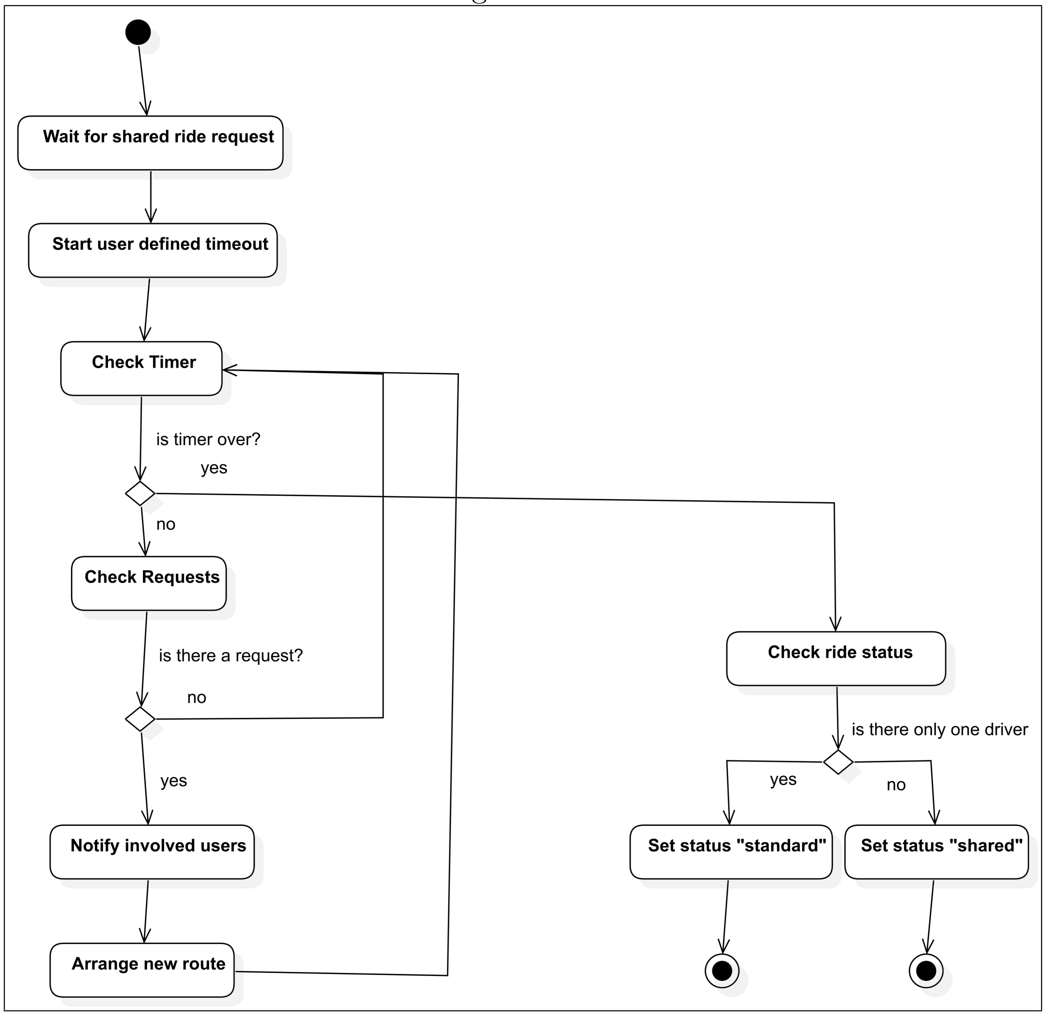

RidingTaxi
accept
PendingRequest
end of ride
ride request
don't accept
EnqueuedTaxi
enqueue
dequeue
FreeTaxi
login
logout
RegisteredTaxi

3.7 Shared ride creation

In order to clarify some aspects we put here a simple scenario that should clarify how the shared ride works: Alice opens the App on her smart-phone and decides to call a taxi, she also selects the shared ride option. At this point the system asks how much time the client is available to wait for the taxi. During that time, the system waits for other shared ride requests, and if no eligible ride requests arrive in this period the ride is classified as a standard ride. If, for example, Bob places a shared ride requests that involves at least a part of the same route of the ride request by Alice, the system notifies all the parties involved in the shared ride (Taxi driver included) and begins to arrange the optimal route for the ride. The procedure is repeated if other ride requests arrive. After the taxi driver picks up the first customer the system will no more accept other requests for the ride.

Flowchart

Figure 3.4:



3.8 Shared ride fee distribution

The sharing of the ride fee will be managed in this way: the fee will be divided equally among the customers, and will not be exactly proportional to the portion of the ride covered. Instead we think that the base taxi fee (the one that is independent from the number of kilometers of the ride) should be split in equal parts among the customers. The remaining fee instead can be charged proportionally to the kilometers each customer has traveled. Other choices can be evaluated, but we think that we are not the best candidates to do it. This is a problem that should be discussed with the parties involved, in case asking to the representatives of the taxi drivers, the representatives of the institutions and also by making surveys. Also in this case we think that this kind of choice should be left open during the beta program, in order to meet everybody's needs.

Syntax: Python flavored pseudocode

```
getPassengersFee(SharedRide currentride):
    passengersnumber = currentride.getPassengerList().len
      ()
    kilometerstotal = currentride.getKilometers()
    fixedfeetotal = currentride.getFixedFee()
    variablefeetotal = currentride.getVariableFee()
    fixedfeequota = fixedfeetotal/passengersnumber
    kilometersquota = []
    passengersvariablefee = []
    feequotas = []

    for passenger in currentride.getPassengerList()
        kilometersquotas.append(passenger.
      kilometerspercurred()/kilometerstotal)

    quotassum = sum(kilometersquotas)

    for quota in kilometersquotas
        passengersvariablefee.append((quota/quotassum)*
      variablefeetotal)

    for variablefee in passengersvariable
        feequotas.append(fixedfeequota + variablefee)

    return feequotas
```

code/feequota.py

Chapter 4

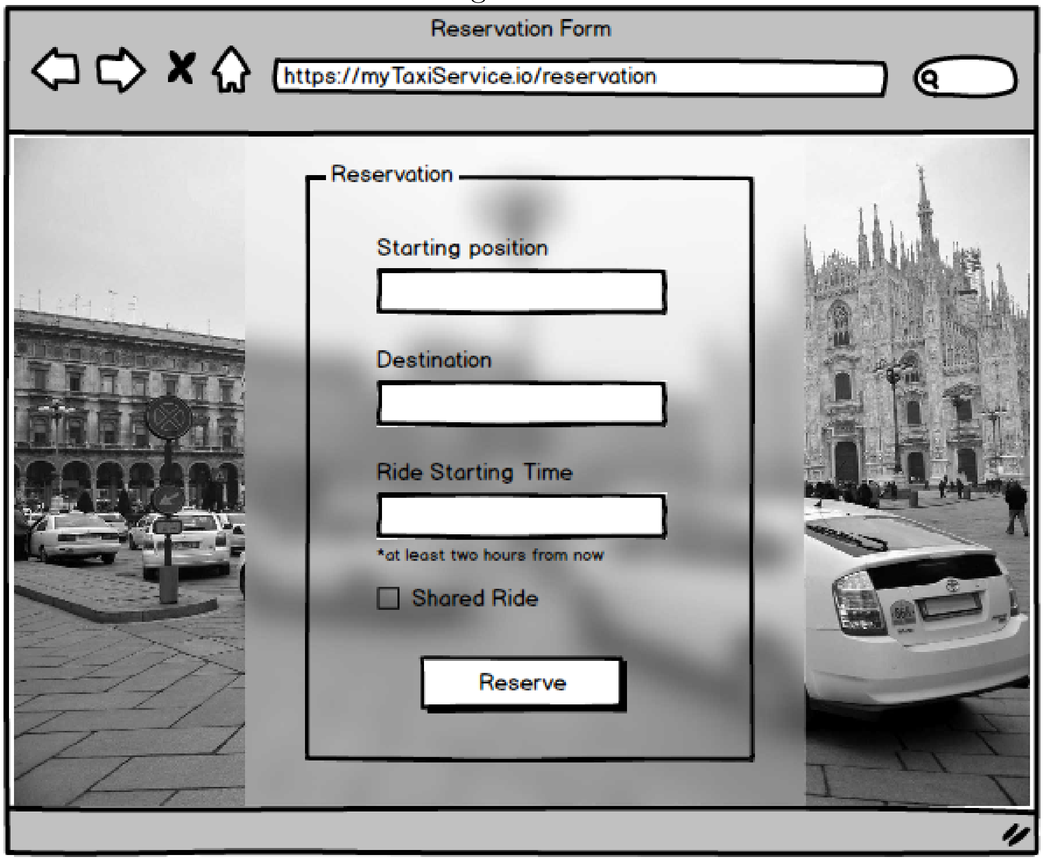
User Interface Design

A first draft of the User Interface Design has already been provided in the RASD document (for more information you can see RASD, paragraph 3.1.1). In this section we will expand and discuss more over some functionalities of the interface, especially on the Shared Ride and Reserved Ride, due to the fact that now all the major design choices had been done, and we can deepen in a more complete technical analysis.

4.1 Reservation of a Ride

This is a mock-up for the web interface that will allow a customer to book a Reserved Ride. As stated before a Reserved Ride must take place at least two hours after the booking. At the moment of the booking the customer must provide in addition to the desired starting time also the desired pick-up and the drop location.

Figure 4.1:



4.2 Shared Ride

We noticed that the mock-up that shows the User Interface for the customer is missing the part that ask for the desired destination of the ride. So in this paragraph we will present the mock-up of the restyled interface.
The main changes we introduced are the insertion of a research box for both the pick-up and destination location. In this way the user can also enter the address of the desired location, and the interface will move the pointer to that location. As usual the user can also move manually the pointer in order to express a more precise position. These are the two mock-ups restyled:

Figure 4.2:

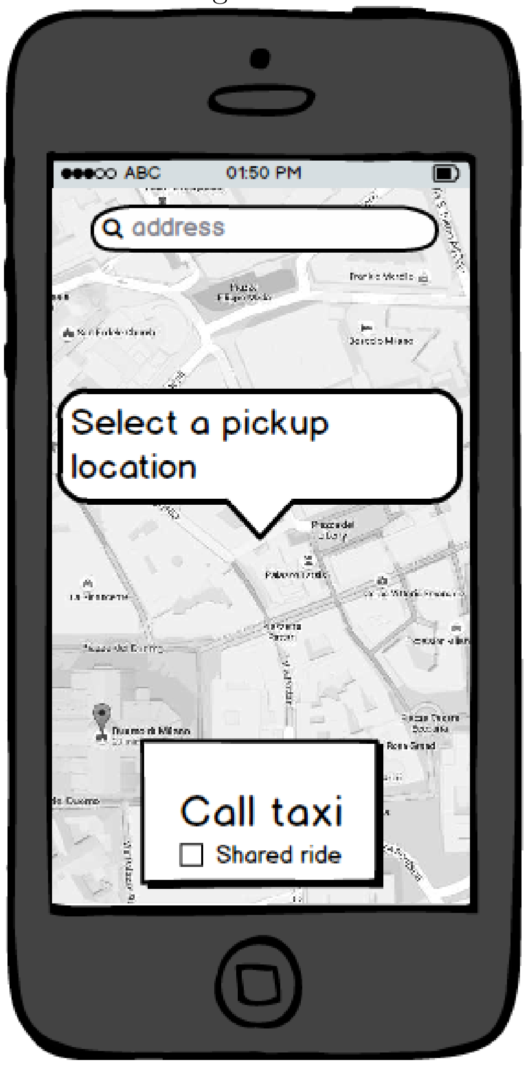
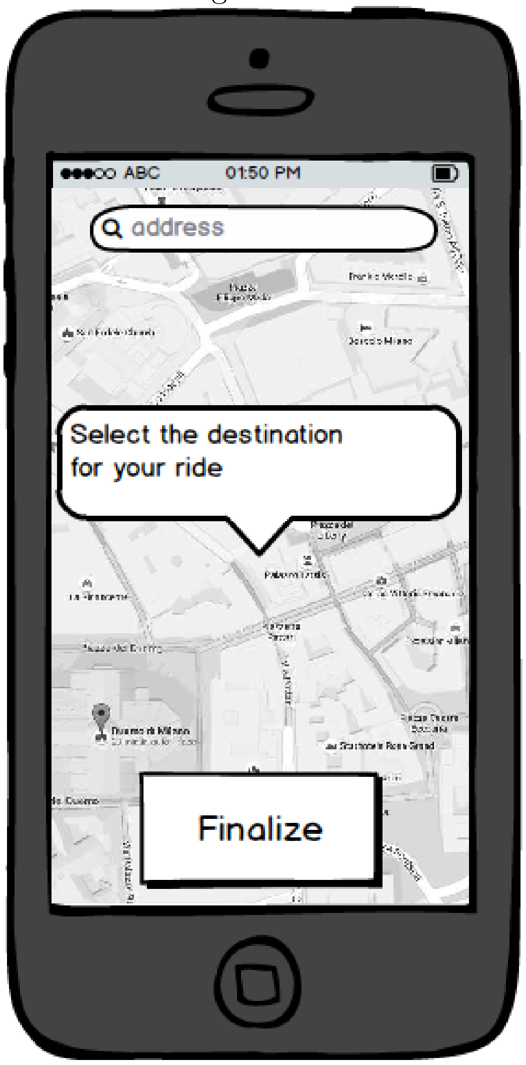

ABC
01:50 PM
address
Select a pickup location
Call taxi
Shared ride

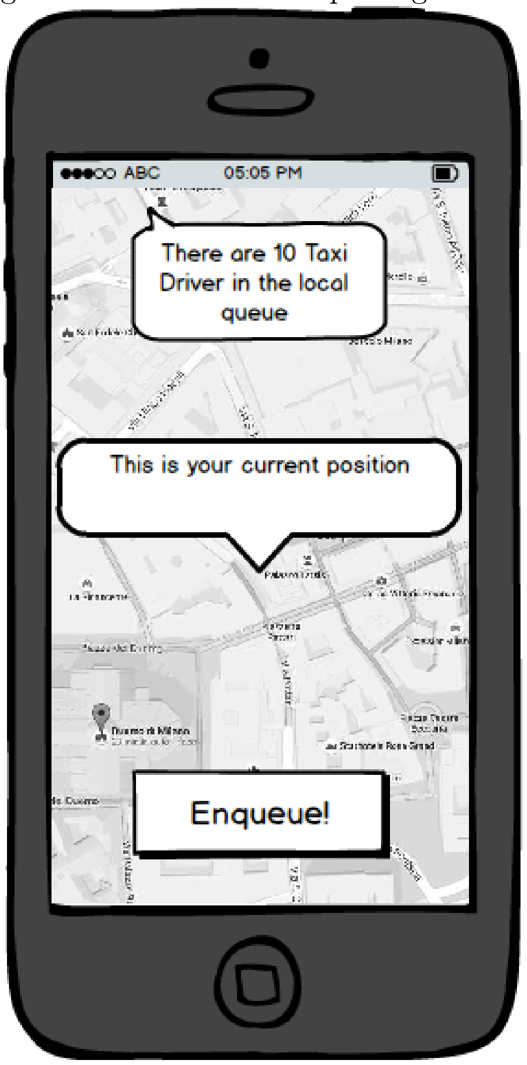
Figure 4.3:


●●●○○ ABC
01:50 PM
address
Select the destination for your ride
Finalize

4.3 General consideration about User Interface Design and User eXperience

The Design of the User Interface will have a great impact on the overall good success of the project. In fact the interface will be the main point of contact between our system and the users (except for those kind of user that will use the API's). So a good design that takes in account the usability of the User Interface and that guarantees a great User eXperience will be a key factor for the overall development. This means that we have to provide a complete and at the same time not confusing Interface. All the main functionalities available must be accessible with few clicks or taps on the interface, but at the same time the interface must remain simply and user-friendly. This must be especially true for the interface dedicated to the taxi drivers. In this case indeed they have to operate with the interface without being distracted, because they may be driving around. Through all the design phases we payed particular attention to this aspect, and in fact the interaction required to the taxi driver is the minimum. After the authentication procedure that must be performed with the vehicle stopped, both the enqueuing procedure and the acceptance/rejection of a ride can be done with a single tap. Particular attention must be payed in the choice of fonts and colors for the interface. The designed background must not interfere with the legibility of the text parts. As usual this must be in particular true for the mobile interface that with high probability will be used by the taxi driver during their ride. In this case we preferred not to put any distraction factors in the interface, as a background image. Instead we have put a interactive map, on which the taxi driver can receive in real-time updates about the customer location. The notification for accepting/rejecting the ride will exist in the form a pop-up, in order to attract the attention of the driver.
For similar reasons instead the interface of the customer can be a little more complete and fancy. In this case we preferred to have all the necessary elements for requesting a ride on the same page, in order to reduce errors and help the customer to spot errors in the location or time inserted.
Thanks to this observation we think that the interface for the enqueuing of the taxi driver must be revisioned, in particular adding a counter that shows the number of the taxi already in the queue, the current position of the driver, and a button that allows to enqueue. The first version with the pop-up would have been too invasive and did not show the number of the driver in the queue.
Here a mock-up:

Figure 4.4: Revisioned enqueuing interface

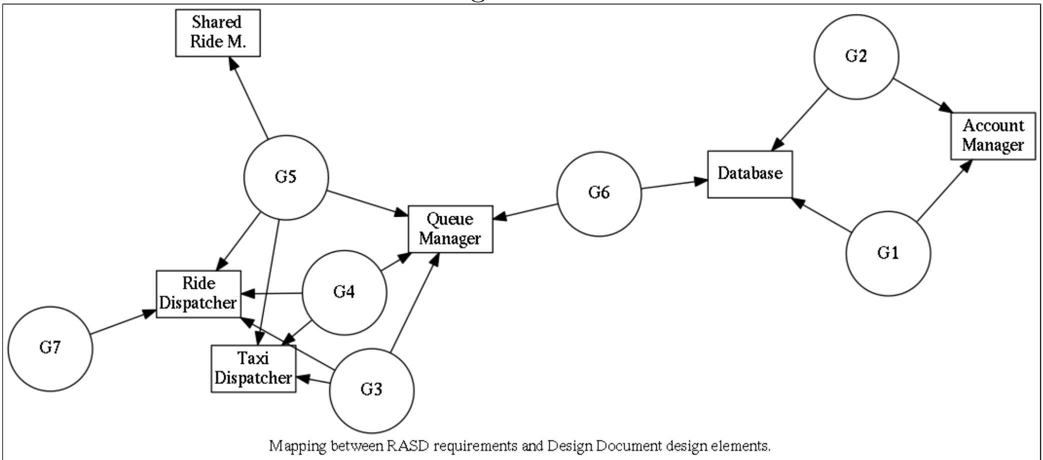

●●●○○ ABC
05:05 PM
There are 10 Taxi Driver in the local queue
This is your current position
Enqueue!

Chapter 5

Requirements Traceability

There exist a well defined relationship between the requirements examined in the Requirement Analysis and Specification Document and the design elements introduced in this Design Document. This relationship is expressed in the form of a directed bipartite and graph. One set is composed by the requirements and the other is composed by the design elements.

Figure 5.1:



Mapping between RASD requirements and Design Document design elements.

- G1: Customer Registration The customer registration is handled by the Account Manager component, which collects all the informations required for an account creation, and store them on the system Database, in the appropriate tables.
- G2: User Login The user login requirement is similar to the registration in the use of components. In fact, the user login informations such as

the passwords are stored in the central database. The whole procedure of login is carried out by the account manager.

- G3: Standard Ride This requirement is complex in its analysis, in fact it is composed by various phases: the ride manager collects the ride request from the user and asks the taxi dispatcher for an available taxi. The taxi dispatcher queries the database for the current queue status and select a taxi driver according to the algorithms defined in section 3.3. The ride status is then written into the Database.

- G4: Reserved Ride The reserved ride involves the same design elements of the standard ride, but it show a main difference: in this case the Database is accessed first of all to store the ride reservation. Only when the ride departure time is approaching, the system selects the taxi cabs and arranges the ride.

- G5: Sharing-enabled Ride The sharing-enabled ride is similar to the standard ride, with the difference that also the Shared Ride Manager is involved in the process, in fact this component handles all the shared ride functionalities, that includes for example the fee distribution, as described further in section 3.7.

- G6: Taxi Cab Enqueueing The Taxi enqueueing process is handled by the taxi dispatcher component. This component cooperates with the queue manager, by managing the current taxi position in the queues. This last component ultimately writes the data about the taxi drivers and the queues in the database.

- G7: Taxi Drive Confirmation of a Ride This requirement is handled by the Ride Dispatcher, which interconnects the taxi driver's mobile app with the current backend status. The component as a consequence of the driver actions will then update the state of the queue via the Queue Manager. This will then change the queue status in the Database.

Chapter 6

References

We list here some resources that we used for the development of this document:

- Specification Document: *Structure of the Design Document.*
- Information the modeling of diagrams for the architecural design chapter 2 found on `http://www.agilemodeling.com/`.
- Slides of the lectures.
- Slides on the Java-EE part for the choice of implementation details.
- Official Java Documentation for the identification of suitable elements for the construction of components `https://docs.oracle.com/javaee/7/index.html`.

Chapter 7

Appendix

7.1 Used Software

For the redaction of this document we used various software, here a little list:

- LaTeX framework (MacTeX on OS X and TeX Live on GNU/Linux) to generate the document.
- Various editor to edit the source file:
  - Sublime Text 3 (Beta)
  - Atom.io
  - Vim
- Self-Hosted ShareLaTeX to collaboratively edit the document.
- Git to version the source, GitHub to host the repository.
- Balsamiq Mockups 3 to create the mock-ups of the user interface.
- StarUML to create the use-case diagrams.
- Astah Professional to create the sequence diagrams.
- Gimp, Inkscape and ImageMagick to edit some images (.svg to .png).
- Draw.io for drawing some diagrams.
- Teamspeak and Hangouts used to organize conference-calls in order to work together.

7.2 Hours of Work

We tried to distribute in an equal way the work load for each team-member. In particular we tried to arrange meetings in person or conference calls for the parts where important choices had to be made. We estimated that each of us spent on average 35 hours on the drafting of this document.