Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
M.Sc. in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

myTaxiService

Software Engineering 2 - Project

DD
Design Document

version 2.0

22nd February 2016

Authors:
Alberto Maria METELLI Matr. 850141
Riccardo MOLOGNI Matr. 852416

Academic Year 2015–2016

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of the DD (*Design Document*) is to provide a representation of the *myTaxiService* software design to be used for recording design information and communicating it to key design stakeholders. This document starts from the functional and non functional requirements described in the RASD and will deal with the main *architectural* choices and *design* issues. It will focus on the architectural decomposition of the system and on the main design concerns related to both algorithms and patterns. However this document should not be considered the final draft for the architectural and design issues since in the following phases several fixing may be necessary.

DD plays a pivotal role in the development and maintenance of software systems being the blueprint for the following process of development. Being a much more specific document its audience is rather different with respect to the RASD; DD is intended to be used by project managers, quality assurance staff, configuration managers, software designers, programmers, testers, and maintainers. Since each of these stakeholders have different needs both in terms of required information and level of technical detail, DD should benefit of a mixed level of technical and informal exposition.

1.2 Scope

The *myTaxyService* is an application intended to optimize taxi service in a large city, making the access to service simpler for the passengers and ensuring a fair management of the taxi queues.

Passengers will be able to request a taxi either through a web application or a mobile app; of course the "traditional" ways to call for a taxi, like a phone call or stopping the taxi along the road, will be still available and integrated into the system to-be. The software will make the procedure of calling a taxi simpler (by using GPS information passenger doesn't need to know the address if the taxi is needed for the current position) and more usable (passenger will be provided with information about the waiting time). Moreover, by means of the application, the passenger can reserve a taxi for a certain date and time, specifying the origin and the destination of the ride.

Taxi drivers will use a mobile app to inform the system about their availability and to confirm that they are going to take care of a call (or to reject it for any reason). The software will make the taxi management more efficient: the system will be able to identify the position of each taxi by using GPS; the city will be divided in virtual zones and a suitable distribution of the taxi among the zones will automatically be computed.

1.3 Definitions, Acronyms, Abbreviations

In this paragraph all the terms, acronyms and abbreviations used in the following sections are listed.

1.3.1 Definitions

- *Request*: the action performed by the passenger of calling a taxi for the current position.
- *Confirmed request*: a request that has been accepted by a taxi driver.
- *Reservation*: the action performed by the passenger of booking a taxi for a specific address and specific date and time.
- *Waiting time*: an estimation of the time required to taxi driver to get to passenger's position.
- *Taxi code*: a unique alphanumerical identifier of the taxi.

- *Available taxi queues*: data structures used to store the references of the available taxis, also used to select the taxis to which forward a request.
- *Automatic geolocalization*: a system that provides the geographic coordinates of the user. For this document it can be either a GPS system or browser geolocalization.
- *Passengers' application*: the applications used by passengers to access to TS system. For this document it can be either PMA or PWA.
- *Login credentials*: username and password.
- *Notification*: communication from TS to taxi driver to move to a specific zone.

1.3.2 Acronyms

- TS: myTaxiService.
- PMA: Passenger mobile application.
- PWA: Passenger web application.
- TMA: Taxi driver mobile application.
- QMS: Queue management system.

1.3.3 Abbreviations

- [Gn] n-th goal.
- [Dn] n-th domain assumption.
- [Rn.m] m-th requirement related to goal [Gn].

1.4 Reference documents

[1] IEEE Software Engineering Standards Committee, "IEEE Standard for Information Technology - Systems Design - Software Design Descriptions", IEEE Std 1016™-2009 (Revision of IEEE Std 1016-1998).

[2] ISO/IEC/ IEEE 42010 "Systems and software engineering - Architecture description", First edition 2011-12-01.

[3] Software Architecture: Foundations, Theory, and Practice. Richard N. Taylor, Nenad Medvidovic, Eric Dashofy.

[4] Software Engineering 2 course slides.

[5] Federico Malucelli, Lecture notes.

[6] RASD (Requirements Analysis and Specification Document) of the *myTaxiService*.

1.5 Document Structure

This document is composed of five sections and an appendix.

- The first section, this one, is intended to define the goal of the DD, a very high level description of the main functionalists of the *myTaxiService* system and the resources used to draw up this document.
- The second is the core section of the document. It provides a detailed description of the architectural choices made to fulfill functional and non functional requirements. A first high level description of the architectural structure will be given at the beginning of the section and it will be discussed in deep, according to different criteria, in the following subsections. In particular, a *component* and *connectors* view will be described and represented using UML Component diagram. Then those components will be allocated to physical hardware devices in the *deployment view* specified by means of a UML Deployment diagram. Dynamical behavior and interaction among components will be expressed by means of UML Sequence diagram, inspired to those present in RASD diagram but more detailed.
- The third section is entirely devoted to the definition of the most significant algorithms designed for the system, the description will be given by means of pseudocode.
- The fourth section is dedicated to the user interface design. Starting from the mockup provided in the RASD and integrating information related to non functional requirements a more specific description will be given both in terms of new mockups and user interface graph structure expressed by means of UX diagrams.
- The fifth section is the link between DD and RASD: here we will emphasize how design choices described in the DD will realize the requirements expressed in the RASD.
- The appendix contains a brief description of the tools used to produce this documents, the number of hours each group member has worked towards the fulfillment of this deadline and the revision hystory.

2 Architectural design

2.1 Overview

The choice of the architectural styles and patterns suitable to meet stakeholder's functional and non requirements is typically one of the key steps of the design phase, therefore we will expose the process discussing, in order of decreasing level of abstraction, the following aspects.

- *Architectural pattern*[1]: is a named collection of architectural design decisions that are applicable to a recurring design *problem* parametrized to account for different software development contexts in which that problem appears. **Our architectural pattern will be MVC.**
- *Architectural style*: is a named collection of architectural design decisions that are applicable in a given development *context*, constrain architectural design decisions that are specific to a particular system within that context, and elicit beneficial qualities in each resulting system. **Our architectural style will be client/server.**
- *Architectural style flavour*[2]: is a named collection of architectural design decisions that are applicable within a specific architectural style defining new constraints not present in the original architectural style definition. **Our architectural style flavour will be three-tier.**

2.2 Selected architectural styles

For each of the aspects defined above we will briefly describe their main characteristics and focus on the most relevant motivations that have driven our choices.

2.2.1 Architectural pattern: MVC

MVC (*Model View Controller*) is an architectural pattern which is widely used to implement application requiring a user interface (the *problem* solved by the pattern) and it prescribes a separation between:

- *model*: the part of the application that handles the logic of the application data, typically interacting with a database;
- *view*: the part of the application that handles the display of the data, typically coming from the model;
- *controller:* the part of the application that handles user interaction, typically controllers read data from a view, control user input, and send input data to the model.

The main advantage in using MVC is related to *separation of concerns*: the distinction into three components allows the re-use of the logic across applications and multiple User Interfaces can be developed without concerning the codebase. Therefore, since *myTaxiService* is a system that involves different actors, they will be able to interact with the system by means of different views (eg. taxi driver and passengers but also between mobile and web passengers) and different controllers, keeping the model centralized that constitutes largest part of business logic.

This developing strategy perfectly meets the *design and conquer* principle allowing parallel development by separated teams in charge of different parts of the application and also favours the *cohesion* within each subsystem and reduces the *coupling* among them. MVC helps also maintainability since each subsystem is rather autonomous and can be modified without affecting the other parts (typically user interface changes more often than business logic).

[1]Some authors tend to consider the phrases "architectural pattern" and "architectural style" as synonyms, but we prefer keeping them separately in order to enfathise the different level of abstraction. Our definitions are taken from [3].

[2]"Architectural style flavour" is not a term used in the literature but we decided to use it to distinguish among different specializations of the same architectural style.

2.2.2 Architectural style: Client/Server

C/S (*Client/Server*) is the most widely adopted architectural style for *distributed applications* (the *context* where the architectural style is applied) in which two *roles* are defined:

- *server*: the component (or process) that provides a function or a service to the clients;
- *client*: the component (or process) that instantiate the communication with the server and uses the function or service provided by the server.

Typically the interaction takes place through messages or remote invocations.

myTaxiService is a distributed system, since actors are typically mobile or web and interact with the system by means of their devices. Most of the relevant elaborations (eg. request storing, reservation evaluation, queue management) has to be carried out in a central point, since a global view of current scenario needed, while the information exploited to perform those elaboration is typically provided by a large number of actors (taxi drivers and passengers). Considering the fact that actors ask the system for a service and tacking into account the distributed nature of the system, C/S architectural style turns out to be a good solution. C/S style also enhances the maintainability being nowadays an established style. P2P (*Pear to pear*) style seems to be inappropriate in this context since a "well-defined" distinction between roles is defined; while *cloud computing* can be taken into consideration as an opportunity for the deployment phase.

2.2.3 Architectural style flavour: three-tier

The C/S model does not impose any constraint neither about how *logical layers* (presentation, application or business logic, data) have to be distributed among the deployment units nor about the number of *tiers* (physical deployment units) has to be designed. In fact this style does not dictate that server-hosts must have more resources than client-hosts, however according to characteristics of the context different "flavours" can be defined. We will relay on the *three tier architecture* that allows a systematic allocation of the logical layers among the tiers. In our specific case the application layer is hosted for the largest part in the middle tier however some business functionalists are also carried out by the presentation layer.

- *Tier 1* (presentation) The interaction with the user has to be dealt with by the presentation layer installed into mobile and web applications. Those application are also in charge of some simple validations of the data and have to realize the interaction with external systems (eg. GPS, GoogleMaps) therefore a part of the business logic has to be hosted here.
- *Tier 2* (application) Information has to be collected from users, further validated and processed in a centralized way (since also information related to previous events is needed) and possibly the results of the elaboration might be sent to the user. This is a pure application tier, containing the largest part of the business logic. As it will be shown later, it can be further split into the level in charge of the *visualization* (web tier) and the level in charge of the information *processing* (business tier).
- *Tier 3* (data) Data has to be stored in persistent memory devices and retrieved; this tier is devoted to the database management.

2.3 High level components and their interaction

In the previous section, navigating from the top to the bottom the different levels of abstraction in architectural design have been exposed and motivated. Now we will discuss the decomposition of the system into components and connectors starting from a high level decomposition in which the mayor components will be shown, we will use an informal graphical notation.

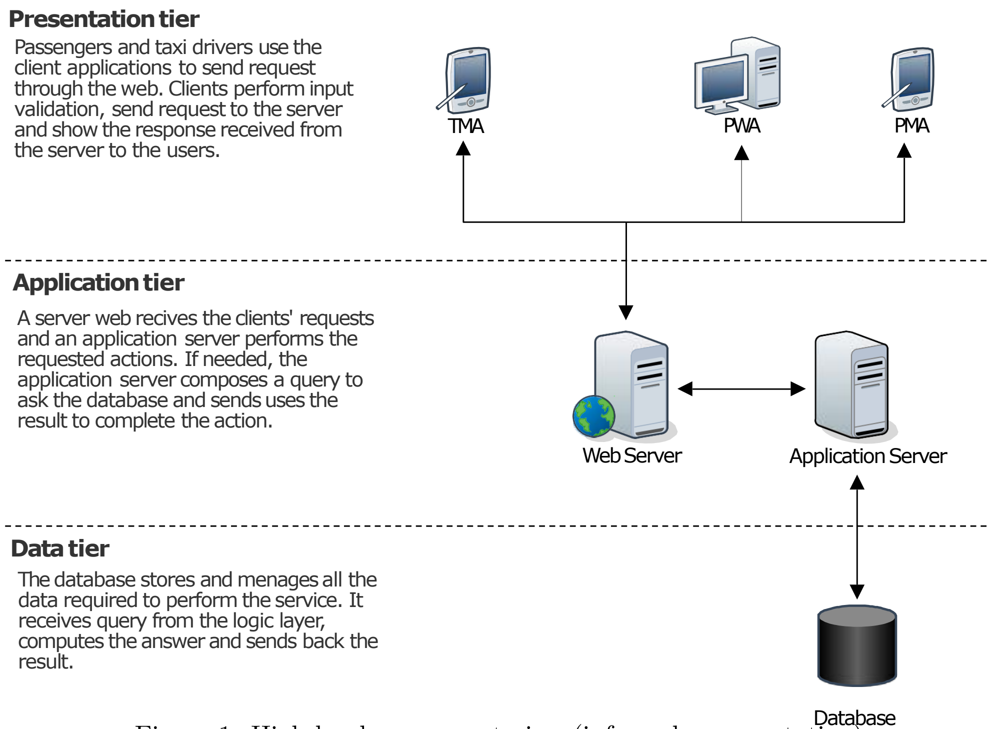


Figure 1: High level component view (informal representation)

2.3.1 Commercial architectural system brief description

Since we would like to design a modular, reliable, secure and portable system we will relay on a consolidated technology like the JEE. JEE (*Java Enterprise Edition*) is a Java specification mainly addressed to business applications with lots of users and lots of requirements; those ones are typically web applications. The platform includes facilities for implementation of network and web services, multi-tiered, scalable, reliable, and secure network applications. The main objective of JEE is to enable developers to concentrate on business logic and to neglect implementative issues related to network communication. Specific libraries to develop the mobile application for passengers and taxi drivers for the different platforms have to be adopted.

We will provide an overall description of JEE architecture with respect to our system; this must not be considered an implementation constraint but just a suggestion about the principles that have driven the design. The Java EE platform uses a distributed multitiered application model for enterprise applications: application logic is divided into components[3] according to function, which are installed on various machines depending on the tier in the multitiered Java EE environment to which the application component belongs.

Java EE applications are divided into the tiers described in the following list.

- *Client-tier*: components that run on the client machine, a Java EE client is usually of two types.
  - *Web clients*: they are composed of dynamic web pages, which are generated by web components running in the web tier and a web browser, which renders the pages received from the server. A web client is sometimes called a *thin client* since usually does not query databases, execute complex business rules or connect to legacy applications. In *myTaxiService* passengers that use the system by means of the web portal are considered web clients, also mobile users (passengers and taxi drivers) can be considered web clients since we assume to establish a communication by means of an XML message format.
  - *Application clients*: run on a client machine and provide a way for users to handle tasks that require a richer user interface than web clients. An application client typically has a customized graphical user interface and interacts directly with the business layer or with a servlet in the web tier. No direct application clients are present in *myTaxiService.*
- *Web-tier*: components that run on the Java EE server that are in charge of the visualization of output and handling the input; they can be either JSP (Java Server Pages), JSF (Java Server Faces) or Servlets. We will suggest to use JSF to develop the web portal and manage the interaction (input insertion and output visualization) being a suitable solution for system that conform to MVC pattern, while communication with mobile users should be performed by means of servlets (with XML formatted messages).
- *Business-tier*: components that run on the Java EE server devoted to the implementation of the business logic, computing and interaction with the database; it is mainly of made of components called EJB (*Enterprise Java Beans*) to manage the business logic and JPA (*Java Persistence API*) to facilitate the interaction with the database. In *myTaxiService* this is the core tier and it entirely is devoted to all logical elaborations (eg. request/reservation handling, queue management, account management).
- *Enterprise information system (EIS)-tier*: software that runs on the EIS server mainly devoted to data management. For our system it not exactly an EIS (that may also include sophisticated business functionalists, like ERP or CMR), but just a DBMS.

[3] The terminology is slightly misleading. The term “component” in this context refers to a programmatic component (like JavaBeans, JPA, ...) while in the rest of the document we use “component” with a more abstract meaning, i.e. a block of cohese functionalities.

Although a Java EE application can consist of all tiers as shown in the figure, Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the Java EE server machine, and the database or legacy machines at the back end[4]. Three-tiered applications that run in this way extend the standard two-tiered client-and-server model by placing a multithreaded application server between the client application and back-end storage.

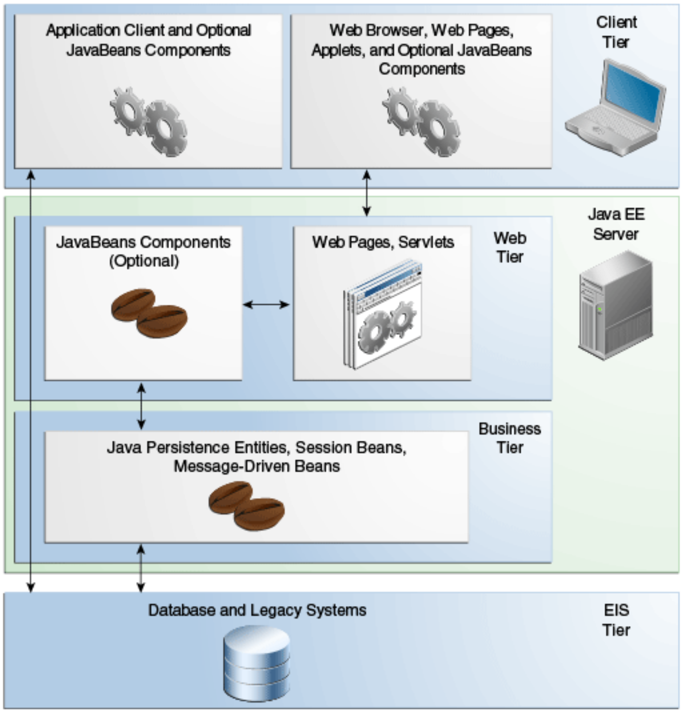


Figure 2: JEE architecture

[4]There is no consensus about the number of tiers of JEE architecture. If we consider the logical decomposition (but in this case talking of "layer" instead of tiers is more proper) we recognize 3 subsystem, while if we refer to the typical allocation of those subsystem on deployment units we clearly have 4 tiers, but this does not exclude the possibility of adopting other deployment policies.

2.4 Component view

In this section we propose a representation of the system in terms of components and connectors by means of the UML Component Diagram. First we will show a "high level" component diagram and then the most significant *subsystems* will be expanded[5].

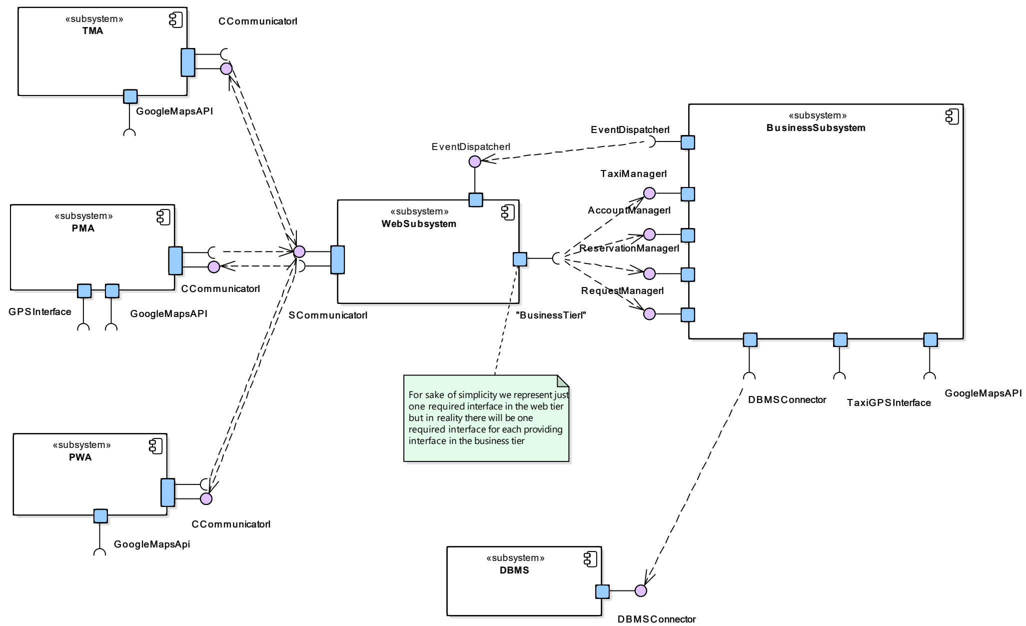


Figure 3: UML "high level" component diagram

[5] *Component* and *subsystems* are informal terms that can lead to many interpretations of different abstraction level. We adopted the following semantics: a *component* is a cohesive and little copuled group of functionalities that can be almost mapped to a programmative class (stateless component are indicated with the stereotype ≪service≫), a *subsystem* is a group of components that belong to the same "role" (eg. business logic, presentation,...) in the system.

The diagram is totally independent of the technology used to implement the system, since it is obtained by identifying the functional units in the system. A brief description of each *subsystem* is now provided.

- *TMA*: it is the subsystem in charge of all communications between the taxi and the central system. It allows taxi driver to inform about his/her availability, accept or reject requests and allows the central system to send requests and notifications to the taxi driver. It is built as a mobile application. It interfaces with GoogleMaps for the the visualization of the passenger position.
- *PMA*: it is the subsystem in charge of all communications between the mobile passenger, either registered or not, and the central system. It allows the passenger to request a taxi, visualize waiting time and number of the incoming taxi and register; it also allows registered passengers to login, reserve a taxi and modify/cancel previous reservations. It interfaces with the GPS application for position retrieval and GoogleMapsAPI for address recognition and designed for web passenger. It is built as a mobile application.
- *PWA*: it is the subsystem in charge of the same functionality of PWA but it is built as a web application and designed for web passengers. It interfaces with GoogleMapsAPI for position retrieval and address recognition.

The previous subsystems constitute the front-end of the application therefore they have to handle user interface, simple input validations, message formatting and network communications.

- *Web Subsystem*: it is the subsystem in charge of the information exchang between TMA, PMA, PWA and the Business Subsystem. It has to be able to send and receive messages in the proper format (HTML for web clients and XML for mobile clients[6]), interpret those message by means of a conversion into commands and invoke the suitable services on the Business Subsystem. It is in charge of the safe communications between the previous subsystems.
- *Business Subsystem*: it is the core subsystem in charge of all logic operations. It has to handle incoming requests and reservations, be able to correctly process them and set up the suitable consequent actions, like taxi search, taxi allocation and modifications in available taxi queues. It has also to deal with the registration and login procedures. For taxi management it has to be able to retrieve taxi position (interfacing with GPS system of each taxi), finally it interfaces with the DBMS.
- *DBMS*: it is the subsystem in charge of the persistent data management. It is accessed by the Business Subsystem to store and retrieve information.

Note that *DBMS* subsystem has not to be expanded more since the internal structure is not relevant for our application; also *PWA* subsystem will not be expanded more since the only important component is the browser; while all the other subsystem will be discussed in details in the following.

[6]We assume that all messages flowing from and to the mobile applications are codified in XML by means of a protocol that we will not discuss since it is an implementation concern as well as we will not discuss how specific web component will be chosen at implementation time (like servlets, JSF,...). Just a suggestion has been given in the previews sections.

2.4.1 TMA

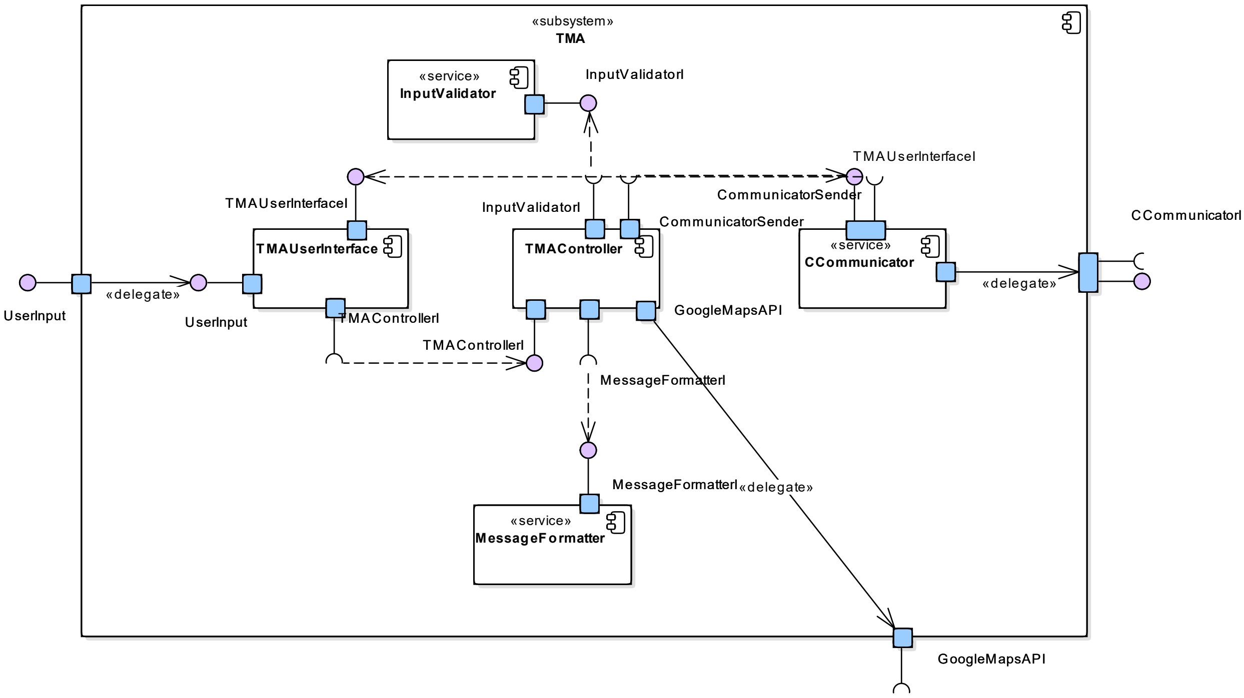


Figure 4: UML component diagram - TMA

- *TMAUserInterface*: it is in charge of showing to the taxi drivers messages coming from the central system and enable taxi driver to insert proper information when needed.
- *InputValidator*: it performs simple input validations.
- *C(lient)Communicator*: it provides the high level functions to send and receive message on the network and it manages the low level network concerns. It is also able to notify the view when a message comes and it is in charge of the secure communication.
- *MessageFormatter*: it is in charge of formatting commands into XML messages to be sent to or received from the network.
- *TMAController*: it is in charge of receiving commands from the TMAUserInterface and perform all operation needed to carry out the command (like input validation, message formatting, checking the applicability of a specific command), possibly using the connected component. It interfaces with GoogleMaps for the the visualization of the passenger position.

2.4.2 PMA

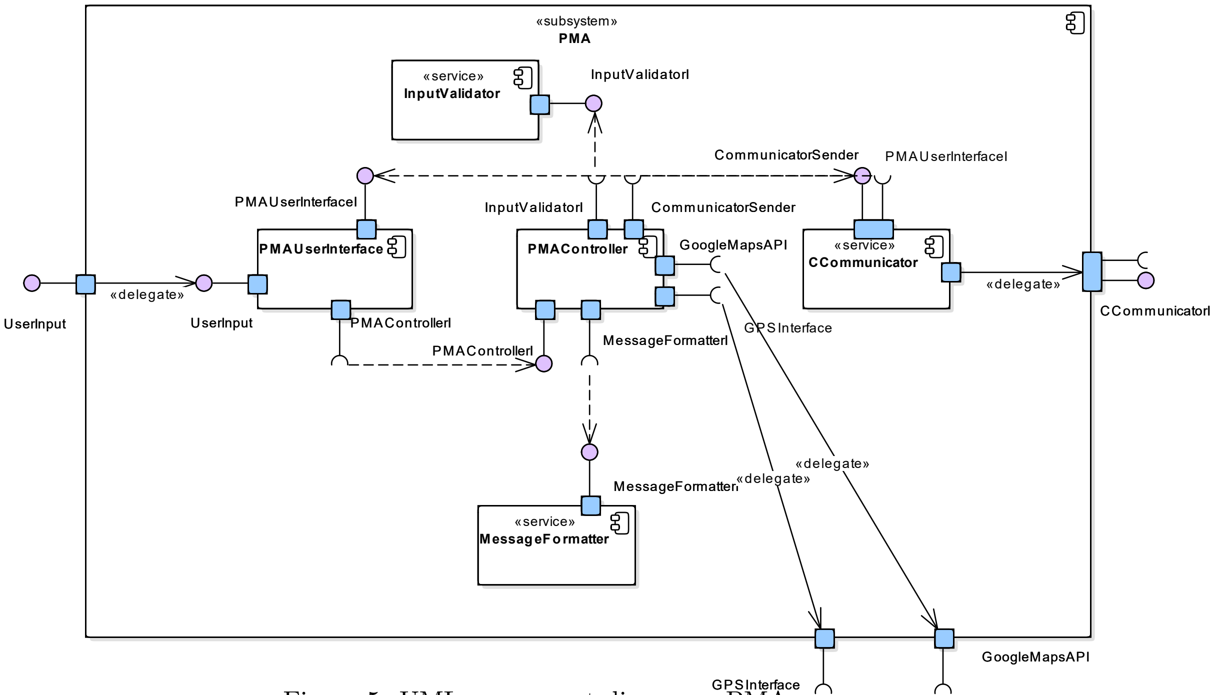


Figure 5: UML component diagram - PMA

- *PMAUserInterface*: it is in charge of showing to the passengers messages coming from the central system and enable passengers to insert proper information when needed.
- *InputValidator*: it performs simple input validations (eg. email correct format).
- *C(lient)Communicator*: it provides the high level functions to send and receive message on the network and it manages the low level network concerns. It is also able to notify the view when a message comes and it is in charge of the secure communication.
- *MessageFormatter*: it is in charge of formatting commands into XML messages to be sent to or received from the network, providing the suitable methods.
- *PMAController*: it is in charge of receiving commands from the PMAUserInterface and perform all operation needed to carry out the command (like input validation, message formatting, checking the applicability of a specific command), possibly using the connected component. It is also able to interface with GPS system to retrieve the current location and to GoogleMapsAPI for address validation.

2.4.3 WebSubsystem

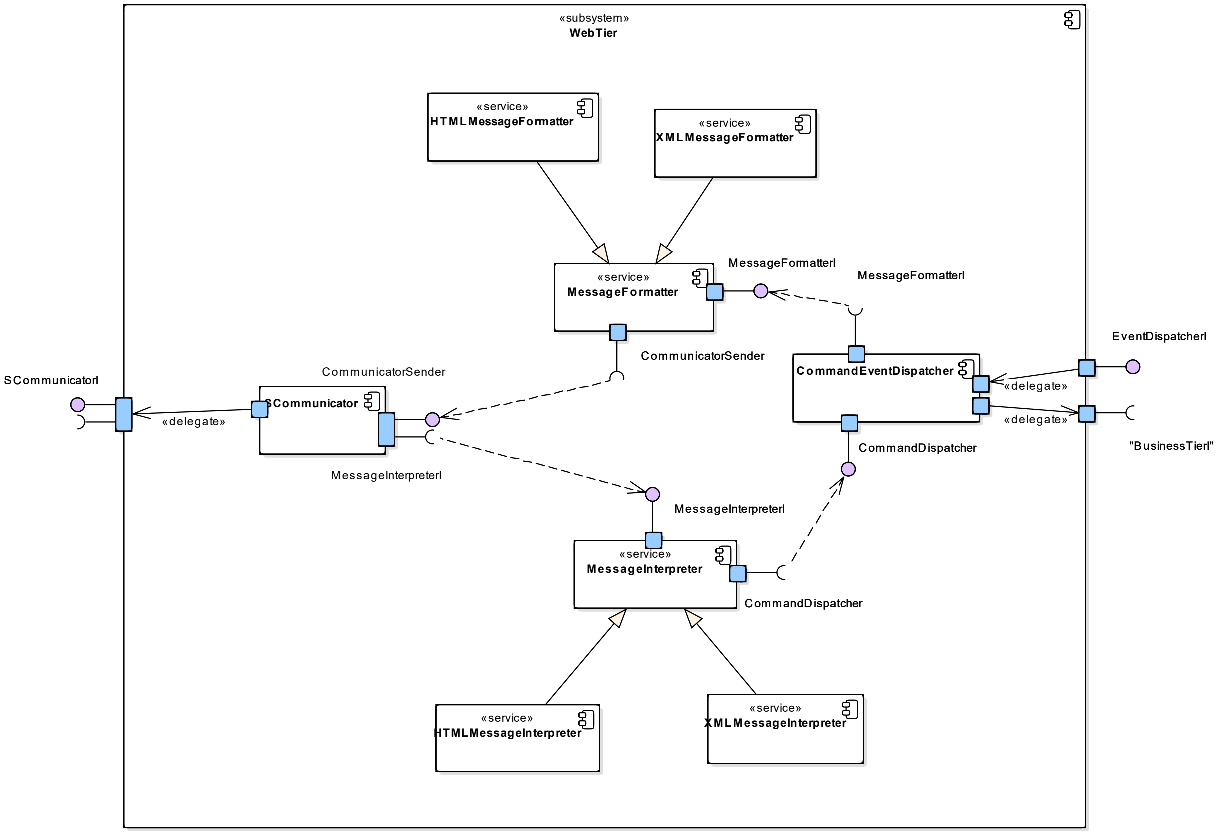


Figure 6: UML component diagram - WebSubsystem

- *MessageFormatter*: it is the component devoted to the translation of events (typically information to be displayed or commands) coming from the BusinessSubsystem into a proper format, that can be sent over the network and interpreted by clients. According to the type of client we have two different implementation of this component:
  - *HTMLMessageFormatter*: it formats an HTML page containing the information related to the event for web clients;
  - *XMLMessageFormatter*: it formats an valid XML document containing the information related to the event for mobile clients.
- *MessageInterpreter*: it is the component devoted to the reverse translation with respect to MessageFormatter, it translates messages coming from the clients into commands to be executed by the CommandEventDispatcher[7]. Symmetrically, according to the type of client we have two different implementation of this component:
  - *HTMLMessageInterpreter*: it converts HTML information (typically parameters passed by means of POST or GET) into a command;
  - *XMLMessageInterpreter*: it converts an XML valid document into a command.
- *CommandEventDispatcher*: it receives commands from the MessageInterpreter and executes them by invoking methods of the BusinessSubsystem and, in case, sends the result to the MessageFormatter by means of an event. It can also be directly invoked by the BusinessSubsystem in case the client has to be notified of an event (eg. the taxi driver has to move to another zone).
- *S(erver)Communicator*: it provides the high level functions to send and receive message over the network and it manages the low level network concerns. It is also able to notify the view when a message comes. It is also in charge of the secure communication (it manages, for instance, cryptography).

[7]Note that in PMA and TMA no MessageInterpreter is present, since the message is always XML formatted and has to be just displayed in a graphical format so no complex convertion is needed; in a way the UI represents a "simple message interpreter".

2.4.4 BusinessSubsystem

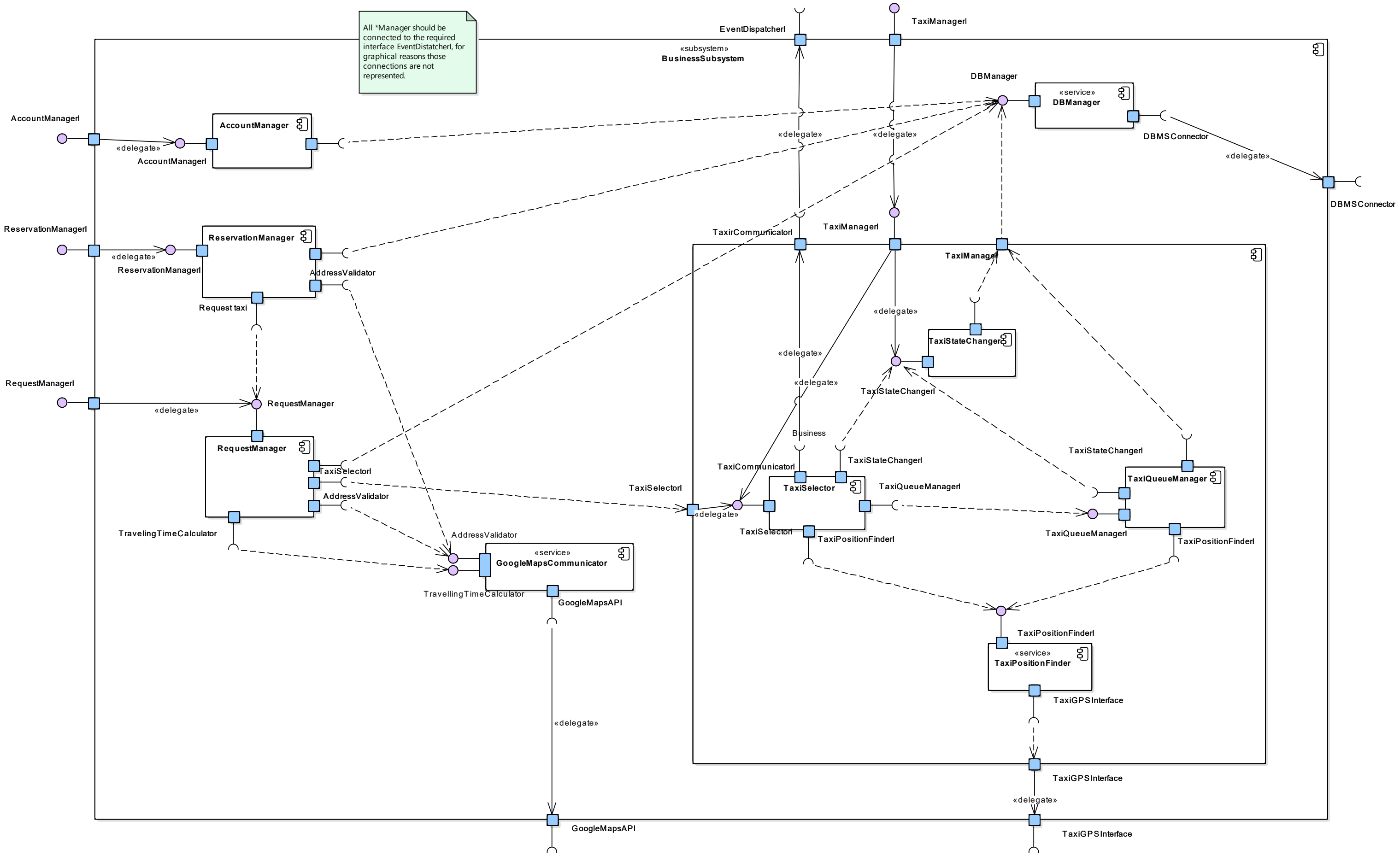


Figure 7: UML component diagram - BusinessSubsystem

- *AccountManager*: it is the component devoted to all operations related to the management of client personal information like registration, login, change of the password, logout. It is also in charge of verifying whether the data provided by passenger at registration time are valid, whether the credentials are correct at login time, possibly querying the database.
- *RequestManager*: it is the component in charge of all operations related to requests coming from the passengers. In particular it provides interfaces to forward a request and to retrieve waiting time and number of incoming taxi and manages the validation of the address (if not already done locally on the passengers' application) and the computation of the waiting time interacting with the GoogleMapsCommunicator, it invokes proper methods of the TaxiSelector in order to process the request and assign the taxi and finally it interacts with the DBManager in order to store the request and in case transform the request to a confirmed request.
- *ReservationManager*: it is the component in charge of all operations related to reservations coming from registered passengers. In particular it provides interfaces to forward, modify or cancel a reservation and it manages the semantic check of the reservation data (date, time and addresses if not already performed locally), the allocation of the request associated to the reservation interacting with RequestManager and the storage of the reservation by means of the DBManager.
- *GoogleMapsCommunicatior*: it is in charge of the interaction between the system and GoogleMapsAPIs. In particular it allows a higher level of abstraction elaborating rough data coming from the APIs and providing interfaces for waiting time calculation and address validation[8].
- *TaxiManager*: it is the component devoted to all operations related to taxi management. Since a lot of operations are possible and they can be rather complex, it is represented as a subsystem split in four components.
  - *TaxiSelector*: it is in charge of the selection of a taxi whenever a request is processed. It is interfaced with RequestManager and exploits the interfaces provided by TaxiQueueManager to look for a taxi to associate to the request and to TaxiStateChanger in order to modify the state of the selected taxi.
  - *TaxiPositionFineder*: it is the component in charge of the localization of each taxi. It manages the interface with GPS system installed on the taxi, by means of TaxiGPSInterface, and provides the interface for retrieving the position of a taxi.
  - *TaxiQueueManager*: it is the component devoted to the management of taxi queues. It provides the interface to get the current distribution of the taxis in taxi queues and it handles the operations of redistribution of taxis when needed. It interacts with TaxiStateChanger to turn the taxi state from available to moving or viceversa and with the interface TaxiCommunicatorI in order to send notification messages to taxi drivers.
  - *TaxiStateChanger*: it is the component in charge of handling the state transitions for taxi drivers. It provides the interface to change the state of a taxi and requires the interface to DBManager in order to store the new state into the database, eventually it performs transition validity checks and, in case, sends messages to the taxi application by means of TaxiCommunicatorI.
- *DBManager*: it is the component in charge of the interaction between the system and the DBMS. In particular is manages the connection, by means of JDBC, and it is able to formulates query to be executed against the database starting from the information required by other components. It provides other components with proper interfaces for querying the database and it handles the persistence of data with proper programmatic representation of the tables (eg. JPA).

[8]Notice that even though PMA and TMA use GoogleMapsAPI their interaction is rather simple (just address validation is needed) so they do not include a specific communicator service, in a sense this role is played by the controller.

2.5 Deployment view

2.5.1 Deployment choices

As it was clearly stated in the initial part of this chapter the main architectural style adopted for *myTaxiService* is the client/server one. Since our system is prone to different loads according, for instance, to the hours of the day or days in the week we think that a proper deployment solution for the back end part (web server and business server) would be *cloud computing*, considering also the recent diffusion and opportunity to have access to powerful services with limited costs. To be as general as possible and avoid further implementation constraints we prefer to rely on the IaaS (*Infrastructure as a Service*) level. Like all cloud computing services, IaaS provides access to a resource belonging to a virtualized environment, in particular IaaS concerns with *virtualized hardware* in which data storage, networking and load balancing are managed by the provider. We will now describe the main motivations that drove our choice.

- *Scalability*: myTaxiService is prone to different traffic loads according to the distribution of requests during the hours of the day and the days in the week, IaaS is very flexible providing either upwards and downwards scalability and avoiding delays in expansion of the capabilities and preventing waste of resources, typically present in an in-house deployment solution.
- *Costs*: base hardware is configured and managed by the cloud provider, therefore no acquisition, installation and maintenance cost are necessary; the cost of the cloud service is almost proportionally to the amount of resource consumed (*pay-as-you-go*), there are various contracts that allows to design a kind of customized service.
- *Security*: while logical level security is not managed by the provider (eg. authentication, cryptography) in IaaS configuration, physical security is ensured since it is typically a critical aspect for the provider. In-house security, on the other hand, is not usually an individual's or a organization's main business and, therefore, may not be as good as that offered by the IaaS cloud provider.
- *Availability*: cloud architectures are very redundant both in hardware and in configurations, so in case of fault the service would be still available. Moreover, there is no need to manage backups, many IaaS cloud providers (like Microsoft Azure) offer automatic backup procedures.

On the other hand some constraints are imposed.

- You are responsible for the versioning/upgrades of software developed.
- The maintenance and upgrades of tools, database systems and the underlying infrastructure is your responsibility.
- To enable autoscaling mechanism you have to design stateless components.

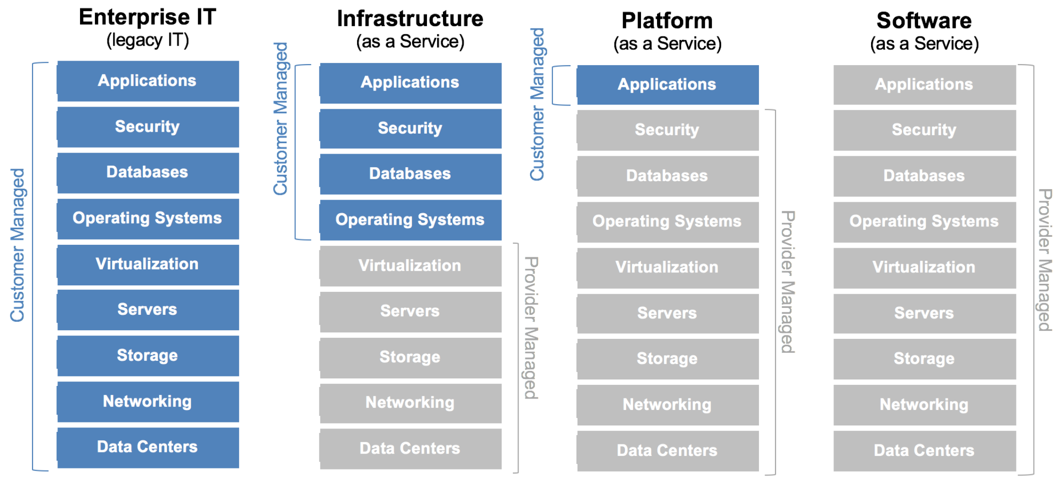


Figure 8: Cloud computing levels

2.5.2 Deployment diagram

Starting from the UML Component Diagram presented in the previous section, we derive the UML Deployment Diagram in accordance to the deployment constraints stated above. In the following diagram we represent only the subsystems, for the inner components refer to the previous diagrams.

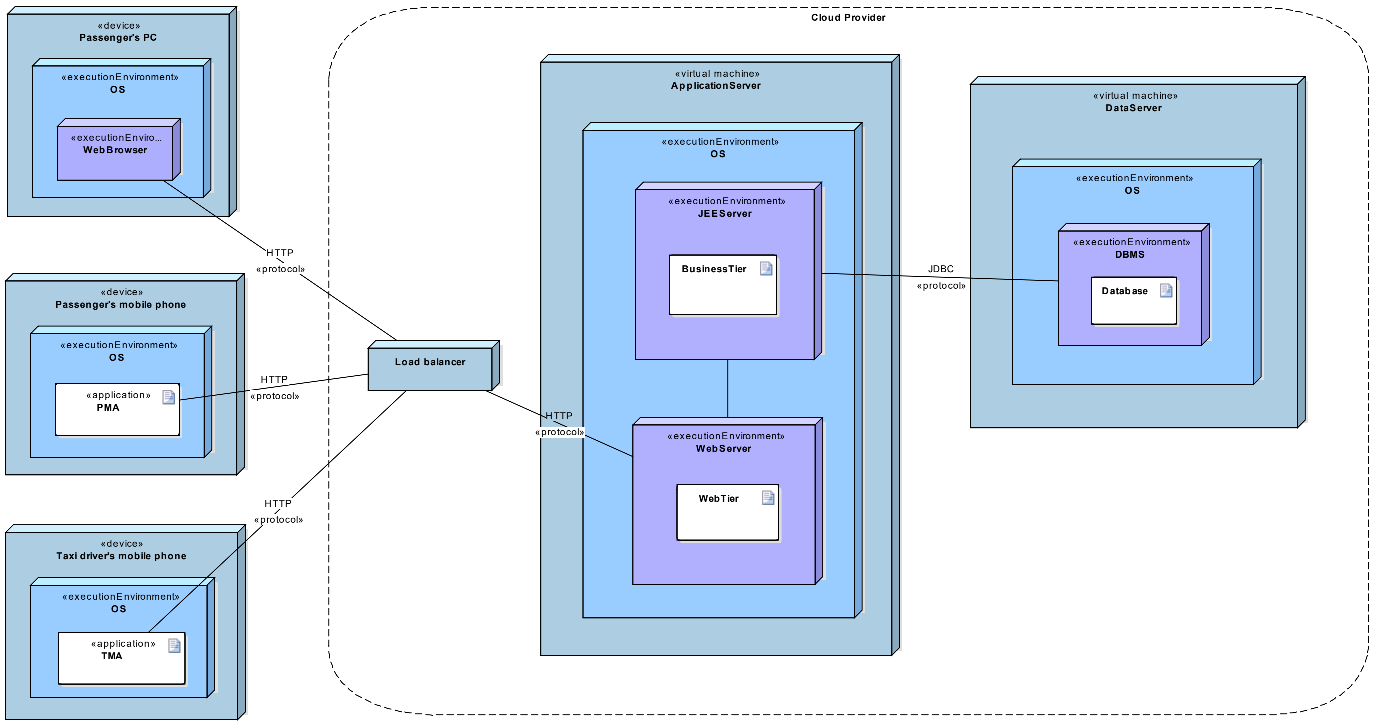


Figure 9: UML deployment diagram

For what concerns the cloud provider we actually don't know how the components will be deployed, this is up to the policy adopted by the provider; however typically a load balancer will be present and devoted to the distribution of the traffic load towards the replica of the system (we represented just one replica) each running on a virtual machine. We adopted a representation that conforms to the JEE tier architecture, this does not exclude the possibility that both DBMS, JEEServer and Web server run on the same virtual machine and also that each one run on a dedicated virtual machine.

On each node that can be either a *device* (a physical machine) or a *virtual machine*, an *execution environment* representing the operating system is running, execution environments can be nested to model for instance multiple server processes running on the system. Within the execution environment the deployment units are represented as *artifacts*. Notice that those artifacts represent the high level components depicted in the previous sections.

2.6 Runtime view

The component diagram gives just a static representation of the components, their dependencies and their interfaces; in order to better understand how those components work we propose in this section a few UML Sequence Diagrams showing the dynamical interaction between components. Note that the flow of actions represented is directly inspired from some of the use cases (see RASD), but here the level of abstraction chosen is lower, we will not cover only the main functionalists.

2.6.1 Registration

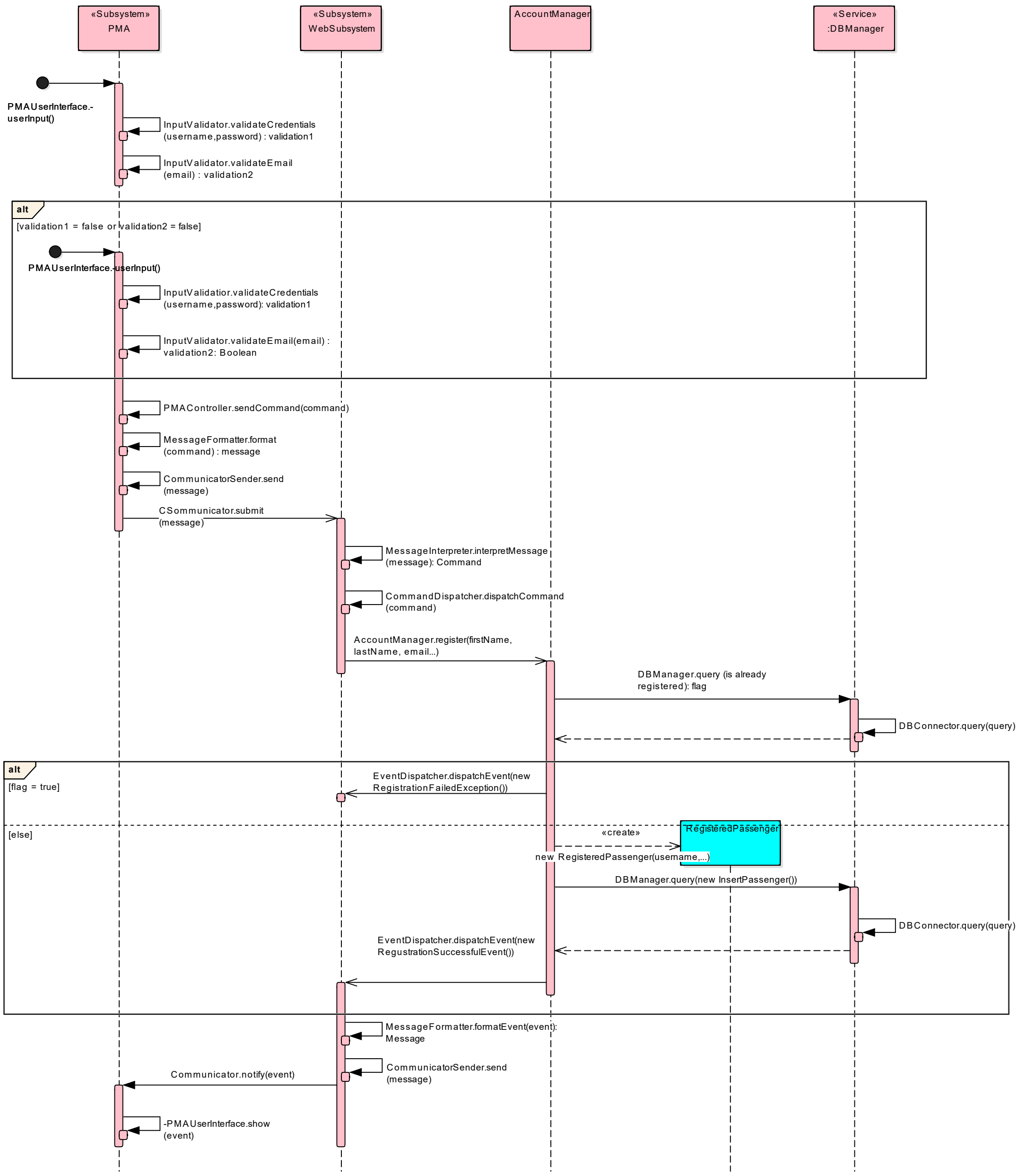


Figure 10: UML Sequence diagram - Registration

2.6.2 Login

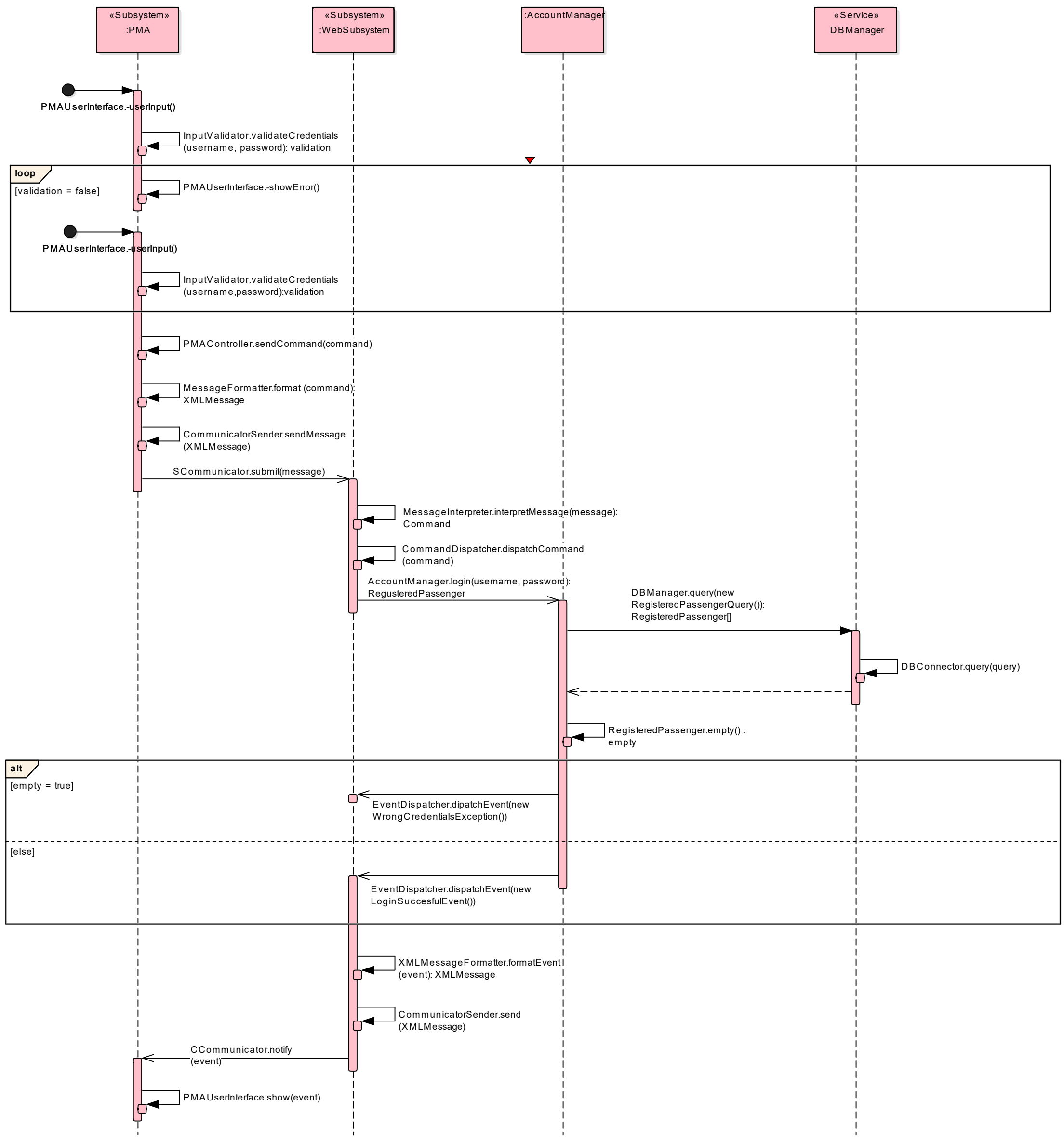


Figure 11: UML Sequence diagram - Login

2.6.3 Request using PMA

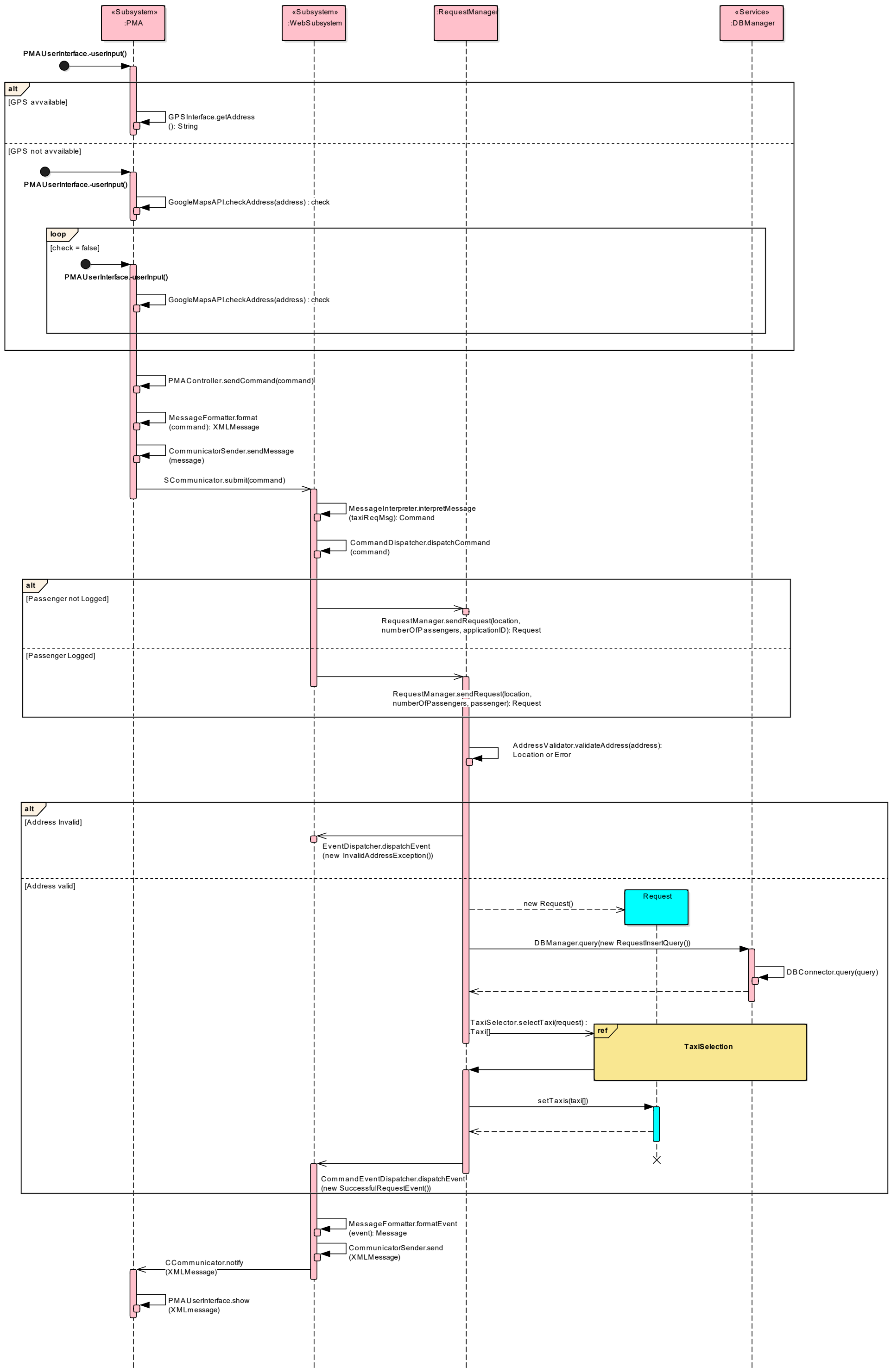


Figure 12: UML Sequence diagram - Request using PMA

2.6.4 Reservation using PWA

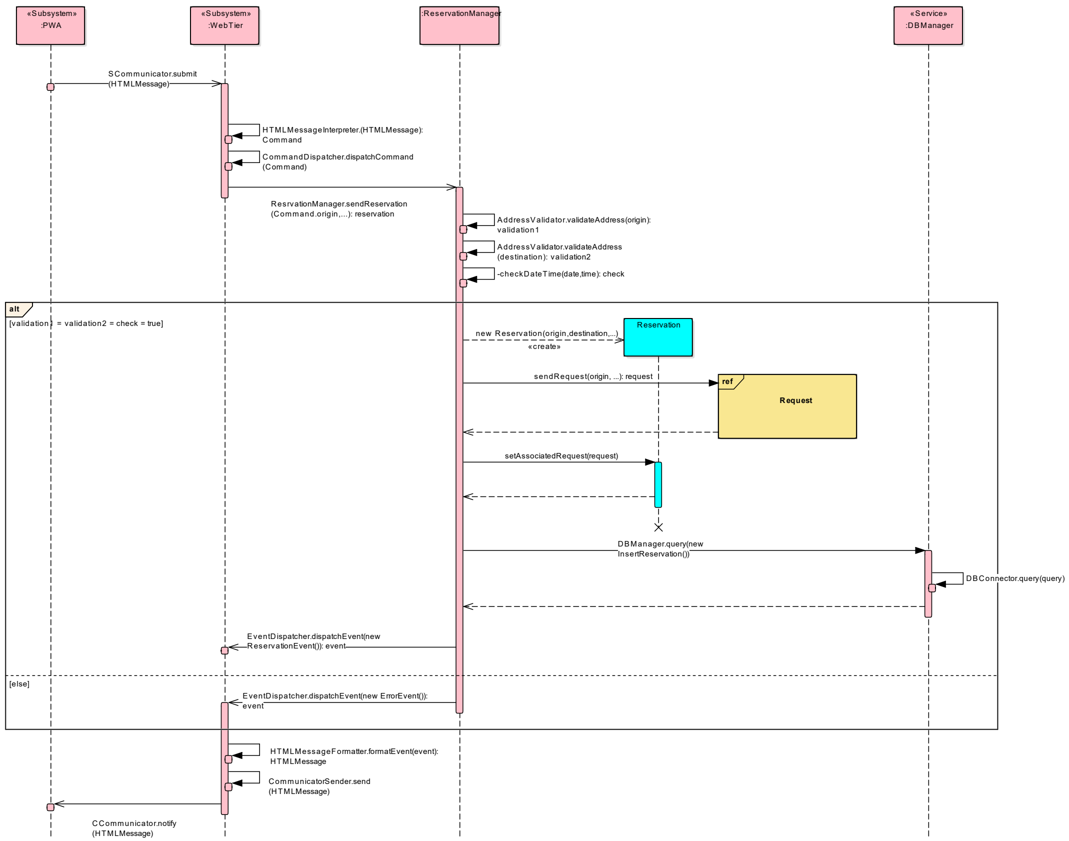


Figure 13: UML Sequence diagram - Request using PWA

2.6.5 Cancel reservation using PMA

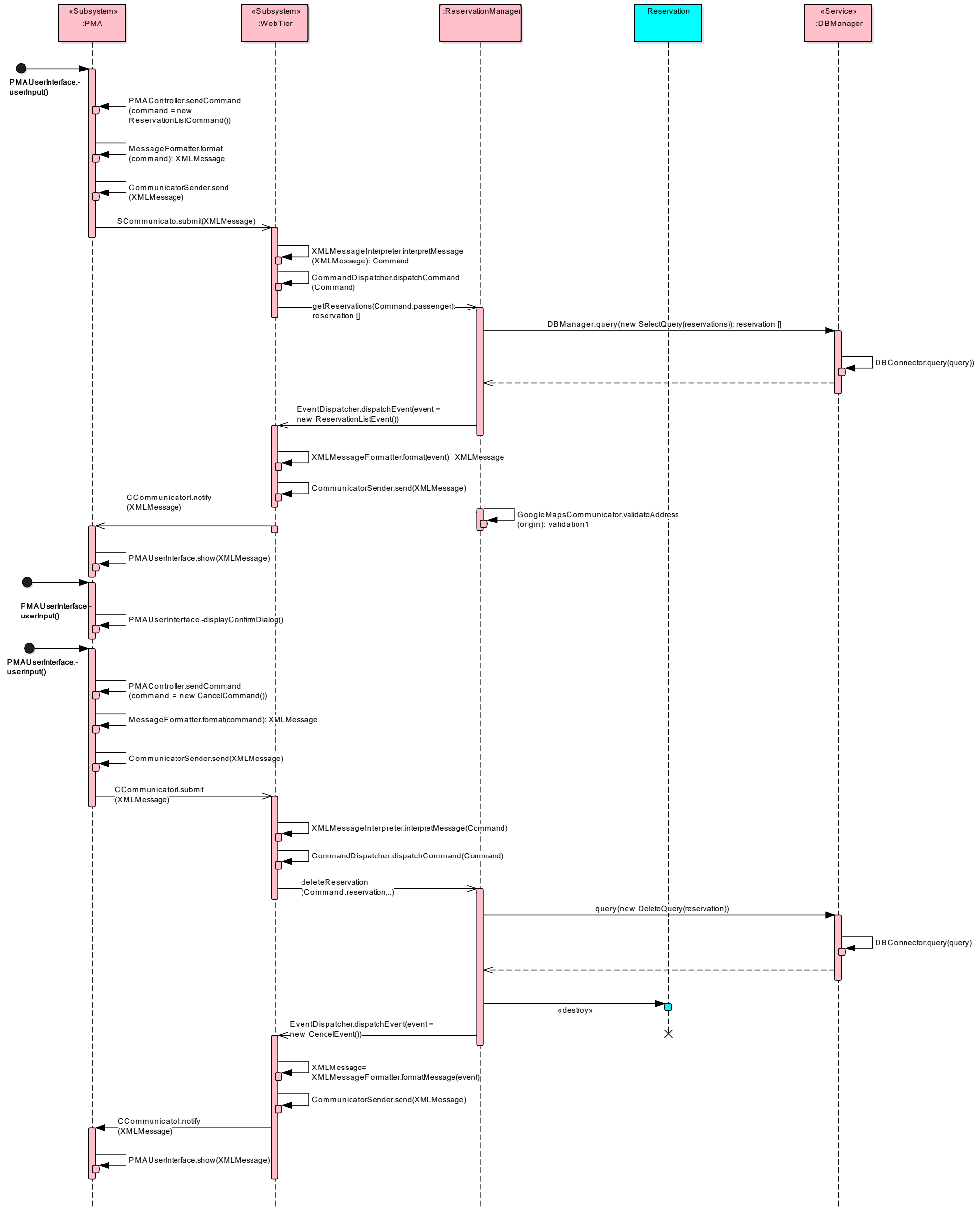


Figure 14: UML Sequence diagram - Cancel reservation using PWA

2.6.6 Taxi selection

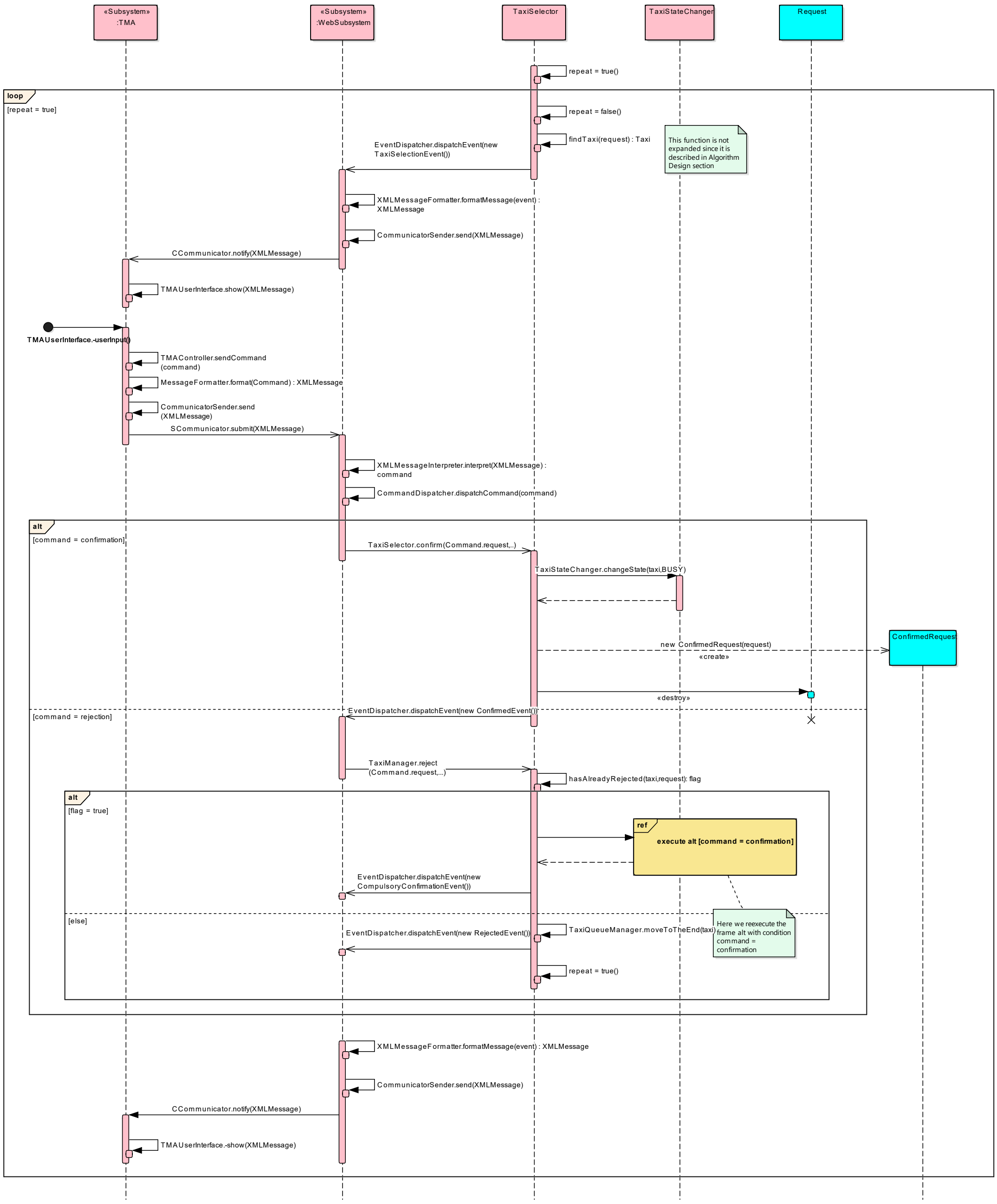


Figure 15: UML Sequence diagram - Taxi selection

2.7 Component interfaces

In this section for each component and for each provided interface we list the main methods, with corresponding parameters and types (possible thrown exception are not written). Notice that they are just the minimal methods required, many other might be added at implementation time.

2.7.1 BusinessSubsystem

Component	*Interface*	*Method*
AccountManager	AccountManagerI	`login(Username:String, Password:String) : RegisteredPassenger`
		`forgotPassword(Username:String, email:String)`
		`register(Username:String, Password:String, Firstname:String, Lastname:String, Address:String, email:String) : RegisteredPassenger`
RequestManager	RequestManagerI	`sendRequest(Location:Location, NumberOfPassengers:Integer, (RegisteredPassenger:passenger|Integer:applicationID)) : Request`
		`getWaitingTime(Request:Request) : TimeInterval`
		`getIncomingTaxiCode(Request:Request) : String`
ReservationManager	ReservationManagerI	`sendReservation(Origin:Location, Destination:Location, Date: Date, Time: Time, NumberOfPassengers:Integer, RegisteredPassenger:Passenger) : Reservation`
		`getRequest(Reservation:Reservation, RegisteredPassenger:Passenge) : Request`
		`deleteReservation(Reservation:Reservation, RegisteredPassenger:Passenger)`
		`modifyReservation(Reservation:Reservation, Origin:Location, Destination:Location, Date: Date, Time: Time, NumberOfPassengers:Integer, RegisteredPassenger:Passenger) : Reservation`
		`getReservations(RegisteredPassenger:Passenger) : Reservation[]`
GoogleMapsCommunicator	AddressValidator	`validateAddress(Addrress:String) : Location`
	TravellingTimeCalculator	`getTravellingTime(Origin: Location, Destination:Location) : TimeInterval`
DBManager	DBManagerI	`query(query : Query) : Object[]`

Component	*Interface*	*Method*
TaxiSelector	TaxiSelectorI	`selectTaxi(Request: Request) : Taxi[]`
		`confirm(taxi: Taxi, request: Request)`
		`reject(taxi: Taxi, request: Request)`
TaxiStateChanger	TaxiStateChangerI	`changeState(taxi : Taxi, TaxiState: TaxiState) : TaxiState`
		`canChange(taxi : Taxi, TaxiState: TaxiState)`
TaxiPositionFinder	TaxiPositionFinderI	`getTaxiPosition(Taxi: Taxi) : Location`
TaxiQueueManager	TaxiQueueManagerI	`getAvailableTaxis() : Taxi[]`
		`getFirst(zone : Zone) : Taxi`
		`getLast(zone : Zone) : Taxi`
		`getPosition(zone: Zone, position : Integer) : Taxi`
		`getTaxis(zone: Zone) : Taxi[]`
		`move(taxi: Taxi, oldZone: Zone, newZone: Zone)`
		`find(taxi : Taxi) : Zone`
		`getZones() : Zone[]`
		`getNumberOfAvailableTaxis() : Integer`
		`getNumberOfTaxis(zone : Zone) : Integer`
		`moveToTheEnd(taxi : Taxi)`

2.7.2 WebSubsystem

Component	*Interface*	*Method*
SCommunicator	SCommunicatorI	`submit(message : Message)`
	CommunicatorSender	`send(message : Message)`
MessageFormatter	MessageFormatterI	`formatEvent(event : Event) : Message`
MessageInterpreter	MessageInterpreterI	`interpretMessage(message : Message) : Command`
CommandEventDispatcher	CommandDispatcher	`dispatchCommand(command : Command)`
	EventDispatcher	`dispatchEvent(event : Event)`

2.7.3 PMA

Component	*Interface*	*Method*
PMAUserInterface	PMAUserInterfaceI	`show(event : XMLMessage)`
PMAController	PMAControllerI	`sendCommand(command : Command)`
MessageFormatter	MessageFormatterI	`format(command : Command) : XMLMessage`
InputValidator	InputValidatorI	`validateEmail(email : String) : boolean`
		`validateDate(date : Date) : boolean`
		`validateTime(time : Time) : boolean`
		`validateCredentials(username : String, password : String) : boolean`
CCommunicator	CommunicatorSender	`send(message : XMLMessage)`
	CCommunicatorI	`notify(event : XMLMessage)`

2.7.4 TMA

Component	*Interface*	*Method*
TMAUserInterface	TMAUserInterfaceI	`show(event : XMLMessage)`
TMAController	TMAControllerI	`sendCommand(command : Command)`
MessageFormatter	MessageFormatterI	`format(command : Command) : XMLMessage`
InputValidator	InputValidatorI	`validateEmail(email : String) : boolean`
		`validateDate(date : Date) : boolean`
		`validateTime(time : Time) : boolean`
Communicator	CommunicatorSender	`sendMessage(message : XMLMessage)`

3 Algorithm design

In this section we will show some of the most significant algorithms that should be implemented in the following phases of project. We prefer to remain abstract with respect to a specific programming language therefore the algorithms will be typically expressed in pseudocode. Notice that the following algorithms do not represent an implementation constraint but just a suggestion for the developer about the way in this phase the algorithms have been designed.

3.1 Taxi queue manager

Taxi queue manager is in charge of ensuring the “fair distribution” of taxis among the zones. The main functionalists can be depicted in the following private methods:

- *computePositions*: by means of the GPS information asked to the taxi GPS, installs the current distribution of the taxis in the queue of each zone, taxis moved into a new zone are added at the end of the queue;
- *selectTaxisToMove*: using a specific algorithm, that will be explained later, selects the number of taxis that have to be moved for each zone;
- *relocateTaxis*: the function checks if there are zones lacking of taxis and in case invokes selectTaxisToMove in order to get the number of taxis to be moved and selects those taxis among the ones in the queues.

Algorithm 1 computePosition

```
function COMPUTEPOSITION()
    taxis ← TaxiQueueManager.getAvailableTaxis()
    for all t ∈ taxis do
        location ← TaxiPositionFinder.getTaxiPosition(t)
        newZone ← location.getZone()
        oldZone ← t.getLocation().getZone()
        if zone ≠ oldZone then
            TaxiQueueManager.move(t, oldZone, newZone)
        end if
    end for
end function
```

3.1.1 Selection of the number of taxis to be moved

Before formalizing and proposing a solution to the problem of moving taxis among zones in order to satisfy the constraint of the minimum number of taxis in each zone, minimizing the number of zones traveled, we give some useful definitions.

- Z the set of zones in which the city is divided.
- N total number of available taxis at the moment.
- n_i number of requests per minute in the zone i[9].
- t_i suitable number of available taxis in the zone i.
- $t_{i,min}(t_{i,max})$ minimum (maximum) acceptable number of available taxis in the zone i.

[9] We assume this datum to be available from previous analysis; if not it can be estimated after a certain time of activity of the system. See RASD, section 2.5

- q_i actual number of available taxis in the zone i.

We would like to distribute taxis among zones proportionally to the number of requests per minute.

$$t_i = \frac{n_i}{\sum_i n_i} N$$

and we accept a tolerance of 30% so $t_{i,max} = 1.3t_i$, $t_{i,min} = 0.7t_i$.

Our algorithm should be able to ensure that, after its execution, $t_{i,min} \leq q_i \leq t_{i,max}$. Note that the most important condition is that $q_i \geq t_{i,min}$ to satisfy *demand* of the taxis, the second constraint, that is $q_i \leq t_{i,max}$ is useful to ensure the *balancing* of the taxis. In the next section we provide a formalization of the problem as a linear programming model and in the following an algorithm to solve it.

3.1.2 Linear formalization of the problem in section 3.1.1

The zones can be naturally represented as an undirected graph $G = (Z, A)$.

$$A = \{(i, j)|i, j \in Z, zone\ i\ adjacent\ to\ zone\ j\}$$

Note that, since the graph is undirected, the adjacency relation is symmetric so if $(i, j) \in A$ also $(j, i) \in A$. The graph G is unweighted, since we are interested in minimizing the number of zones traveled (and not directly the distance in km!). Starting from G we can easily compute its transitive closure[10] (we assume that G is connected for obvious reasons), registering in the meanwhile the distance (in terms of number of zones traveled) between each pair of zones. This can be done, for instance, iterating BFS for each source node (with a complexity of $O(|Z||A|)$) the output is a weighted graph $G^+ = (Z, A^+)$ where the weights are d_{ij}, i.e. the length of the shortest path (from now on called distance) between node i and node j. Let's partition the zones into two categories $\{Z_+, Z_-\}$ where Z_+ contains the zones s.t. $q_i \geq t_{i,min}$ and Z_- contains all zones s.t. $q_i < t_{i,min}$, so Z_- contains the zones lacking of taxis. Notice that we are only interested in the arcs of G like (i, j) where $i \in Z_+$ and $j \in Z_-$ because taxis have to be moved from a zone having more than needed taxis to a zone in which some taxis are needed, so we assume to erase the others and we obtain a bi-partied directed graph.

Let's call x_{ij} the number of taxis moved from zone $i \in Z_+$ to zone $j \in Z_-$ (decision variable). Referring to the previous notation, we can easily define the objective function:

$$\min \sum_{i \in Z_+} \sum_{j \in Z_-} d_{ij} x_{ij}$$

subject to the following constraints:

- $x_{ij} \geq 0\ \forall i \in Z_+, j \in Z_-$ (non negativity constraint);
- $q_i - \sum_{j \in Z_-} x_{ij} \geq t_{i,min}\ \forall i \in Z_+$ (availability constraint);
- $q_j + \sum_{i \in Z_+} x_{ij} \geq t_{j,min}\ \forall j \in Z_-$ (demand constraint);
- x_{ij} integer $\forall i \in Z_+$, $\forall j \in Z_-$.

[10]Just transitive, not reflexive because we are not interested in paths from one node to itself.

It's not difficult to recognize in that formulation a *minimum cost flow* problem[11] in a network $G' = (Z \cup \{s,t\}, A')$ where $A' = \{(i,j) \in A^+ | i \in Z_+ j \in Z_-\} \cup \{(s,i) | i \in Z_+\} \cup \{(j,t) | j \in Z_-\}$. Notice that we have added a source and a sink suitably connected to the other nodes. The costs are given by the distances and we assume that $d_{si} = d_{jt} = 0$ and the capacities are $k_{si} = q_i - t_{i,min}$,$k_{ij} = q_i - t_{i,min}$[12] and $k_{jt} = t_{j,min} - q_j$ finally we look for a flow of value sufficient to fulfill the demand constraint, i.e. $\phi = \sum_{j \in Z_-} k_{jt}$.

In addition to the classic formulation we may also impose a constraint to ensure that each zone has a number of taxis that is $\leq t_{i,max}$ but we have already observed this is less important with respect to the demand constraint, therefore we will not consider it.

3.1.3 Minimum cost flow algorithm for problem 3.1.1

The algorithm we present, known as *negative cycle elimination algorithm,* starts with a feasible flow **x** that can be found by means of a maximum flow algorithm like Edmonds-Karp (the complexity of the algorithm is [13]$O(|Z||A'|) = O(|Z|^3)$). Starting from graph G' and an initial feasible flow **x** we can build the residual (or incremental network) $\overline{G} = (Z \cup \{s,t\}, \overline{A})$ where we add an arc $(i,j) \in \overline{A}$ whenever there is:

- a non saturated arc $(i,j) \in A'$ with residual capacity $\overline{k_{ij}} = k_{ij} - x_{ij}$ and residual cost $\overline{d_{ij}} = d_{ij}$,
- a non empty arc $(j,i) \in A'$ with residual capacity $\overline{k_{ij}} = x_{ij}$and residual cost $\overline{d_{ij}} = -d_{ij}$.

A path from s to t in the residual network corresponds to a new feasible flow. Since we start with a maximum feasible flow (and by construction we cannot reduce it) the only way of modifying the solution without violating the flow conservation constraints is to vary the flow on a cycle (or on a set of cycles) by an identical quantity δ. In fact, a flow variation on a route which does not close into itself would immediately bring to a violation of the flow conservation constraints which were satisfied by the original flow x. Considered a cycle $C \in A'$, the maximum practicable flow variation is given by:

$$\delta = \min_{(i,j) \in C} \{\overline{k_{ij}}\}$$

The flow is updated according to the following rules:

$$x'_{ij} = \begin{cases} x_{ij} + \delta & if (i,j) \in A' \cap C \\ x_{ij} - \delta & if (j,i) \in A' \cap C \\ x_{ij} & otherwise \end{cases}$$

The variation of the total cost is given by

$$\delta \sum_{(i,j) \in C} \overline{d_{ij}}$$

This means that there is an improvement in the solution value only if cycle C has sum of negative residual costs. It can be proved that a flow x is a minimum cost flow if and only if there exist no negative cycles in the residual graph. Here is the pseudocode.

[11]The minimum cost flow problem consists in determining the flow on the arcs of the network so that all available flow leaves from sources, all required flow arrives at origins, arc capacities are not exceeded and the global cost of the flow on arcs is minimized.

[12]There is no limitation in the number of taxis that can be sent on arc (i,j), we choose the capacity as the maximum number of taxis that can be sent from node i (we could also choose the capacity as the number of taxis that zone j must recieve).

[13]Notice that in the network G' the number of nodes is $|Z| + 2$ and the number of arcs is $|Z| + |Z_+||Z_-|$.

Algorithm 2 selectTaxisToMove

```
function selectTaxisToMove(G', K, D):(X, cost)              ▷ Minimum cost flow

                                          ▷ Find an initial maximum feasible flow
    X ← Edmonds-Karp(G', K)
    cost ← 0
    for all (i, j) ∈ G'.A' do
        cost ← cost + x_ij d_ij
    end for

                                       ▷ Until there is a negative cost cycle...
    optimal ← false
    while not optimal do
                                           ▷ ...compute the residual network...
        (G̅, D̅, K̅) ← residualNetwork(G, K, D, X)
                                                     ▷ ...find a negatite cycle...
        C ← findNegativeCycle(G̅, D̅, K̅)
        if C = {} then
            optimal ← true
        else
                                                           ▷ ...update the flow
            δ ← min_{(i,j)∈C} k̅_ij
            for all (i, j) ∈ G'.A' do
                if (i, j) ∈ C then
                    x_ij ← x_ij + δ
                    cost ← cost + δ d̅_ij
                end if
                if (j, i) ∈ C then
                    x_ij ← x_ij − δ
                    cost ← cost + δ d̅_ij
                end if
            end for
        end if
    end while
    return (X, cost)
end function
```

The algorithm is based on searching for negative cost cycles but no criterion has been specified to select one negative cost cycle when more than one are present, this has a negative impact on the complexity. Let's call $k_{max} = \max_{(i,j)\in A'}\{k_{ij}\}$ and $d_{max} = \max_{(i,j)\in A'}\{d_{ij}\}$, since the choice of the initial feasible solution is not in the least dictated by cost criteria, at the beginning in the worst case the cost (i.e. the total number of zones traveled by all taxis) of the provided solution is given at most by $|A'|k_{max}d_{max}$, we may suppose as well that the optimal solution in the extreme case has cost 0. Observing that at each iteration of the algorithm the flow varies by at least one unit and that this induces a decrease in the value of the objective function by at least one unit, we will have to perform, in the worst case, $O(|A'|k_{max}d_{max})$ iterations. Knowing that negative cycles can be recognized in $O(|Z||A'|)$, the algorithm's complexity is $O(|Z||A'|^2 k_{max}d_{max})$, which is not polynomial[14]. It can be demonstrated that, in case of a particular choice of the negative cycle, the algorithm is polynomial. For instance if we implement the function $findNegativeCycle$ with the *minimum mean-cost cycle canceling* algorithm we achieve a polynomial computational complexity of $O(|Z|^2|A'|^2 \lg(||Z|d_{max}))$[15].

[14] The number of bits needed to store k_{max} (or d_{max}) is proportional to the $\lg k_{max}$, so the complexity is exponential with respect to the size of the instance.

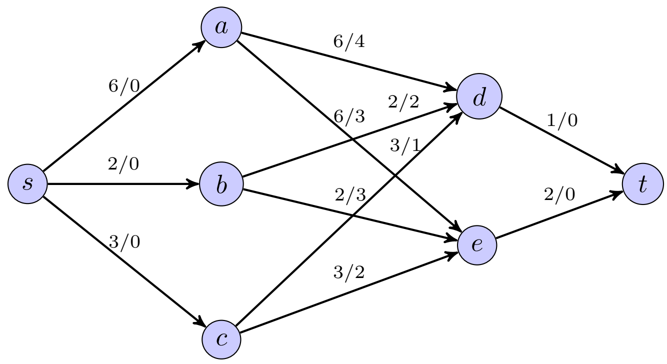
[15] From R. K. Ahuja, T. L. Magnati, J. B. Orlin, Network Flows, 1993

3.1.3.1 Some observations

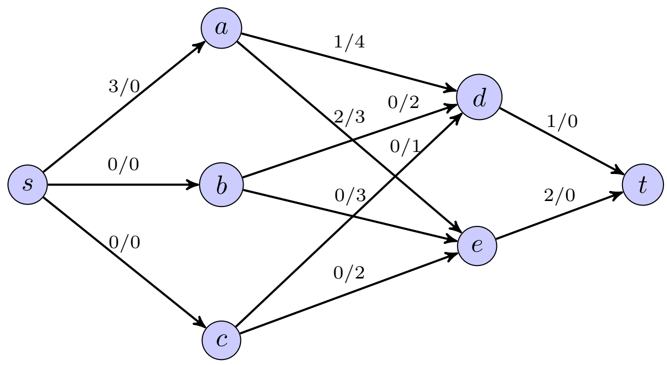
- The algorithm returns the **number** of taxis to be moved from each zone to each zone, not **which** taxis to move. We will assume to move the taxis that are located at the tail of the queue.
- We should require that the algorithm ensures a final value of q_i strictly grater than $t_{i,min}$ to avoid too many invocations of the procedure (because if we ensure to have exactly $t_{i,min}$ taxis in each zone, right after one taxi goes into a non available state the procedure must be rexecuted), this can be done by considering a new value $t^{'}_{i,min} = 1.1t_{i,min}$ that is used only define the number of taxis to be ensured in each zone and **not** to decide when to execute the procedure.

Algorithm 3 relocateTaxis

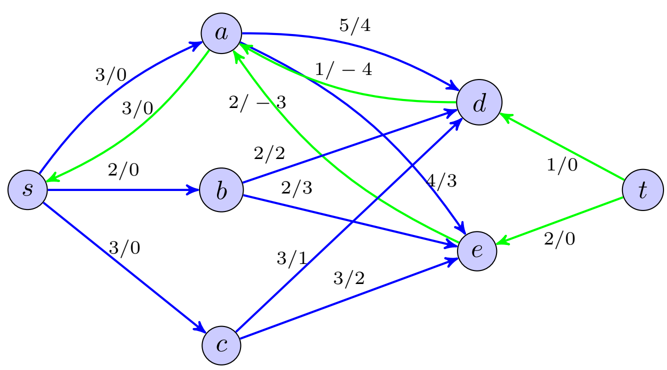
```
function relocateTaxis
    Z ← TaxiQueueManager.getZones()
    numberOfAvailableTaxis ← TaxiQueueManager.getNumberOfAvailableTaxis()

                                              ▷ Compute the closure of the graph
    (G+, D) ← graphTransitiveClosure(Z)

                                              ▷ Build the network G'
    G' ← (Z ⋃ {s, t}, {})
    for all (i, j) ∈ G+.A+ do
        if not i.isLackingOfTaxis(numberOfAvailableTaxis) and j.isLackingOfTaxis(numberOfAvailableTaxis) then
            G'.A' ← G'.A' + {(i, j)}
            k_ij ← TaxiQueueManager.getNumberOfTaxis(i) − t_i,min
        end if
    end for

    for all i ∈ G'.N − {s, t} do
        if not i.isLackingOfTaxis(numberOfAvailableTaxis) then
            G'.A' ← G'.A + {(s, i)}
            k_si ← TaxiQueueManager.getNumberOfTaxis(i) − t_i,min
            d_si ← 0
        else
            G'.A' ← G'.A + {(i, t)}
            k_it ← t_i,min − TaxiQueueManager.getNumberOfTaxis(i)
            d_it ← 0
        end if
    end for

    X ← selectTaxisToMove(G', K, D)
                                              ▷ Send notification to taxis
    for all x_ij > 0 do
        for all 1 <= n <= x_ij do
            taxi ← TaxiQueueManager.getLast(i)
            TMASendNotification(taxi, j)
        end for
    end for
end function
```


a
b
c
d
e
s
t
6/0
2/0
3/0
6/4
6/3
2/2
3/1
2/3
3/2
1/0
2/0

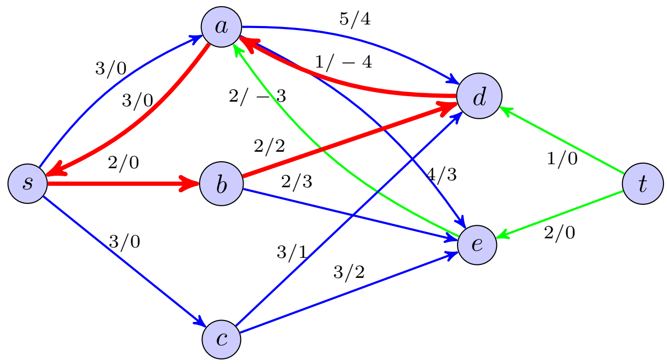
(a)


a
b
c
d
e
s
t
3/0
0/0
0/0
1/4
2/3
0/2
0/1
0/3
0/2
1/0
2/0

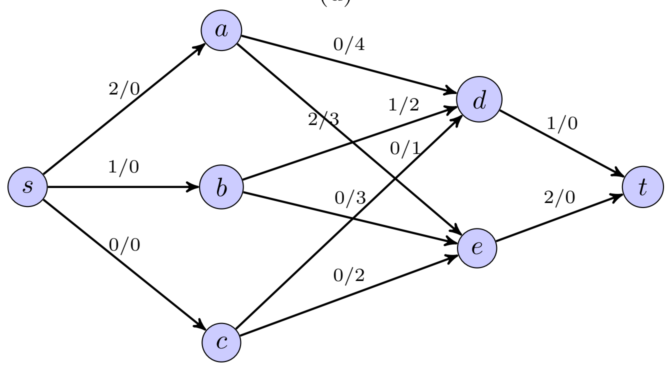
(b)


a
b
c
d
e
s
t
3/0
3/0
2/0
3/0
5/4
1/ − 4
2/ − 3
2/2
2/3
4/3
3/1
3/2
1/0
2/0

(c)



(d)



(e)

Figure 16: An iteration of the minumum cost flow algorithm.
(a) The initial network, arcs are labeled with k_{ij}/d_{ij}. (b) A maximum feasible flow of cost 10 found with Edmonds-Karp, arcs are labeled with x_{ij}/d_{ij}. (c) The incremetal network associated to the flow at point b, arcs are labeled $\overline{k_{ij}}/\overline{d_{ij}}$. (d) A negative cost cycle is found: s - b - d - a - s, $\delta = \min\{2, 4, 1, 3\} = 1$. (e) The new feasible flow obtained applying the negative cost cycle algorithm of cost 8.

3.2 Taxi selector

Taxi selector is the component in charge of searching for a taxi whenever a request comes. The function *findTaxi* looks for the first taxi available according to the policies defined in the RASD document, sends the request to the taxi driver and, in case no taxi are available at all, to puts the request on hold.

Algorithm 4 findTaxi

```
function findTaxi(request)
    visitedZones ← [false]
    zone ← request.getLocation().getZone()
    if TaxiQueueManager.getNumberOfTaxis(zone) = 0 then
        visitedZones[zone] ← (true)
        adjZones ← zone.getAdjacentZones()
        found ← false

        while not found and not adjZones.isEmpty() do
            z ← adjZones.pop()
            visitedZones[zone] ← (true)

            if TaxiQueueManager.getNumberOfTaxis(z) = 0 then
                for all h ∈ z.getAdjacentZones() s.t. not visitedZones[h] do
                    adjZones.add(h)
                end for
            else
                zone ← z
                found ← true
            end if

        end while

    end if

    if not found then
        putOnHold(request)
    else
        taxi ← TaxiQueueManager.getFirst(zone)
        TMASendRequest(taxi, request)
        waitForAnswer(1 minute)
    end if
end function
```

4 User interface design

Look and feel plays an vital role in the commercial success of every application; therefore at design time close attention should be payed in planning its structure. *User-friendliness* is an important feature that any UI should fulfill, it can be decomposed in many characteristics: *navigability* (links between pages are to be designed to make the transition between pages easy, depth of levels of navigation and number of links shouldn't be too many), *accessibility* (the information should be available for any browser), *usability* (user should be able to master the application without technical expertise) and *readability* (information should be presented in an adequate format and in balanced amount).

For *myTaxiService* the minimal requirements are those that were stated in the RASD, in particular:

- TS is meant for user without any particular knowledge or experience in the field of IT so the application must be intuitive and easy to master (usability and readability).
- Every functionality shall be reached surfing no more than 4 pages (navigability).

This section will be structured in two parts: in the first one new mockups will be provided and in the second UX diagrams will be shown.

4.1 User interface mockups

Starting from the mockups provided in the RASD we will extend the functionalists covered providing new representations, however we do not intend to show all possible pages. As usual, they are not constraining the implementation.

4.1.1 PWA

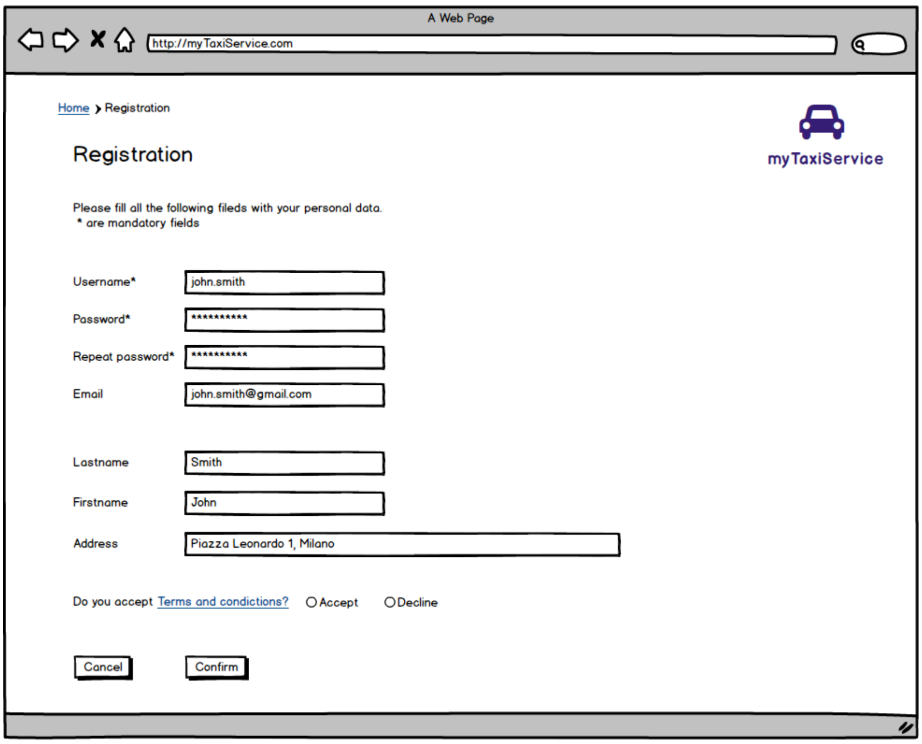


Figure 17: Registration, initial page, web application

Figure 18: Request, page one, web application

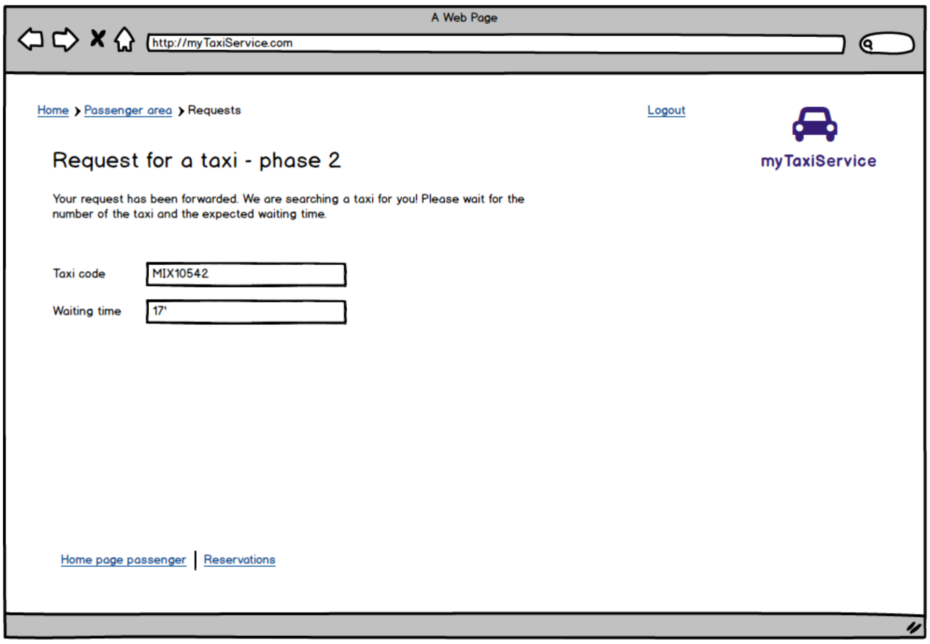


Figure 19: Request, page two, web application

4.1.2 PMA

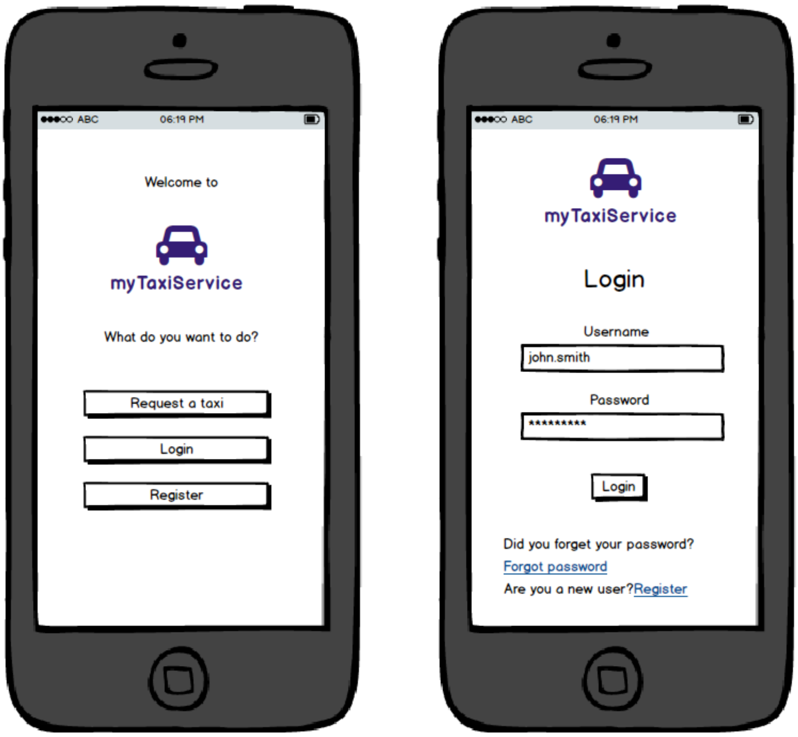


Figure 20: Initial page, mobile application - Login, mobile application

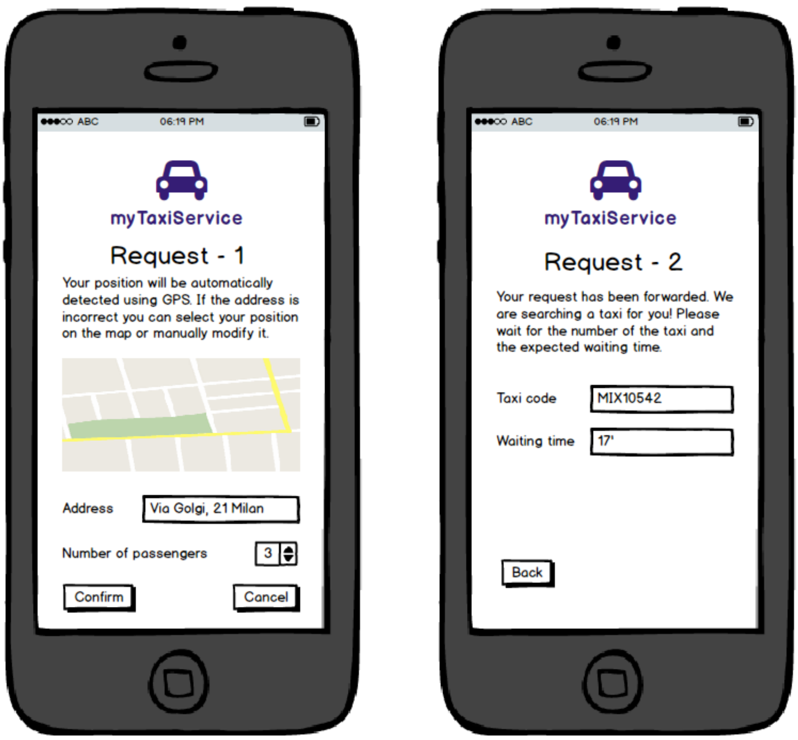


Figure 21: Request, page one and two, mobile application

Figure 22: Reservation, page one and two, mobile application

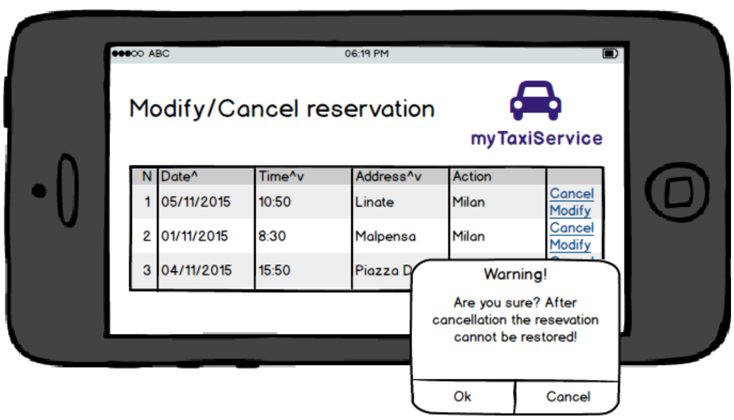


Figure 23: Cancellation confirmation, mobile application

4.1.3 TMA

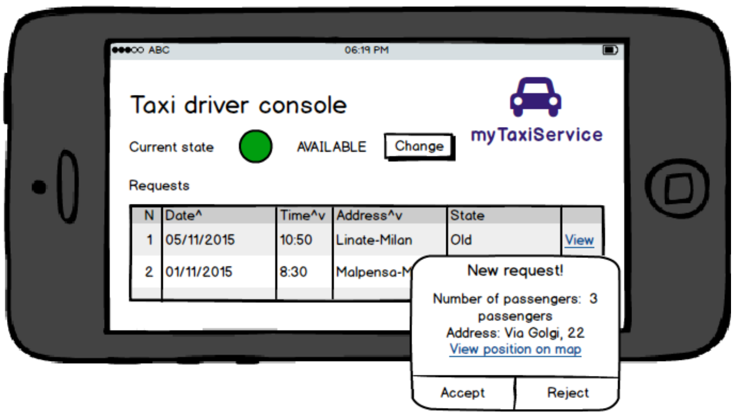


Figure 24: An incoming request

Figure 25: Taxi driver sees the position of the passenger

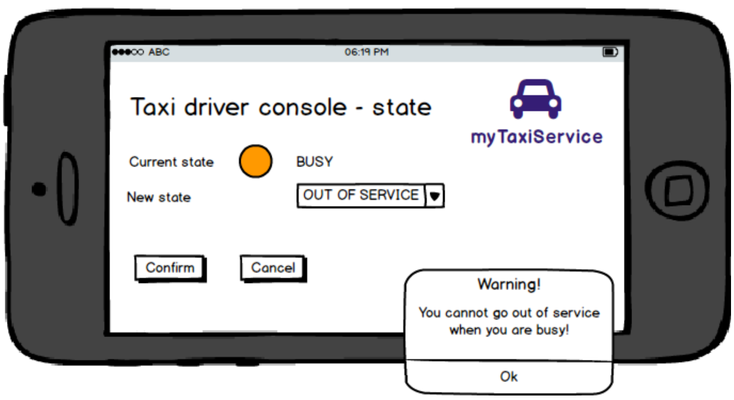


Figure 26: Taxi driver tries go out of service when busy

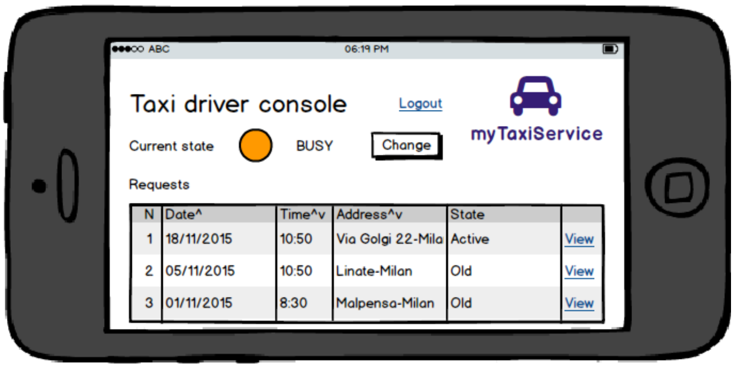


Figure 27: Taxi carrying out a request

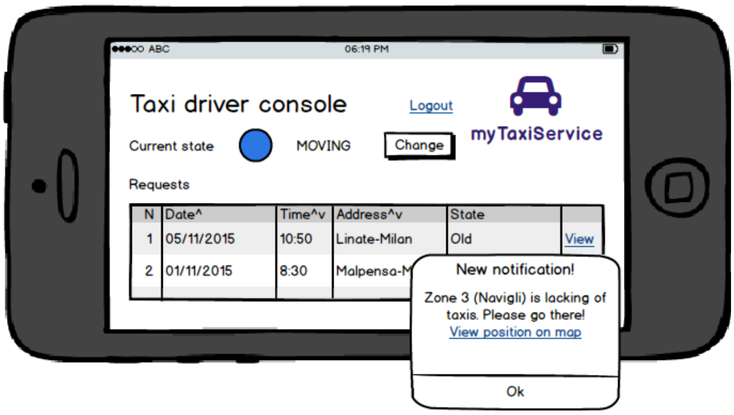


Figure 28: A move notification incoming

4.2 UX diagrams

The UX diagram (*UsereXperience diagram*) shows the organization of the screens and the user can navigate among them. In this section we will provide the diagrams for the unregistered passenger, the registered passenger and the taxi driver

4.2.1 Unregistered passenger

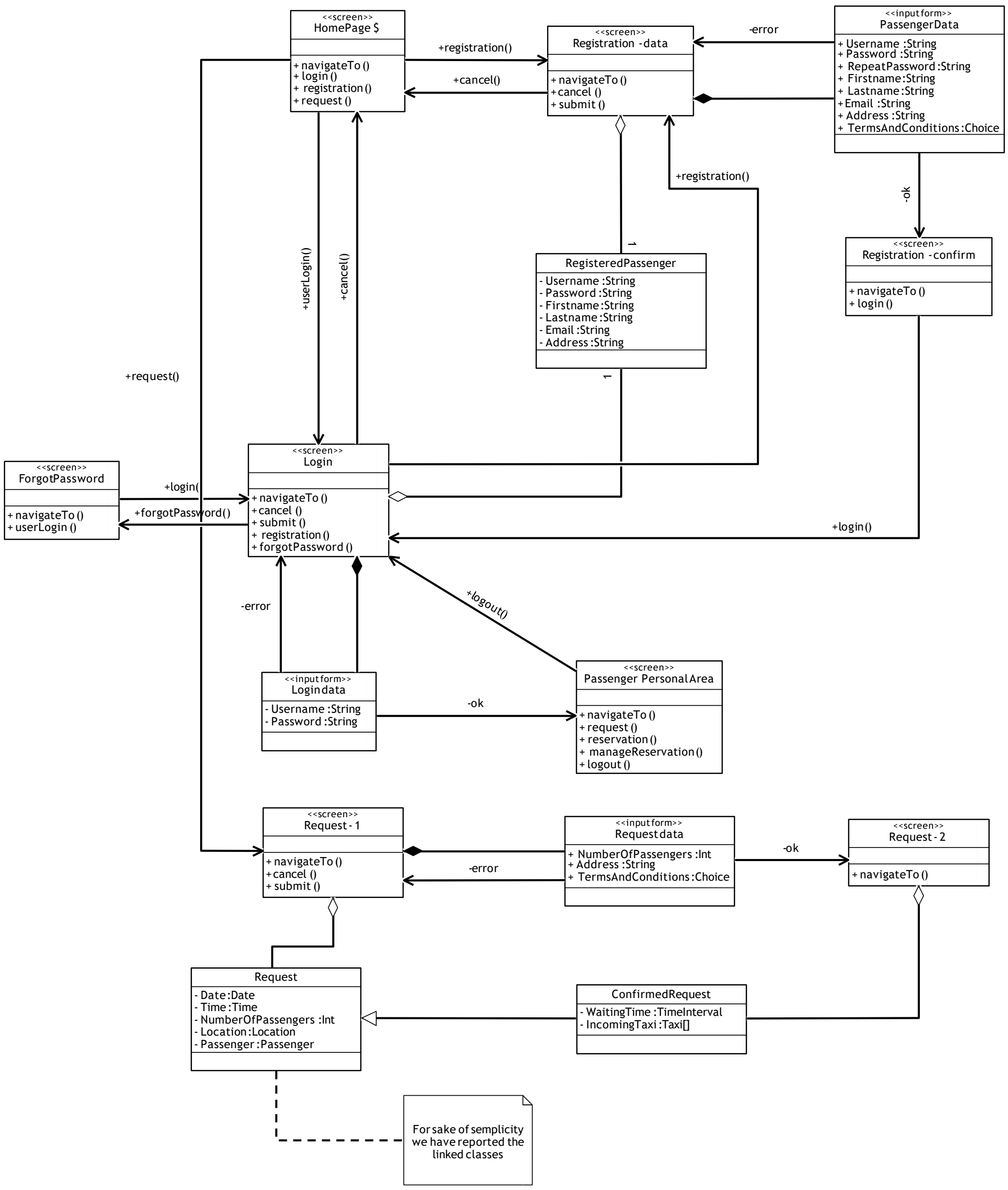


Figure 29: UX diagram - unregistered passenger

4.2.2 Registered passenger

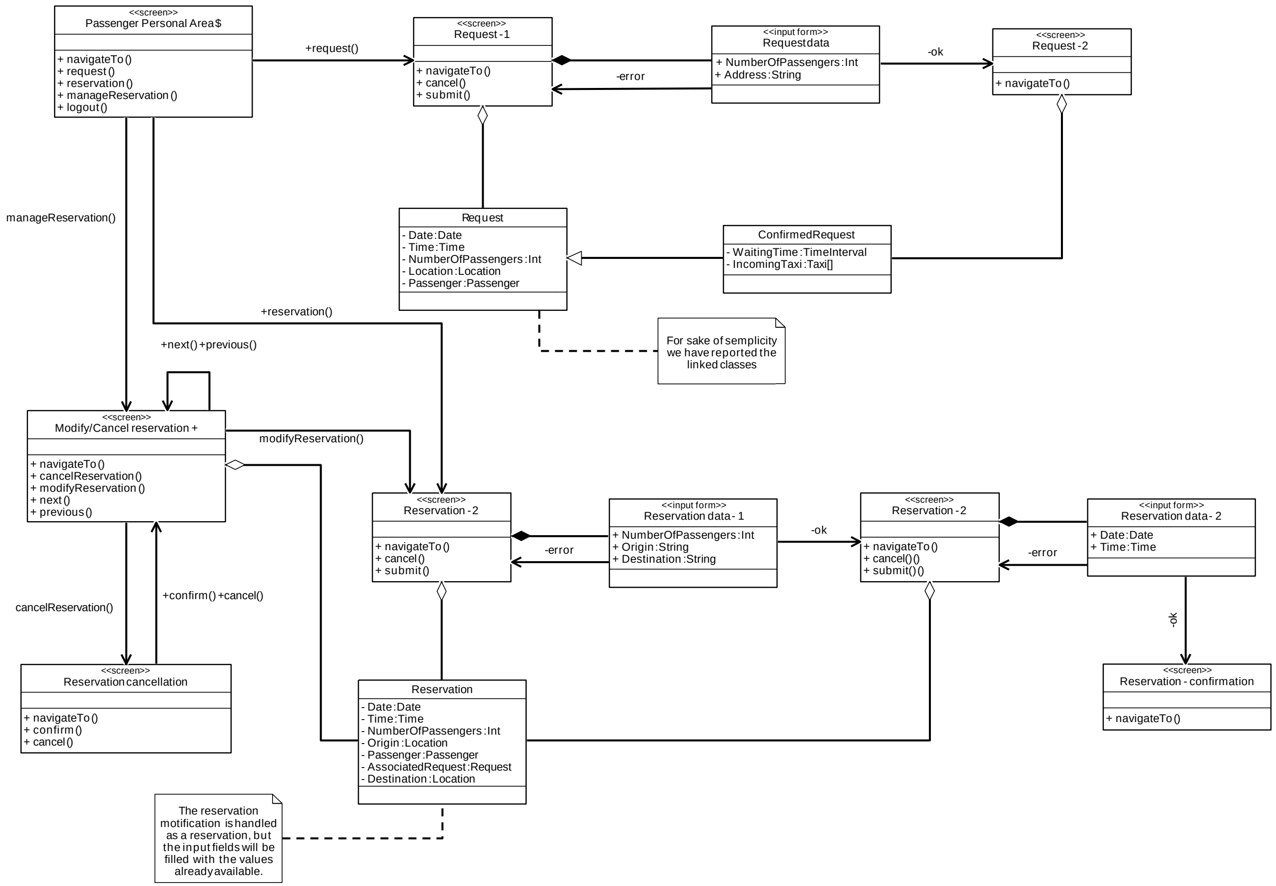


Figure 30: UX diagram - registered passenger

4.2.3 Taxi driver

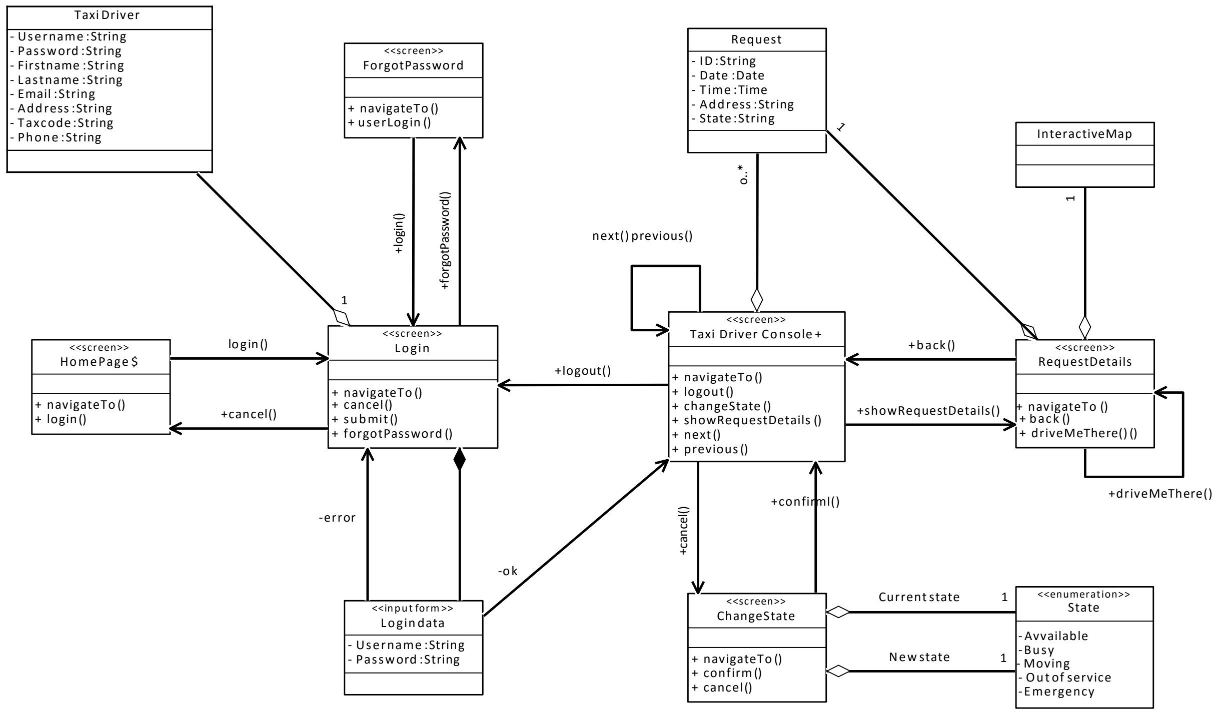


Figure 31: UX diagram - taxi driver

5 Requirements traceability

Providing traceability of system requirements to design components is important for determining if and how system requirements have been realised. The RTM (*Requirements Traceability Matrix*) also is needed to understand if all design components are necessary and to quickly understand the impact of changing a requirement. We will distinguish between functional and non functional requirements. For the detailded description of requirements refer to the RASD.

5.1 Functional requirements

Requirement Component	PMA					TMA					WebSubsystem				BusinessSubsystem									Database
	PMAUserInterface	PMAController	InputValidator	MessageFormatter	CCommunicator	TMAUserInterface	TMAController	InputValidator	MessageFormatter	CCommunicator	SCommunicator	MessageInterpreter	MessageFormatter	CommandEventDisp.	AccountManager	RequestManager	ReservationManager	GoogleMapsComm.	DBManager	TaxiSelector	TaxiStateChanger	TaxiPositionFineder	TaxiQueueManager	
[R1.1]	X	X	X	X	X						X	X	X	X		X		X	X					X
[R1.2]		X																						
[R1.3]																X			X	X	X	X	X	X
[R2.1]	X	X	X	X	X											X		X	X					X
[R2.2]	X	X	X	X	X						X	X	X	X										
[R3.1]	X	X	X	X	X						X	X	X	X										
[R3.2]															X				X					X
[R3.3]	X	X	X	X	X						X	X	X	X										
[R3.4]	X	X	X	X	X						X	X	X	X	X				X					X
[R3.5]	X	X	X	X	X						X	X	X	X	X				X					X
[R4.1]	X	X	X	X	X						X	X	X	X										
[R4.2]																X	X	X	X	X	X	X	X	X
[R4.3]																X	X	X	X					X
[R5.1]	X	X	X	X	X						X	X	X	X										
[R5.2]	X	X	X	X	X						X	X	X	X										
[R5.3]																X	X	X	X	X	X	X	X	X
[R6.1]						X	X	X	X	X	X	X	X	X		X		X	X	X	X	X	X	X
[R6.2]						X	X	X	X	X	X	X	X	X										
[R6.3]																			X	X				X
[R6.4]																				X	X		X	X

Requirement Component	PMA					TMA					WebSubsystem				BusinessSubsystem									Database
	PMAUserInterface	PMAController	InputValidator	MessageFormatter	CCommunicator	TMAUserInterface	TMAController	InputValidator	MessageFormatter	CCommunicator	SCommunicator	MessageInterpreter	MessageFormatter	CommandEventDisp.	AccountManager	RequestManager	ReservationManager	GoogleMapsComm.	DBManager	TaxiSelector	TaxiStateChanger	TaxiPositionFineder	TaxiQueueManager	
[R7.1]						X	X	X	X	X	X	X	X	X					X	X	X	X	X	X
[R7.2]						X	X	X	X	X	X	X	X	X					X	X	X	X	X	X
[R7.3]																			X	X	X	X	X	X
[R8.1]																						X	X	X
[R8.2]																						X	X	X
[R8.3]						X	X	X	X	X	X	X	X	X								X	X	X
[R8.4]																								

5.2 Non functional requirements

Requirement Component	PMA					TMA					WebSubsystem				BusinessSubsystem									Database
	PMAUserInterface	PMAController	InputValidator	MessageFormatter	CCommunicator	TMAUserInterface	TMAController	InputValidator	MessageFormatter	CCommunicator	SCommunicator	MessageInterpreter	MessageFormatter	CommandEventDisp.	AccountManager	RequestManager	ReservationManager	GoogleMapsComm.	DBManager	TaxiSelector	TaxiStateChanger	TaxiPositionFineder	TaxiQueueManager	
Performance	X	X				X	X								X	X	X	X	X	X	X	X	X	X
Relaiability	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Availability											X	X	X	X	X	X	X	X	X	X	X	X	X	X
Security	X		X		X	X		X		X	X			X					X					X
Maintainability	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Portability	X	X				X	X																	
User interface and human factors	X					X																		

A Appendix

Used tools

1. LyX visual editor for LaTeX (`http://www.lyx.org/`) to write this document.
2. Enterprise Architect 11 (`http://www.sparxsystems.com.au/products/ea/`) for UML diagrams.
3. Balsamiq Mockup (`http://balsamiq.com/products/mockups/`) for user inferface mockup generation.
4. Smart Draw (`http://www.smartdraw.com/`) for high level component diagram.

Hours of works

Time spent by each group member:

- Alberto Maria Metelli: 35 h
- Riccardo Mologni: 30 h

Revision history

Version	*Date*	*Revision description*	*Revision notes*
0.1	29-11-2015	Initial draft	-
1.0	4-12-2015	Final draft	-
1.1	8-12-2015	Revision before presentation	Fixed algorithm and requirement traceability sections.
2.0	22-2-2016	Final release	Fixed architectural design and some terminology.