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

Software Engineering 2
Project

version 2.0

22nd February 2016

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

Academic Year 2015–2016

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

RASD

Requirements Analysis and Specification 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 RASD *(Requirements Analysis and Specification Document)* is to give a detailed description, analysis and specification of the requirements for the *myTaxiService* software. This document will explain the *goals* derived from stakeholders' expectations, the characteristics of the application domain and the *assumptions* made to solve ambiguity and incompleteness. Starting from goals and domain properties, *requirements* will be formulated according to a specific systematic methodology and then specified using both informal and formal notations. However this document should not be considered the final draft for the software specifications since in the following phases several fixing may be necessary.

The main objective of the RASD is to achieve good understanding among *analysts, developers, testers* and *customers*, in particular explaining both the application domain and the system to-be; it is also aimed to be a solid base for project planning, software evaluation and possible future maintenance activities. Therefore this document is primarily intended to be proposed to the stakeholders for their approval, to the analysts and programmers for the development of the project and to the testing team the validation of the first version of the software.

1.2 Present system

At the moment taxi service is entirely managed by phone calls. A passenger who wants to request or reserve a taxi has to contact the call center. In case of request, the call center operator moves the call to the first available taxi, otherwise, in case of reservation, a taxi is booked for the specific date, time and address provided by the passenger. No available taxi queues management is implemented at the moment.

1.3 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.

1.4 Definitions, acronyms, abbreviations

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

1.4.1 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 (see 1.4.2).
- *Login credentials*: username and password.
- *Notification*: communication from TS to taxi driver to move to a specific zone.

1.4.2 Acronyms

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

1.4.3 Abbreviations

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

1.5 Actors

In this paragraph a brief description of the various actors affected by myTaxiService system is provided.

- *Passenger*: a person that interacts with myTaxiService to request or reserve a taxi. The interaction with the system may occur by means of either PMA (mobile passenger) or PWA (web passenger). Each passenger can be either a registered passenger or an unregistered passenger.
- *Registered passenger*: specific case of passenger that has already registered to the system. He/She can login, request, reserve a taxi and also visualize and modify the previous reservations.

- *Unregistered passenger*: specific case of passenger that hasn't registered to the system. He/She can only request a taxi.
- *Taxi driver*: a person that drives a taxi and is associated with myTaxiService. He/She interacts with the system confirming or rejecting requests and informing the system about his/her availability by means of TMA.
- *Call center operator*: a person working at the call center that interacts with the system inserting taxi requests coming from phone calls.

1.6 Requirement engineering (Jackson Zave approach)

In order to ensure a sound and complete requirement engineering activity, we decided to follow a systematic technique for requirements formulation proposed by Jackson and Zave. This approach is based on the distinction between the *machine*, the portion of the system to be developed (myTaxiService in our case), and the *world*, the portion of the real world interacting with the machine. Machine and world are tipically non disjoint, some phenomena may affect both of them, they are known as shared phenomena. From this viewpoint, requirement engineering consists in identifying phenomena shared between world and machine, according to a set of **goals** (which express the desired behavior of world phenomena, shared or not) and a set of **domain assumptions** (assertions supposed to be always valid in the world). Formally a set of **requirements** is complete if together with domain assumption it ensures the goals.

1.6.1 Goals

Starting from the available documentation, integrated with some interviews with the stakeholders, the following minimal goals has been identified.

[G1] Allow a passenger to request a taxi for its current position without registration.

[G2] Allow the passenger to visualize the waiting time and the code of the incoming taxi for confirmed requests.

[G3] Allow a registered passenger to have a personal area.

[G4] Allow a registered passenger to reserve a taxi.

[G5] Allow a registered passenger to cancel or modify a previous reservation.

[G6] Allow a taxi driver to either accept or reject a request coming from the system.

[G7] Allow a taxi driver to inform the system about his/her availability.

[G8] Ensure that available taxi queues enjoy the properties specified in sub paragraph 1.6.2.

1.6.2 Queue management

This paragraph is aimed to give a more precise definition of "fair management" of the available taxi queues.

The city is divided into several zones, to each zone a taxi queue is assigned. Each zone is characterized by a different load of requests n_i measured in request/minute. Let N be the total number of taxis available at a certain moment, the number q_i of taxis available in the zone i has to be proportional to n_i, in particular $q_i = Nn_i / \sum_i n_i$. Every time one taxi turns from available to busy or out of service or viceversa a new distribution of the taxis has to be computed; the taxis to be moved have to minimize the cost of movement calculated as number of zones passed through. To prevent too many movements, a fluctuation between -30% and 30% from the value q_i is accepted without performing taxi movements.

1.7 Reference documents

[1] IEEE Software Engineering Standards Committee, "29148-2011 - Systems and software engineering — Life cycle processes — Requirements engineering", 2011.

[2] P, Zave, M. Jackson, Four dark corners of requirements engineering, TOSEM 1997.

[3] Software Abstractions: Logic, Language, and Analysis, revised edition Edition by Daniel Jackson, MIT Press.

[4] Software Engineering 2 course slides.

[5] The assignment of *myTaxiService*.

1.8 Overview

This document is drawn up in accordance to the IEEE Std 830-1998 for Software Requirements Specifications and it is composed of four sections and an appendix.

- The first, this one, gives a general description of the document and brief information about the actors and the purposes of the software.
- The second section provides an overview of the software, highlighting the interaction with external system interfaces and explaining the main functions carried out. It also focuses on constraints and domain assumptions.
- The third section is entirely dedicated to the derivation and specification of the requirements. Several scenarios expressed in natural language will be provided. A generalization of the set of scenarios will be specified as a set of use cases that will be expressed both in natural language and using UML use case diagram. For some groups of use cases a dynamical description will be provided mainly using UML sequence diagram. A high level conceptual description of the classes affected by the system will be given using UML class diagram. For some of the objects involved, we will design a UML state chart diagram showing the evolution of its state.
- The fourth section presents a formalization of a subset of the requirements using Alloy; some significant instances will be shown.
- The appendix contains a model of the goals, 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 Overall description

2.1 Product perspective

myTaxiService (TS) software can be decomposed into four different interacting subsystems (Figure 1); those subsystem are "abstract subsystems" therefore they do not necessarely reflect the architecture designed in the following phases:

1. the *passenger web application* (PWA): it's a web portal that allows passenger to request a taxi, register, login, reserve a taxi and cancel or modify previous reservations. PWA has to be able to identify passenger's position using, if available, the browser geolocalization support.

2. the *passenger mobile application* (PMA): it's an application that shall be installed on passengers' smartphone performing the same functions of PWA. PMA needs also to communicate to a GPS application within the mobile phone, if any available, to retrieve the passenger's position.

3. the *taxi driver mobile application* (TMA): it's an application that shall be installed on taxi drivers' smartphone in order to allow them to receive requests coming from the system, decide to confirm or reject requests and inform the system about their availability.

4. the *queue management system* (QMS): it's a software aimed to compute realtime the distribution of the taxis in the city interfacing with the GPS system of each taxi, decide which taxi assign to a request and send to taxi drivers several notifications.

TS has also to be integrated with the previous taxi management system based on phone calls in order to allow call center operators to forward requests, therefore a specific interface shall be designed. Moreover, the system has to be provided with specific interfaces and APIs in order to allow future requirements extensions.

2.2 User characteristics

The main addressee of myTaxiService are passengers and taxi drivers. Users are not expected to have specific knowledge or technical expertise but it is assumed they are able to operate the internet and to have access to it.

2.3 Constraints

2.3.1 Regulatory policies

The following regulatory policies has to be met by the software.

- Since user's geographic position needs to be shared within the application (either PMA or PWA) to ensure the expected behavior of the system, users has to agree in advance to specific terms and conditions.

- Taxi drivers are obliged not to spread possible collected information about passengers.

2.3.2 Hardware limitations

The following hardware limitations has to be met.

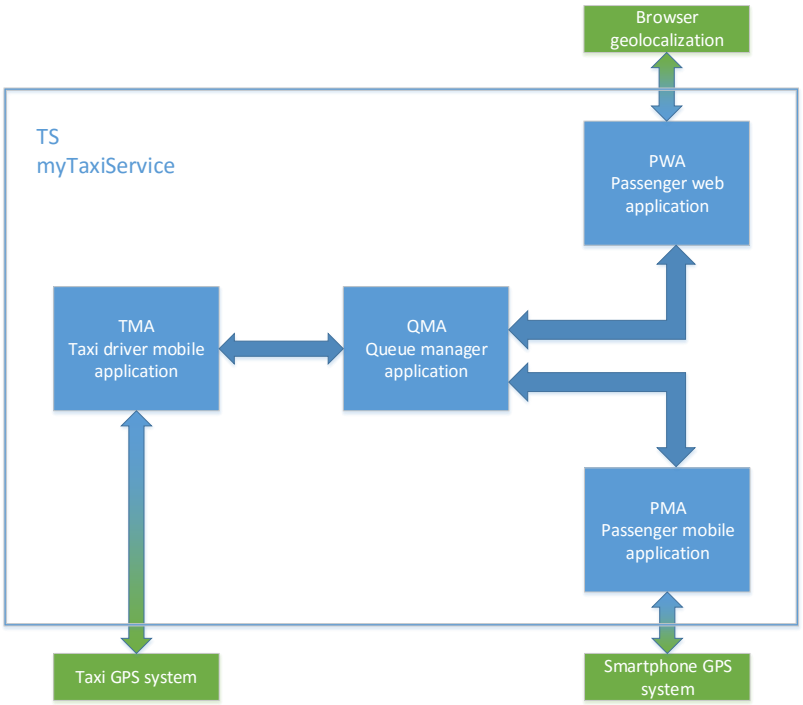


Figure 1: Block schema representing the conceptual interaction between subsystems.

- Mobile passengers has to download the free application from the store (PlayStore for Android users, AppStore for iPhone users, Windows store for Windows users). It is assumed that the mobile phones have enough primary memory to run the application.
- The browser used by web passengers to access the system must have cookies enabled.
- Each taxi driver is provided with a smartphone and the application TMA must be installed.

2.4 Domain assumptions

Considering the specific application domain and according to the information provided by stakeholders we can assume that the following assertions are always valid.

[D1] A taxi driver always executes indications communicated by the system (e.g. move notifications), except in case of emergency.

[D2] Each taxi is provided with a GPS system. If GPS system is not available taxi is considered out of service.

[D3] A taxi can be stopped along the road by a passenger if and only if it is waiting without passenger or moving but not for carrying out a request. In this case taxi driver informs the system about his/her unavailability.

[D4] When a taxi driver finishes to carry out a request he/she informs the system about his availability.

[D5] When a taxi driver starts his work-shift sets his/her state from out of service to available.

[D6] When a taxi driver ends his work-shift if the current state is available then the taxi state becomes out of service; otherwise taxi driver finishes to carry out the current request and after that the taxi becomes out of service.

[D7] When a taxi driver gets to the meeting place, waits for 10 minutes; if passenger does not arrive taxi driver informs the system.

[D8] A taxi is assigned to a unique taxi driver at time. It is possible that some taxi drivers are not assigned to any taxi or vice versa.

[D9] If a ride gets out of the city the taxi driver comes back to the last zone before informing TS of his availability.

[D10] There are only two types of taxi: normal (4 seats) and minivan (9 seats).

[D11] Each available taxi belongs to exactly one queue at time. Busy or out of service taxis do not belong to any queue.

[D12] TS is available only in the city, no requests coming from outside of the city boundary are accepted.

[D13] Taxi drivers have always access to the Internet.

[D14] Taxi drivers' work-shifts are managed in order to ensure that at each moment the number of in service taxis is at least 50% of the total number of taxis during the day and 20% during the night.

[D15] Taxi drivers go in emergency state only in case of car accident or similar events.

[D16] Taxi can move without limitations inside the zone assigned by TS system but they cannot change the zone without a notification from the system.

Note that also taxi drivers are identified by the system but registration of the taxi driver is not part of the TS system since it reasonably involves contractual issues (taxi driver has to make an agreement with the company) that cannot be directly managed by the system. Therefore registration is not performed by taxi driver.

2.5 Possible future extensions

The following are reasonable possible future extensions to the TS system; they are mainly meant to further improve the usability and the performances of the service. They will not be discussed in details.

- At present, queues has a fixed suitable number of available taxis which is supposed to be calculated using previous data about the number of requests coming from each zone. However the distribution of the requests can vary not only *spatially* (from one zone to an other) but also *temporally* (for each zone at different moments of the day the number of requests might be different). Also in different days of the week the distribution of requests may vary significantly. A possible solution to make TS more adaptive is the one in which the suitable number of taxis for each queue is periodically determined integrating data collected from the requests in a certain time horizon (for instance once a week).
- At the moment TS system does not handle payments, since users are expected to pay the ride cash or with credit card to the taxi driver; online payments can be implemented within TS system. Both PWA and PMA should allow registered users to pay the price of the ride using credit card or paypal; cash payment should be still possible.

- At the moment, passenger requesting a taxi can only see the estimated waiting time and the code of the taxi, while visualizing also the current position of the incoming taxi should be more useful.
- An evaluation system of the quality of service can be added, allowing passengers to express an opinion about taxis.

3 Specific Requirements

3.1 External Interface Requirements

This section provides a description of the interaction between TS system and users (passengers and taxi drivers) and external systems (GPS system, browser geolocalization and Google Maps). We will put the accent on the characteristics of each interface.

3.1.1 User Interfaces

Since users are not expected to have a technical knowledge, user interface has to be designed in order to enhance the usability of the software. In the following, some mockups of the main features available for the passengers by means of both web and mobile interface will be presented; they should be considered just a draft.

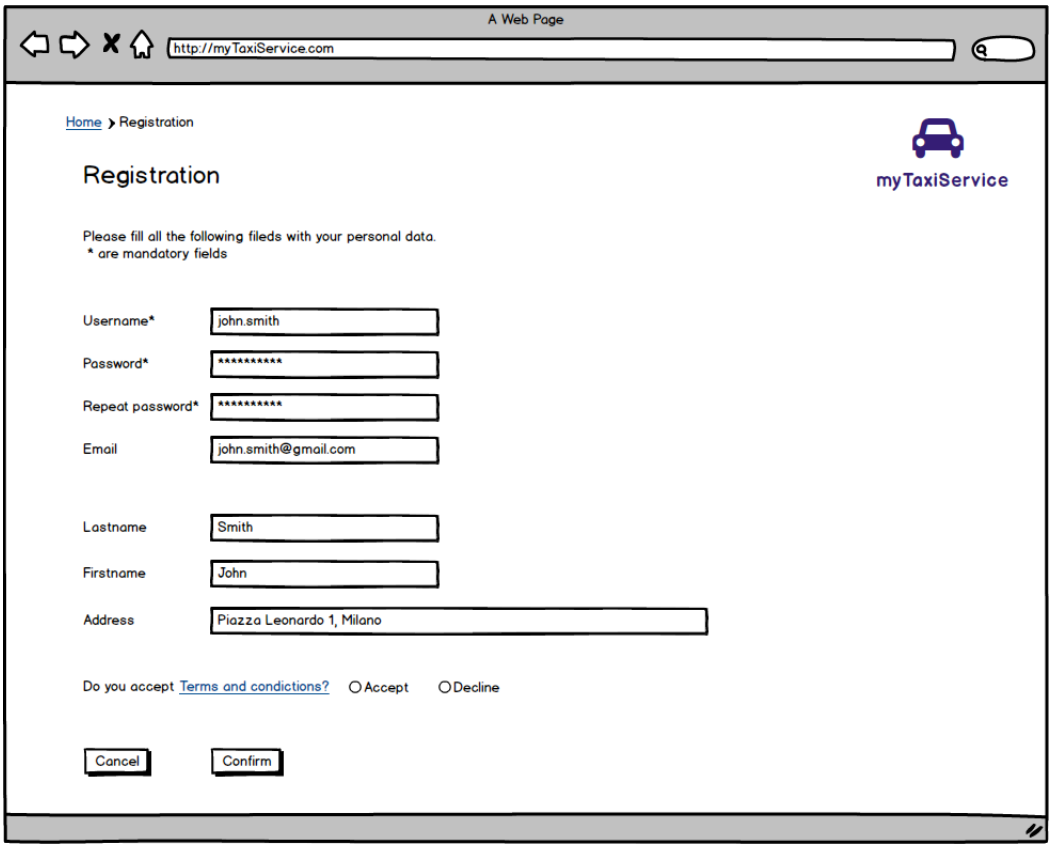


Figure 2: Registration form (web interface)

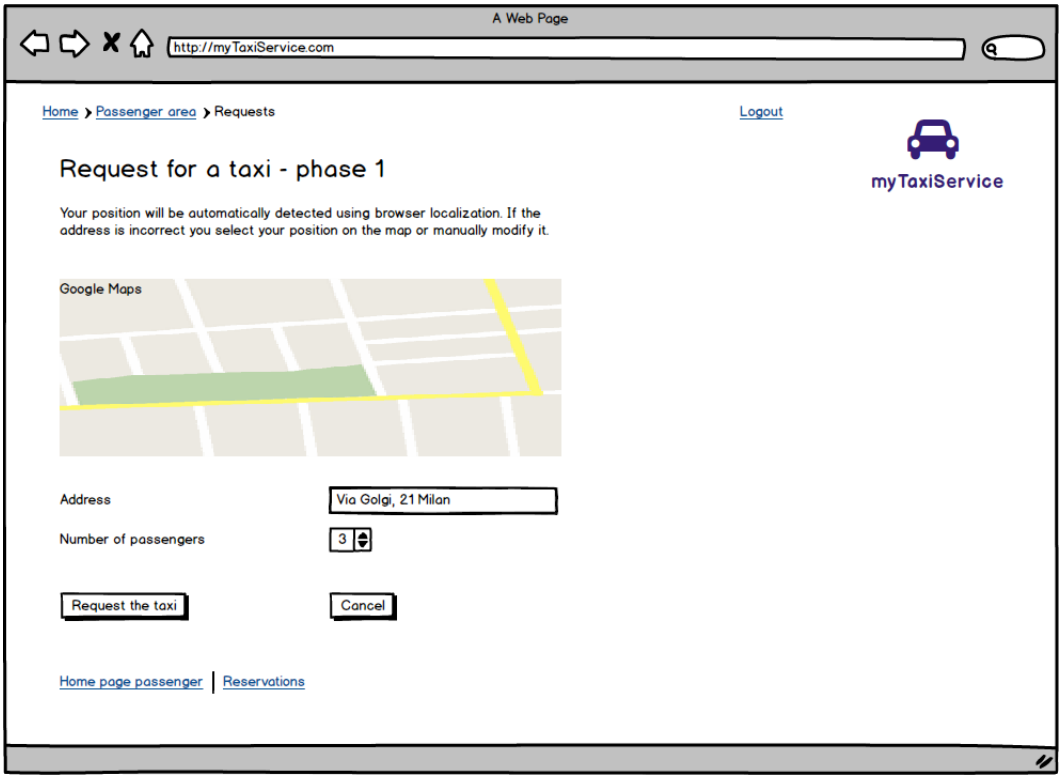


Figure 3: Request for registered passenger 1 (web interface)

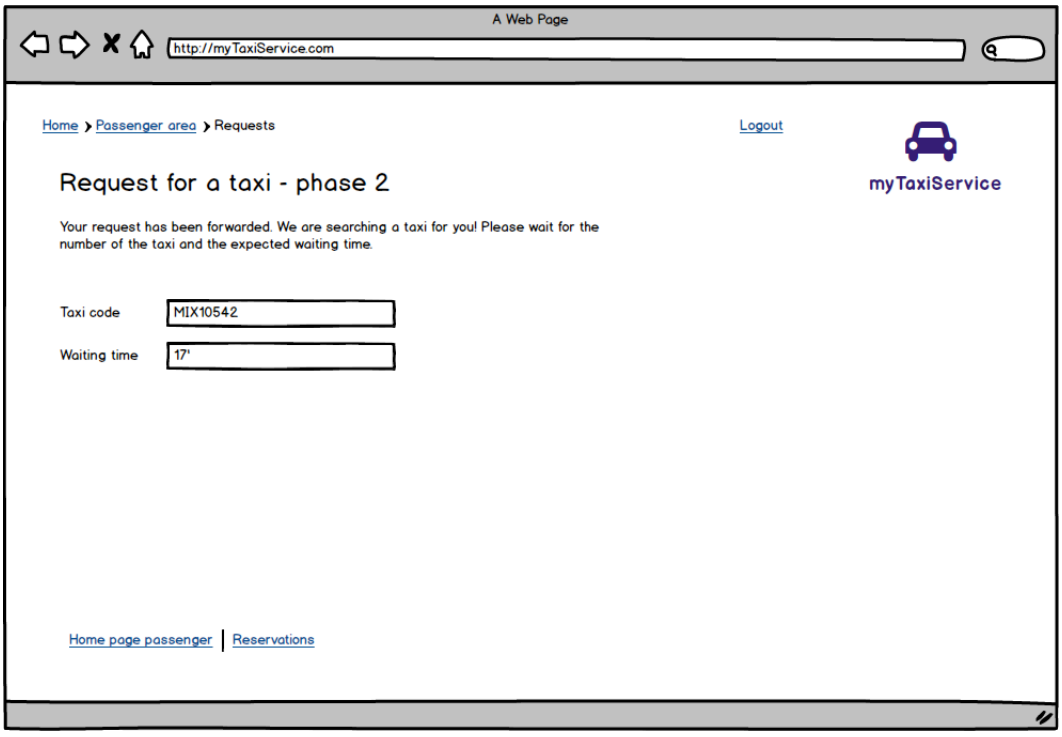


Figure 4: Request for registered passenger 2 (web interface)

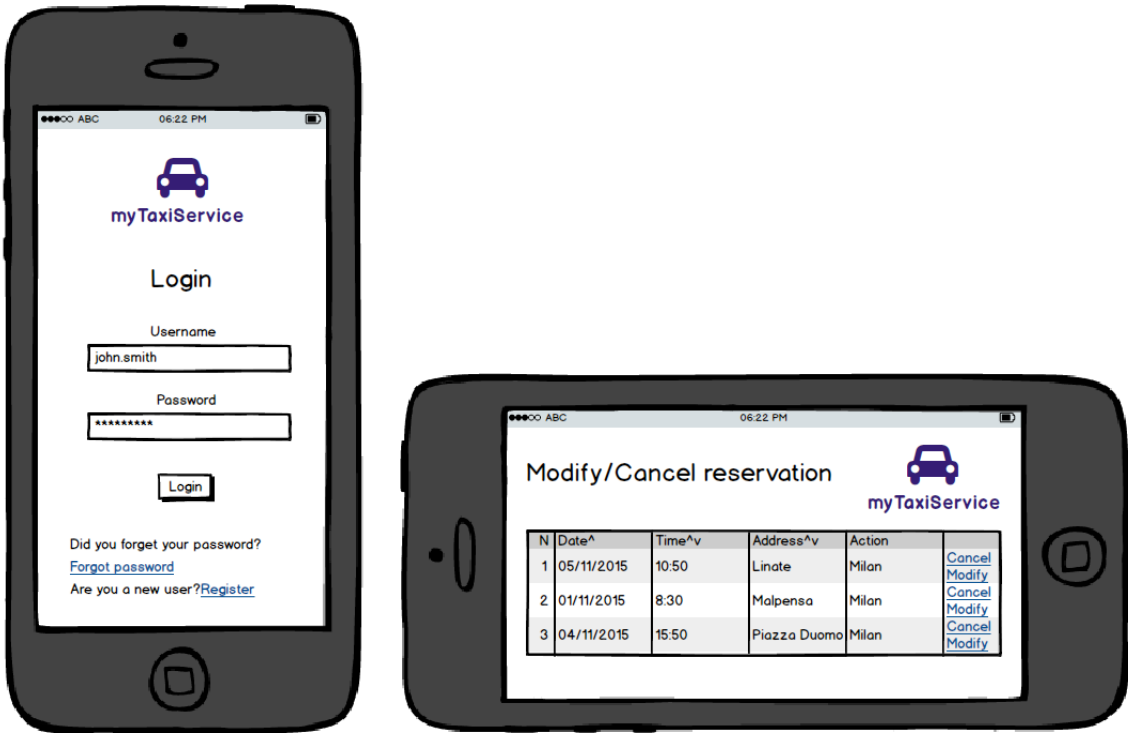


Figure 5: Login (mobile interface) - Modify/Cancel reservation (mobile interface)

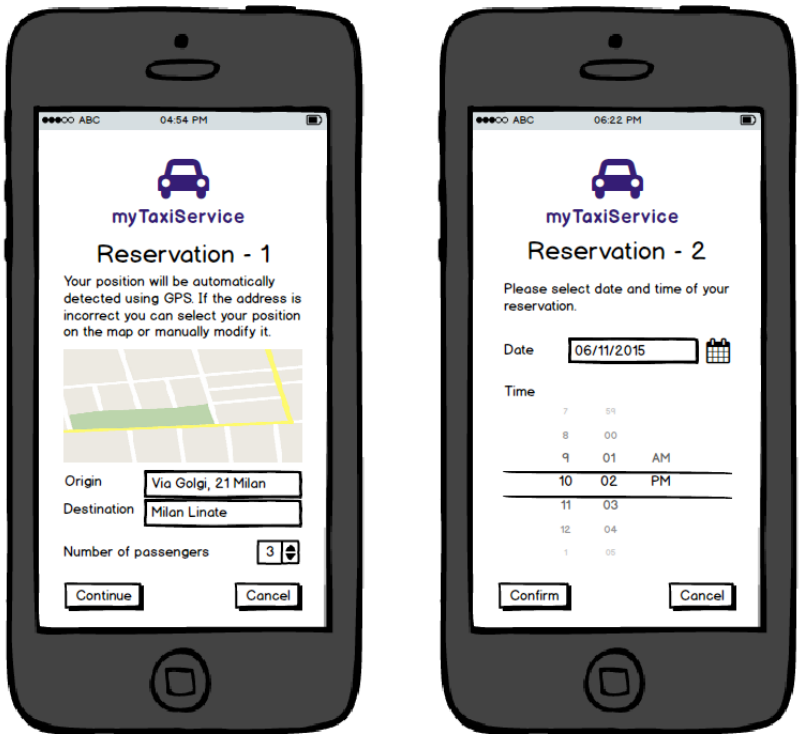


Figure 6: Reservation 1/2 (mobile interface)

3.1.2 Software Interfaces

TS is interfaced to the following external systems:

- GPS system of mobile phones for mobile passengers;
- browser geolocalization for web passengers;
- GPS system of taxis;
- Google Maps for the estimation of the waiting time and the specification of the address when geolocalization or GPS are unavailable.

Moreover, TS offers a set of APIs to enable the development of additional services (e.g., taxi sharing). Those APIs allows developers to have access the basic functionalists of TS.

3.1.3 Communication Interfaces

In order to work properly TS system must have access to the Internet and all devices involved has to communicate by means of TCP/IP.

3.2 Functional Requirements

In this section, according to the *Jackson Zave approach*, we derive the requirements for TS system. Each requirement is related to the goal it is intended to satisfy to make the reading simpler.

3.2.1 [G1] Allow a passenger to request a taxi for its current position without registration.

[R1.1] TS shall provide the passenger with a form in which he/she has to insert the total number of passengers and accept terms and conditions.

[R1.2] TS shall retrieve automatically passenger's position if GPS or browser geolocalization is available, otherwise user has to specify his address.

[R1.3] After confirmation, TS shall store the request and

- [R.1.3.1] Assign it to the first available taxi in the queue of the zone.
- [R.1.3.2] If the queue is empty, TS shall look for taxis in the queues of adjacent zones and, if necessary, repeat the process for the other adjacent zones.
- [R.1.3.3] If no taxi is found, TS shall inform passenger and put request on hold.

3.2.2 [G2] Allow the passenger to visualize the waiting time and the code of the incoming taxi for confirmed requests.

[R2.1] TS shall compute the expected waiting time for a confirmed request.

[R2.2] TS shall provide the user with a form showing the waiting time and the code of the incoming taxi.

3.2.3 [G3] Allow a registered passenger to have a personal area.

[R3.1] TS shall provide the user with a registration form in which he/she has to insert firstname, lastname, email and address and choose a username and a password.

[R3.2] TS shall verify if the user is not already registered, if not TS shall send a confirmation mail, otherwise cancel the process.

[R3.3] TS shall provide the user with a login form in which he/she has to insert username and password.

[R3.4] TS shall verify if username and password are correct, if so TS shall show the passenger's home page, otherwise report the login failure.

[R3.5] TS shall provide a procedure to recover the password.

3.2.4 [G4] Allow a registered passenger to reserve a taxi.

[R4.1] TS shall provide the registered passenger with a form in which he/she has to insert the total number of passengers, the origin and destination of the ride, the date and time of the meeting.

[R4.2] TS shall accept only reservations made at least two hours in advance.

[R4.3] TS shall store the reservation, allocate a request 10 minutes before the meeting time.

3.2.5 [G5] Allow a registered passenger to cancel or modify a previous reservation.

[R5.1] TS shall provide the registered passenger with a table containing all previous reservations.

[R5.2] TS shall provide for each of them a cancellation and modification procedure.

[R5.3] TS shall verify after modification the correctness of the new data.

3.2.6 [G6] Allow a taxi driver to either accept or reject a request coming from the system.

[R6.1] TS shall show to the chosen taxi driver the request indicating coordinates of the passenger and total number of passengers.

[R6.2] TS shall provide the taxi driver with a form allowing him to choose if accept or reject the request.

[R6.3] TS prevents taxi driver to reject twice the same request.

[R6.4] In case of acceptance, TS shall put the taxi driver into state *busy*, otherwise put taxi driver at the end of the queue and repeat [R1.3]. If no answer from the taxi driver in one minute it is interpreted as a rejection.

3.2.7 [G7] Allow a taxi driver to inform the system about his/her availability.

[R7.1] TS shall provide the taxi driver with the possibility to set the state to *busy* if the current state is *available* and he/she is picking up a passenger along the road and viceversa when the passenger gets out or the passenger isn't there.

[R7.2] TS shall provide the taxi driver with the possibility to set the state to *out of service* if the current state is *available* and viceversa.

[R7.3] Whenever a taxi goes into *available* it is put at the end of the queue of the zone it is currently located.

3.2.8 [G8] Ensure that available taxi queues enjoy the properties specified in sub paragraph 1.6.2.

[R8.1] TS shall retrieve the position w.r.t. the zones and the state of the taxi whenever the state of a taxi driver changes or a request comes or anyway periodically.

[R8.2] TS shall insert in each queue all *available* taxis currently located in each zone.

[R8.3] For each queue lacking of taxis, TS shall move to that zone several taxis minimizing the total cost (number of zones traversed).

[R8.4] TS shall inform the taxi driver to move to a certain zone (if needed for queue management) by means of notification. That taxi is put in *moving* state[1] and it goes back to *available* when arrives to the specified zone.

[1] *Moving* state from the outside is equivalent to *available* state, it is just used by TS to remember that the taxi is moving to a specific zone and avoid to move other taxis. In particular, that taxi during the movement is added at the end of the destination zone queue.

3.3 Scenarios

In order to clarify the expected functionality of the system, in this subsection we propose several possible scenarios. In each scenario we emphasize several specific interactions between the users and the system, but, in some of them, we do not describe the entire process carried out interacting with TS to avoid useless repetitions.

3.3.1 Scenario 1

Bob is an unregistered passenger of TS. He has decided to visit a friend that lives in another district. He accesses the TS using the PWA via browser installed on his PC, he specifies his current address because his PC can't provide him with browser geolocalization. After accepting terms and conditions related to the service he confirms the request. The TS system processes his request and sends a notification to Bob. Meanwhile the system sends the request to the first available taxi in the queue associated to the area where Bob is. The taxi driver, Tom, after having received the request on his smartphone via TMA, he confirms it. Tom visualizes the request information details and leaves from his location to get to the address specified by Bob. TS system calculates the waiting time and shows it to Bob. When Tom arrives to Bob's, he picks up him and brings him to his friend. Before leaving Bob pays Tom for the service and Tom informs the system that he is available again.

3.3.2 Scenario 2

As soon as Alice left the theater it started raining, so she decided to try TS service for the first time. She uses her 3G connection to download the PMA and starts it. After that she accepts terms and conditions of TS, the system retrieves Alice's current position by means of her GPS. She confirms the requests and the system contacts the first taxi available in her zone. Unfortunately it has run out of fuel so, Tim, the taxi driver is waiting his turn at the gas station and decides to refuse the request. The system puts the Tim's taxi at the end of the queue and sends a request at the second taxi available. Tess, the new taxi driver, accepts the request, picks up Alice and brings her to home.

3.3.3 Scenario 3

Robb needs to reach the university campus because he has to addend Software Engineering 2 morning class, so he reaches the bus stop but he notices a sign informing that an unexpected public transportation strike is going on. Unfortunately this happens quite often, that's why Robb is a registered passenger of TS. He logs in the PMA and requests a taxi. Due to the strike a large number of requests has been forwarded the system so no taxi is available at the moment in that zone of the city. Therefore the first taxi in the queue of an adjacent zone is contacted. The taxi driver, Taylor, accepts the request. The system shows the expected waiting time on Rob's smartphone. After 15 minutes Taylor picks up Robb who will get to university, in time for the class.

3.3.4 Scenario 4

Arya booked a low cost flight to Marrakesh but the plane leaves early in the morning so she decided to register on TS website to reserve a taxi. She filled the registration form with the requested information and confirmed, accepting terms and conditions. The system sends her a confirmation e-mail. A few days before the departure she logs in her personal area on the PWA and reserves a taxi, specifying her address and the meeting time. The leaving day she hasn't heard the alarm clock and she hasn't woken up. When Takashi, the taxi driver, arrives at Arya's place he doesn't find anyone so he waits for 10 minutes and then informs the system. The system puts Takashi's taxi at the end of the available taxi queue.

3.3.5 Scenario 5

In the city a huge sport event has been organized for today so the city center is closed. Jon wants to reach his house after a stressful working day. Jon usually comes back home using public transportation but today it's a mess so he decides to take a taxi. He sends a request using his PMA but all the taxis in all the zones are busy due to the unexpected number of people coming to attend the event. The system informs Jon about the situation and provides Jon with the expected waiting time: one hour, too much for Jon. So he decides to ask one of his colleagues for a lift.

3.3.6 Scenario 6

Sansa always uses taxis to move in the city because she has never passed her driving license exam. Today she needs a taxi to reach the shopping center but unfortunately she doesn't have access to the Internet so she calls the taxi service phone number. A call center operator, Trudy, answers the phone and asks the location where the taxi should pick up her and the number of passengers. Sansa tells him her current address and says she would be the only passenger. Trudy uses a PWA to send a request to the TS system and announce to Sansa the expected waiting time when she receives the confirmation by TS.

3.4 Use Cases

Starting from the requirements and generalizing scenarios we have identified several use cases that represent basic functional units carried out by TS. The following comprehensive UML diagram shows how use cases are related one another and how actors interact with them; each one is also provided with a table describing its main features.

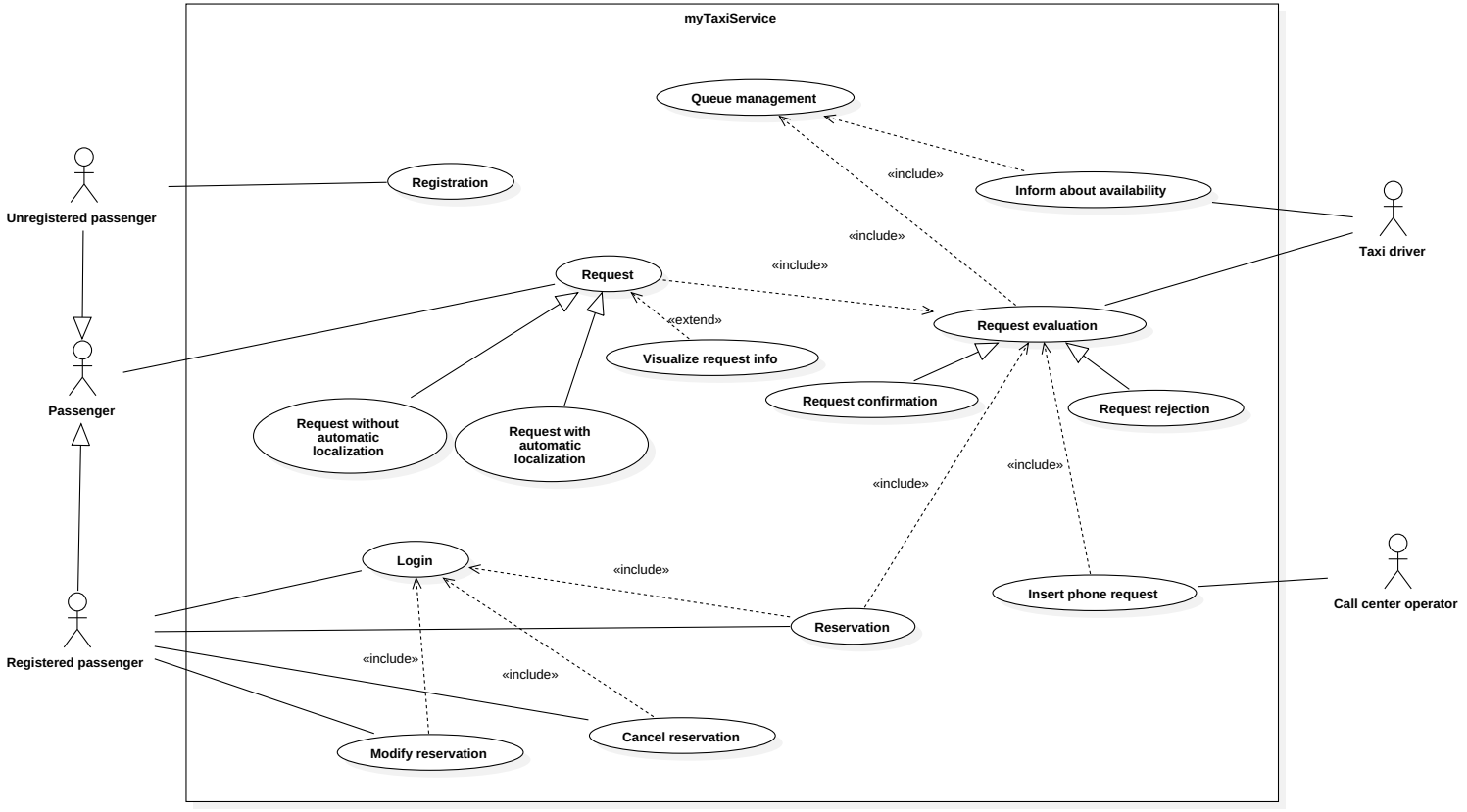


Figure 7: UML Use Case Diagram

3.4.1 Registration

Name	Registration
Related goals	[G3]
Actors	Unregistered passenger
Entry condition	Nothing.
Flow of events	1. Unregistered passenger opens the registration page on passenger's application. 2. Unregistered passengers fills the form with the required information (firstname, lastname, email, username, password, address). 3. Unregistered passenger agrees on terms and condition. 4. Passenger's application submits the passenger's data to TS system. 5. TS system checks their validity. 6. TS system creates a new RegisteredPassenger with data provided by unregistered passenger. 7. TS sends a confirmation e-mail to the user.
Exit condition	The passenger information are lasting memorized in the TS system.
Exceptions	• If some information used to fill the form are invalid, the registration process is interrupted and an error message is shown to the passenger. The passenger can restart the procedure. • If the procedure was interrupted before its termination by external event (e.g. connection lost, system error, hardware failure) the procedure is rolled back and no modifications are done in the TS system.
Special requirements	Nothing.

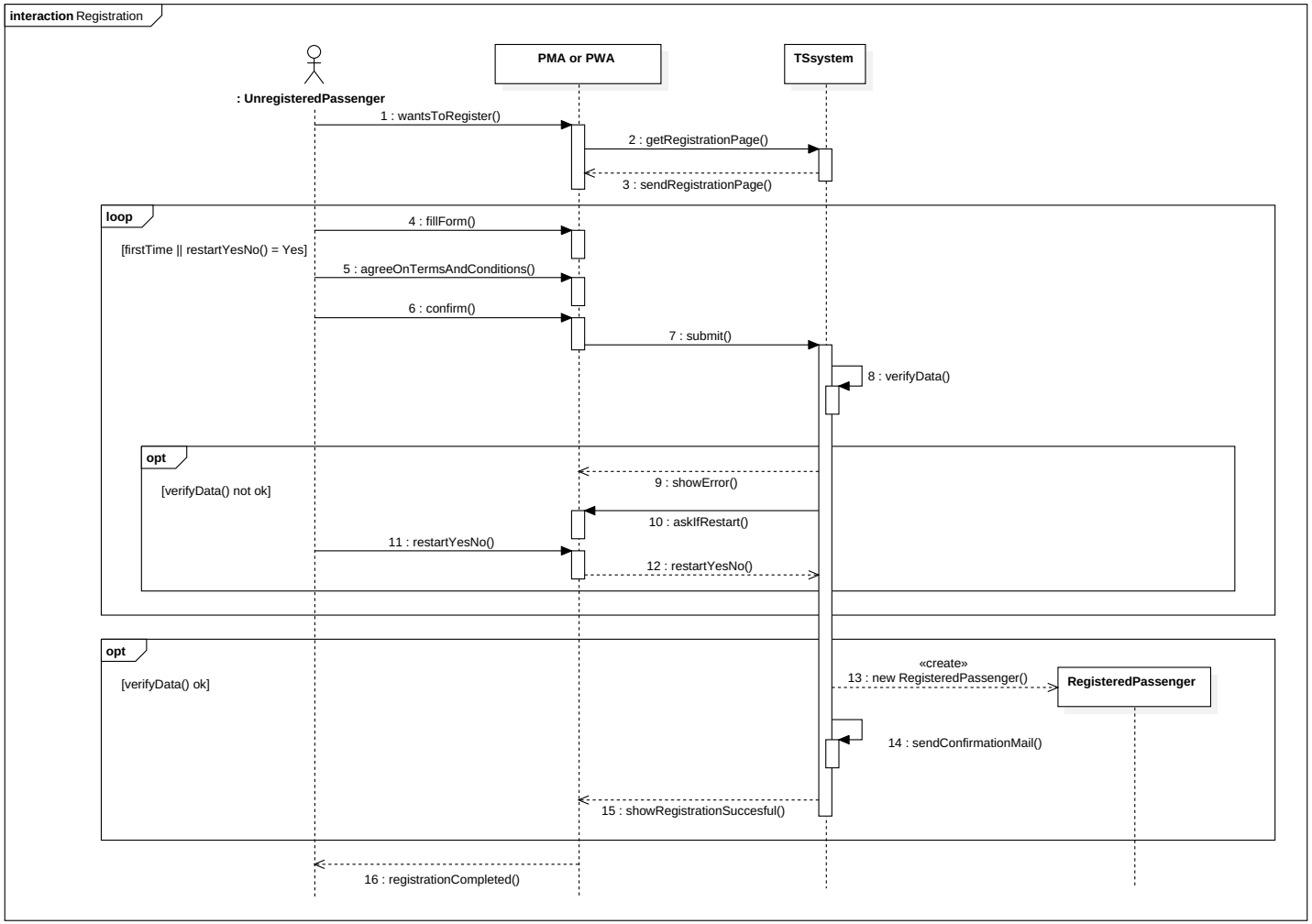


Figure 8: Registration - UML sequence diagram

3.4.2 Request

Name	Request
Related goals	[G1]
Actors	Passenger
Entry condition	Nothing.
Flow of events	1. Passenger opens the request page on passengers' application. 2. The passenger's application retrieves the localization asking to passenger or using automatic geolocalization. 3. Passenger inserts the number of passengers. 4. Passanger accepts terms and conditions. 5. Passenger's application sends data to QMA. 6. QMA creates a new request.
Exit condition	A new request is created by QMA and "Request evaluation" is performed.
Exceptions	See "Request without automatic geolocalization" and "Request with automatic geolocalization".
Special requirement	Nothing.

3.4.3 Request without automatic geolocalization

Name	Request without automatic geolocalization→Request
Related goals	[G1]
Actors	Passenger
Entry condition	Device used to perform request that can't provide an automatic geolocalization.
Flow of events	1. Passenger opens the request page on passengers' application. 2. Passenger specifies his/her address within passenger's application. 3. Passenger inserts the number of passengers. 4. Passanger accepts terms and conditions. 5. Passenger's application sends data to QMA. 6. QMA creates a new request.
Exit condition	A new request is created by QMA.
Exceptions	• If the address specified by passenger isn't found by TS system (also considering very similar addresses) an error message is sent and the user has to insert a new address.
Special requirement	Nothing.

3.4.4 Request with automatic geolocalization

Name	Request with automatic localization →Request
Related goals	[G1]
Actors	Passenger
Entry condition	Device used to perform request is able to provide a automatic geolocalization.
Flow of events	1. Passenger opens the request page on passengers' application. 2. The passenger's application retrieves retives the localization using automatic geolocalization. 3. Passenger insterts the number of passengers. 4. Passanger accepts terms and conditions. 5. Passenger's application sends data to QMA. 6. QMA creates a new request.
Exit condition	A new request is created by QMA.
Exceptions	• If the system fails to retrive automatic localization or the information are invalid, an error message is shown and the passenger that can insert manually the address ("Request without automatic geolocalization" is activated)
Special requirement	Nothing.

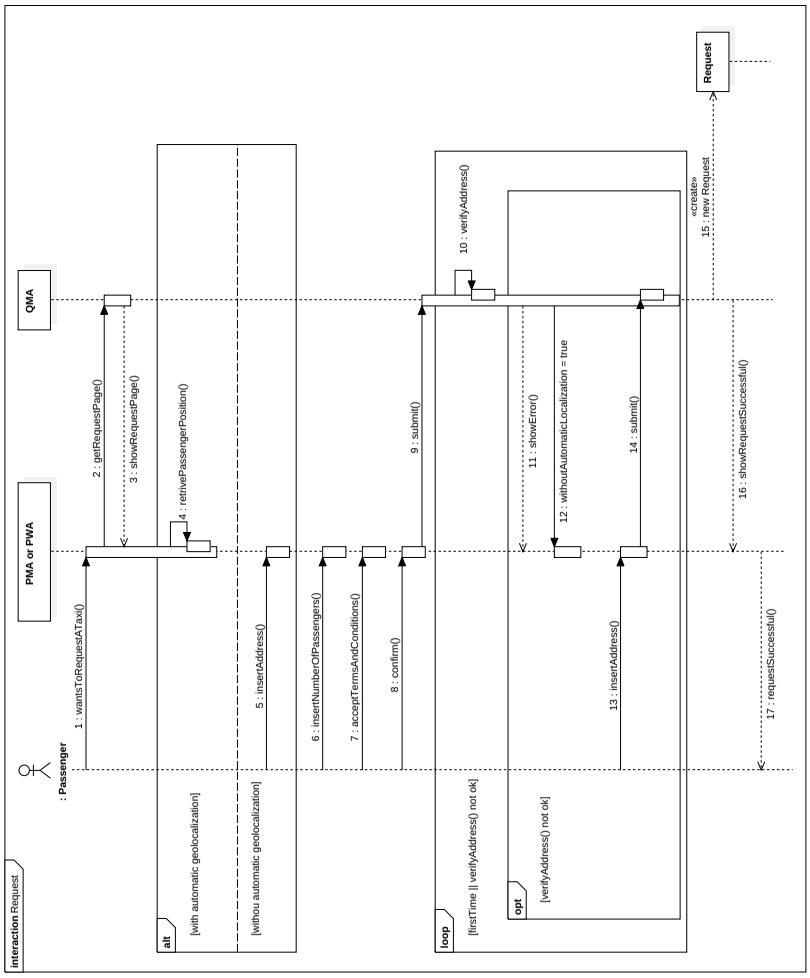


Figure 9: Request - UML sequence diagram

3.4.5 Login

Name	Login
Actors	Registered passenger
Related goals	[G3]
Entry condition	The passenger is already registered to TS system.
Flow of events	1. Passenger accesses to the login area. 2. Passenger inserts credentials (username and password). 3. The passenger's application sends the credentials to TS system. 4. TS system checks for correctness of credentials. 5. TS system sends a confirmation to the passenger's application.
Exit condition	Passenger can see personal area.
Exceptions	• If credentials are wrong, a message is shown to the passenger and he is redirected to the login page.
Special requirement	Nothing.

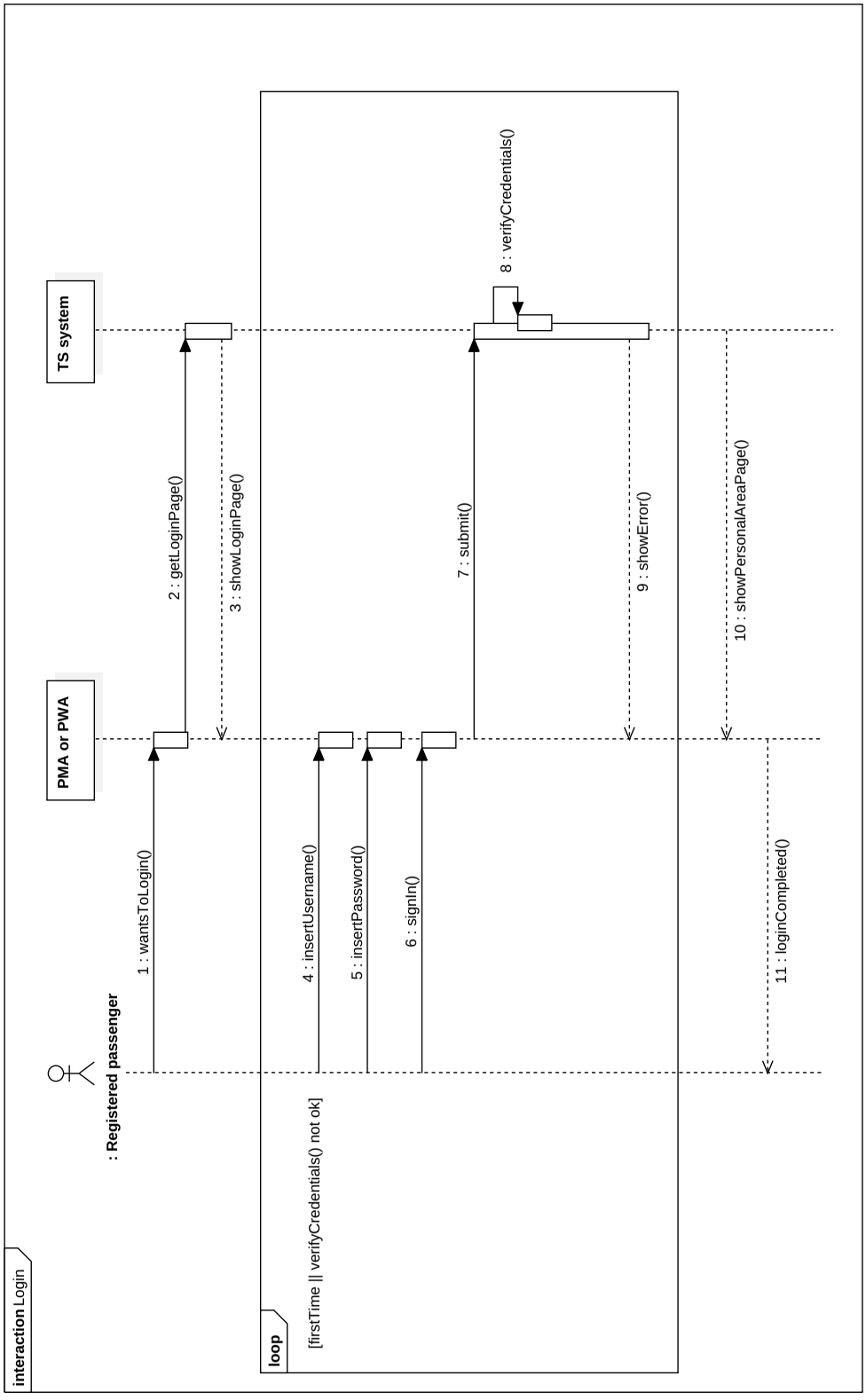


Figure 10: Login - UML sequence diagram

3.4.6 Reservation

Name	Reservation
Related goals	[G4]
Actors	Registered passenger
Entry condition	Passenger is logged in.
Flow of events	1. Passenger accesses to the reservation area. 2. Passenger inserts the required data (origin, destination, date, time, number of passengers). 3. Passenger confirms the reservation. 4. TS system whether data are valid. 5. QMA creates a new reservation and the related request is allocated.
Exit condition	The reservation is added to the TS system.
Exceptions	• If passenger does not confirm the operation is not performed. • If the data are not valid (origin, destination, number of passengers) an error message is shown to passenger and the operation is not performed. Passenger can repeat the process. • If the date and time are such that the reservation is not made at least two hour in advance an error message is shown to user and the operation is not performed. Passenger can repeat the process.
Special requirement	Nothing.

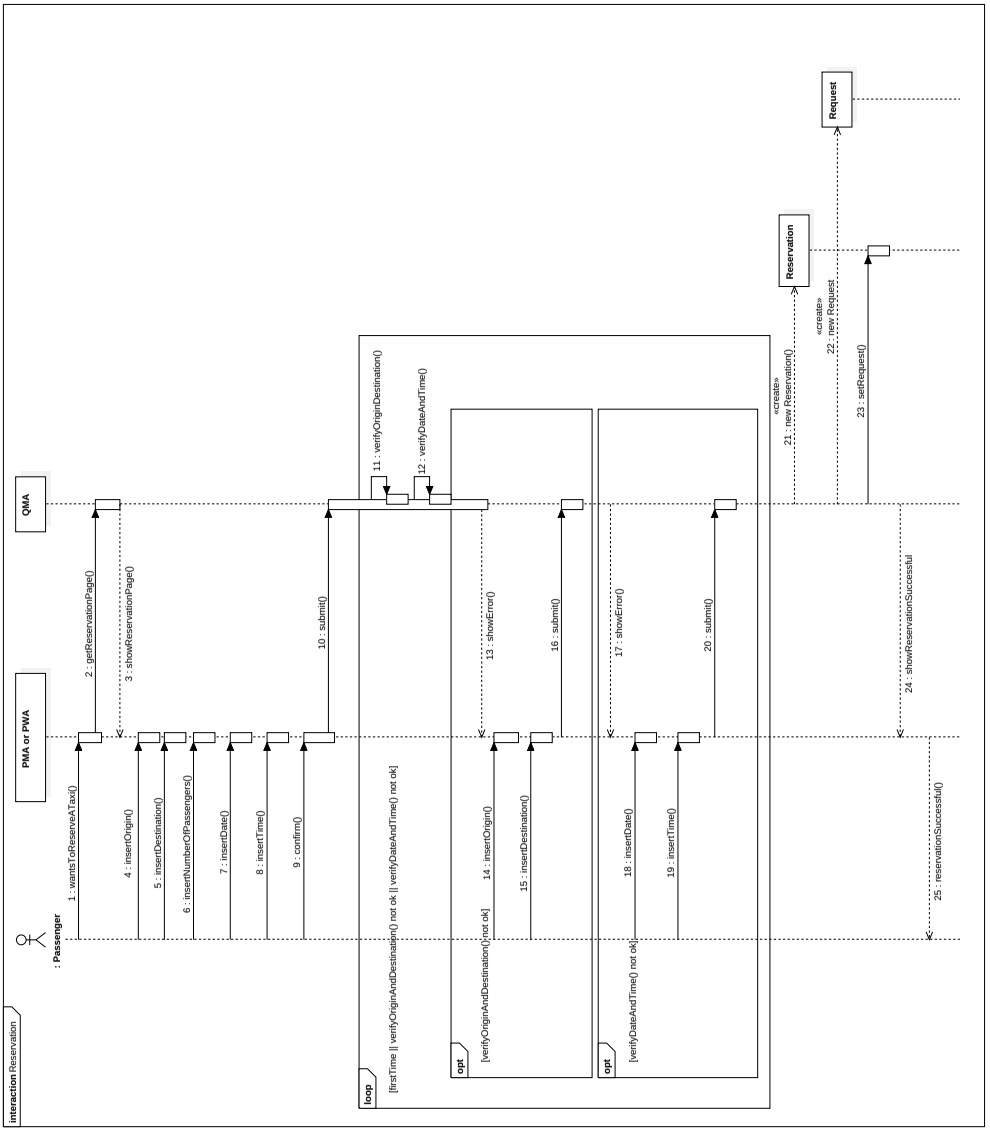


Figure 11: Reservation - UML sequence diagram

3.4.7 Cancel reservation

Name	Cancel reservation
Related goals	[G5]
Actors	Registered passenger
Entry condition	Passenger has already forwarded a reservation and he/she is logged in.
Flow of events	1. Passenger accesses to previous reservations area. 2. Passenger selects the reservation to be canceled. 3. TS asks passenger for a confirmation. 4. Passengers confirms the operation. 5. TS system removes the reservation and the associated request.
Exit condition	The reservation is delated by the TS system.
Exceptions	• If passenger does not confirm the operation is not performed.
Special requirement	Nothing.

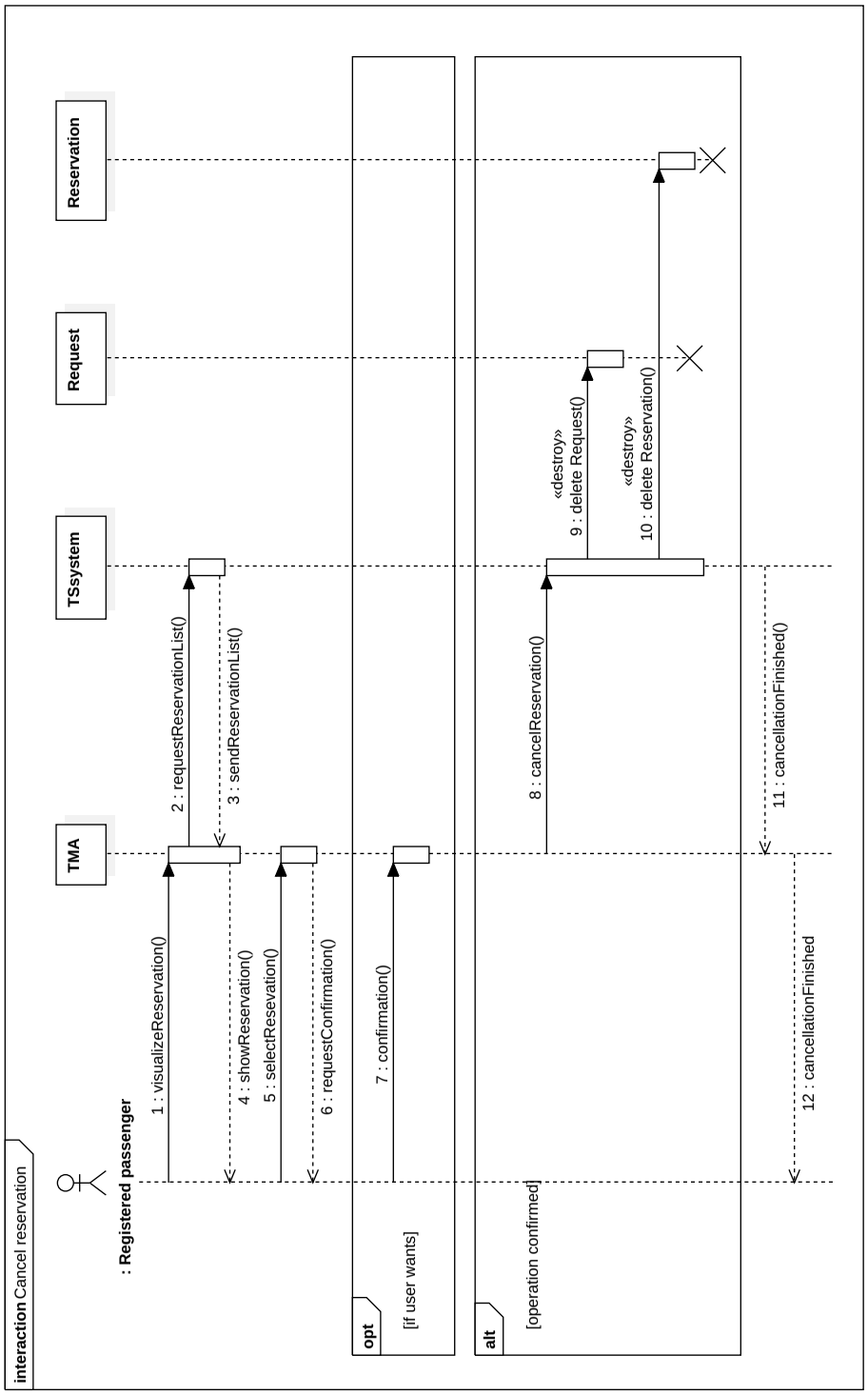


Figure 12: Cancel reservation - UML sequence diagram

3.4.8 Modify reservation

Name	Modify reservation
Related goals	[G5]
Actors	Registered passenger
Entry condition	Passenger has already forwarded a reservation and he/she is logged in.
Flow of events	1. Passenger accesses to previous reservations area. 2. Passenger selects the reservation to be modified. 3. Passenger modifies data related to reservation (origin, destination, date time, number of passengers). 4. Passengers confirms the operation. 5. The application sends the data to TS system. 6. TS system checks whether new data are valid. 7. QMA updates the reservation and the associated request.
Exit condition	The modifications to the reservations are stored in the the TS system.
Exceptions	• If passenger does not confirm the operation is not performed. • If the new data are not valid (address, date, time, number of passengers) an error message is shown to passenger and the operation is not performed. • If the new date and time are such that the reservation is not made at least two hour in advance an error message is shown to user and the operation is not performed.
Special requirement	Nothing.

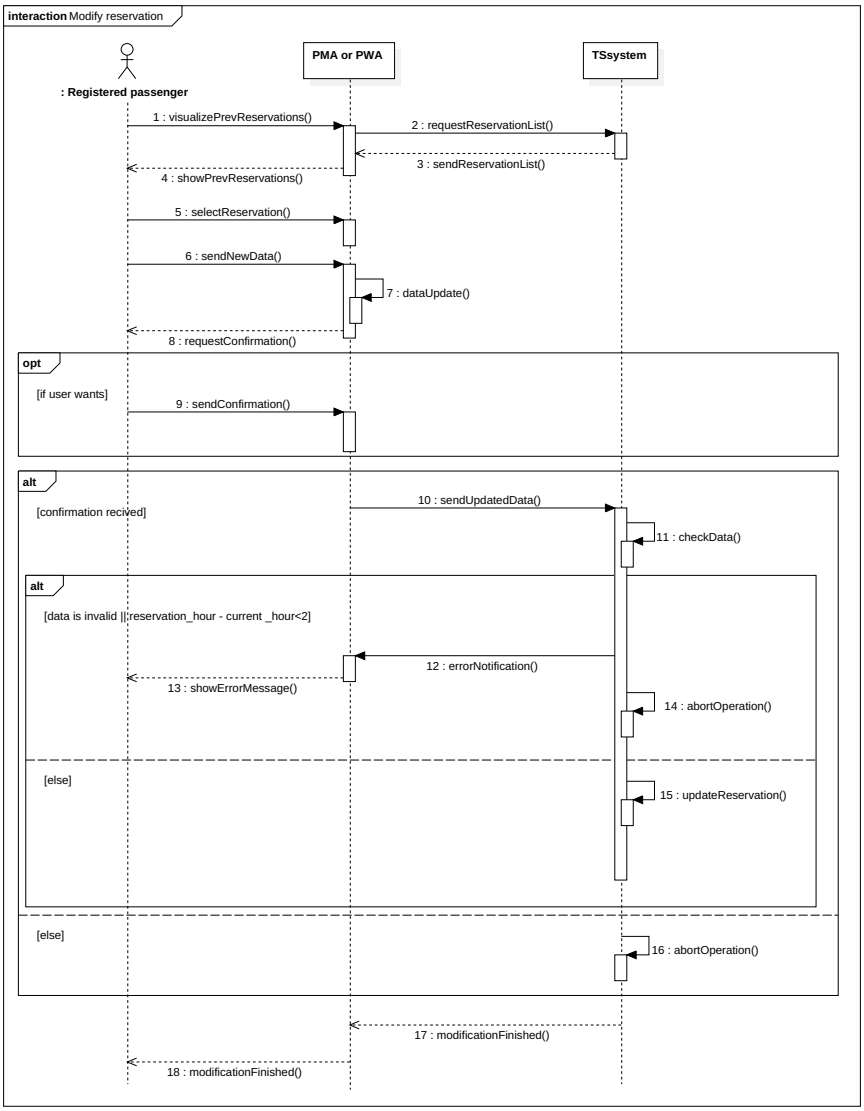


Figure 13: Modify reservation - UML sequence diagram

3.4.9 Queue management

Name	Queue management
Related goals	[G8]
Actors	-
Entry condition	• A new request is created or • a taxi driver changes his state or • performed periodically.
Flow of events	1. QMA retrives localization of each taxi and extracts the zone for each. 2. QMA updates available taxi queue for each zone, possibly moving taxis from a queue to an other (they are added at the end of the queue). 3. QMA computes the taxis to be moved to one zone to another. 4. QMA sends notification to those taxis. 5. QMA sets the state of those taxis to *moving*.
Exit condition	New distribution of taxis is stored in TS systems.
Exceptions	Nothing.
Special requirement	Nothing.

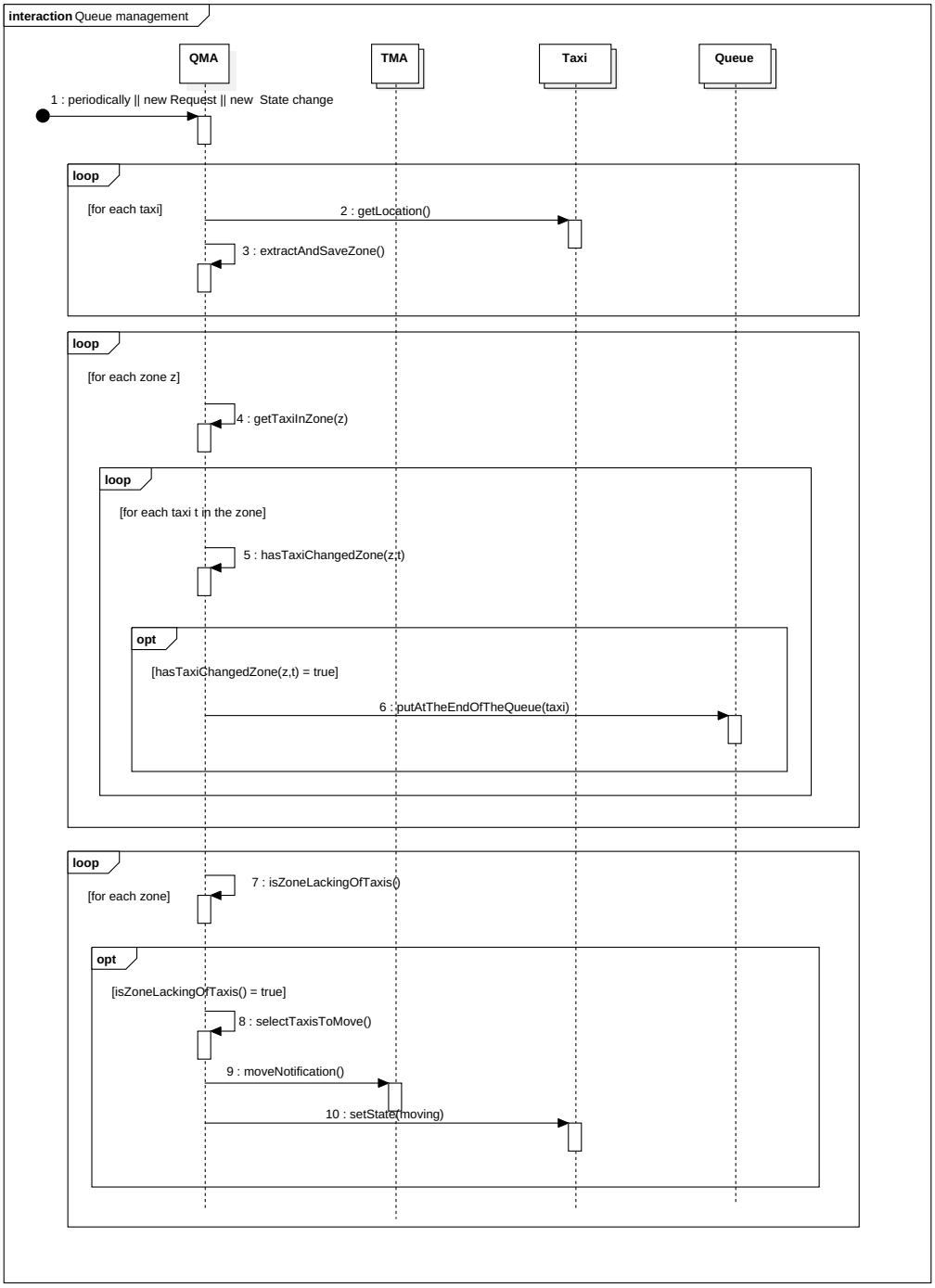


Figure 14: Queue Management - UML sequence diagram

3.4.10 Visualize request info

Name	Visualize request info
Related goals	[G2]
Actors	Passenger
Entry condition	Passenger has already made a request.
Flow of events	1. Passenger's application asks QMA for waiting time and number of incoming taxi. 2. QMA calculates the requested information. 3. QMA sends those information to passenger's application. 4. Passenger's application display the information to the passenger
Exit condition	The requested information are shown to the passenger
Exceptions	Nothing.
Special requirement	Nothing.

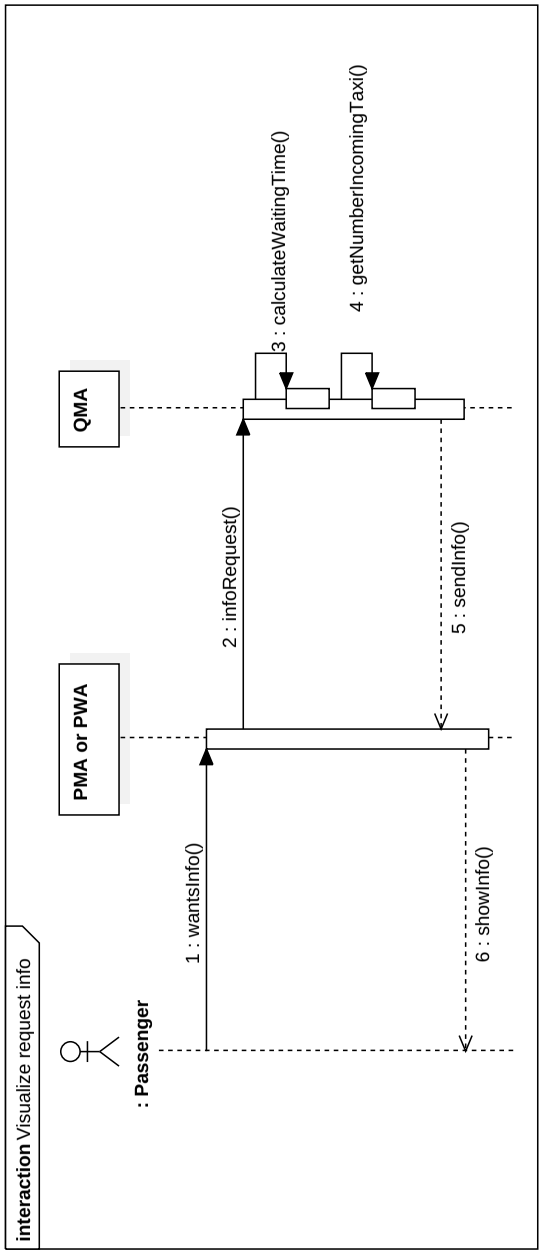


Figure 15: Visualize request info - UML sequence diagram

3.4.11 Request evaluation

Name	Request evaluation
Related goals	[G6]
Actors	Taxi driver
Entry condition	A new request is created by QMA.
Flow of events	1. QMA extracts the zone from which the request comes from. 2. The request is forwarded to the first taxi (or taxis according to the number of passengers) in the available taxis queue of the zone. 3. Taxi driver sees on TMA informations related to the request (localization and number of passengers). 4. See "Request confirmation" and "Request rejection".
Exit condition	See "Request confirmation" and "Request rejection".
Exceptions	• If the available taxi queue of the zone is empty, the operation is repeated considering the queues of the adjacent zones. • If no taxi is available at all, the request is put on hold until a taxi becomes available. • If no answer in one minute it is interpreted as a rejection.
Special requirement	Nothing.

3.4.12 Request confirmation

Name	Request confirmation→Request evaluation
Related goals	[G6]
Actors	Taxi driver
Entry condition	A new request is created by QMA.
Flow of events	1. QMA extracts the zone from which the request comes from. 2. The request is forwarded to the first taxi in the available taxis queue of the zone. 3. Taxi driver sees on TMA informations related to the request (localization and number of passengers). 4. Taxi driver confirms the request. 5. QMA sets taxi state as busy and removes it from the queue. 6. QMA computes the expected waiting time.
Exit condition	A confirmed request is created in the system and "Queue management" is performed.
Exceptions	• If waiting time cannot be computed the value is set to "unknown"
Special requirement	Nothing.

3.4.13 Request rejection

Name	Request rejection → Request evaluation
Related goals	[G6]
Actors	Taxi driver
Entry condition	A new request is created by QMA
Flow of events	1. QMA extracts the zone from which the request comes from. 2. The request is forwarded to the first taxi in the available taxis queue of the zone. 3. Taxi driver sees on TMA informations related to the request (localization and number of passengers). 4. Taxi driver rejects the request. 5. QMA put the taxi at the end of the queue. 6. QMA repeats the operation.
Exit condition	Nothing.
Exceptions	• Taxi driver is forbidden to reject twice the same request. In this case the request is intended confirmed.
Special requirement	Nothing.

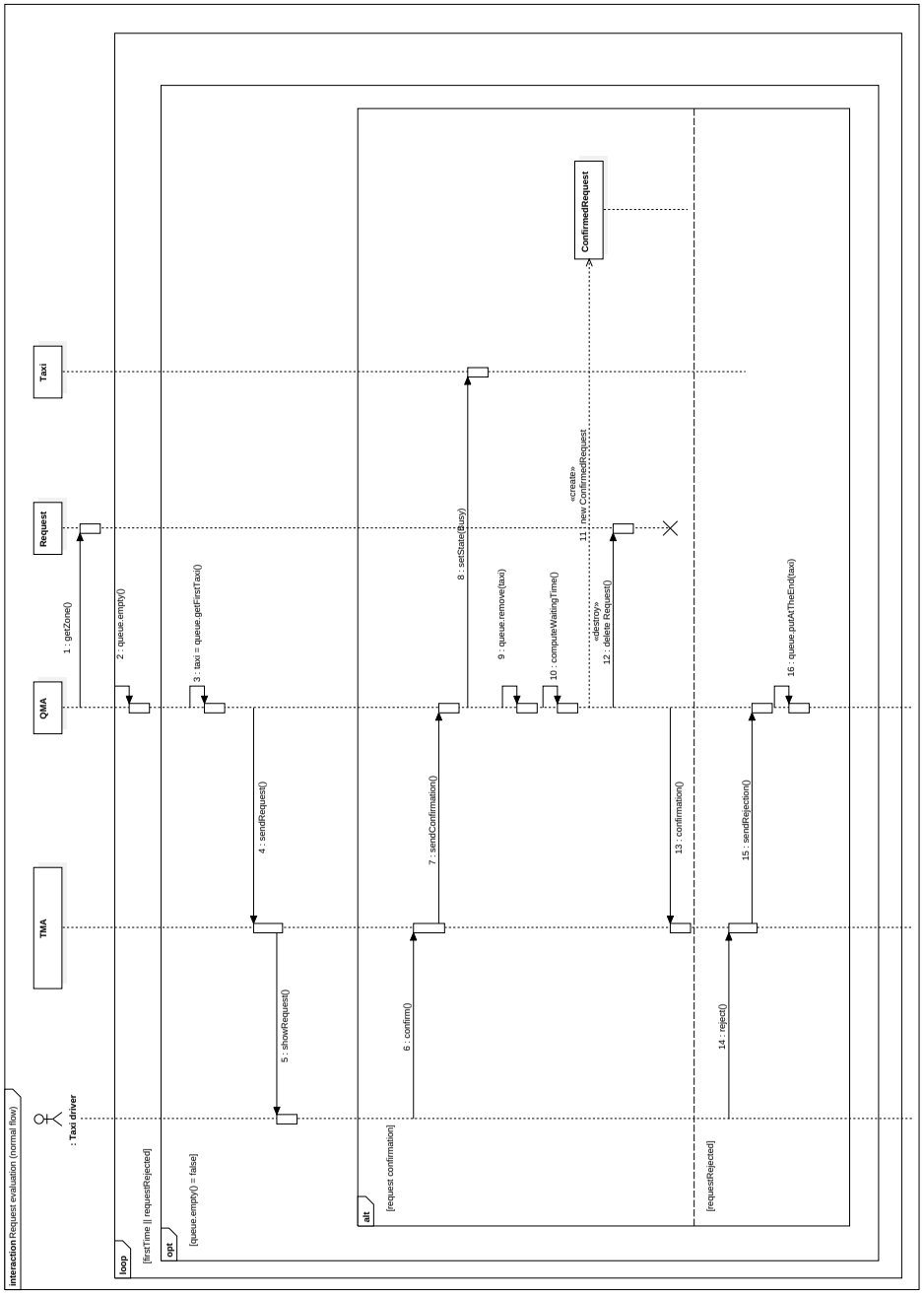


Figure 16: Request evaluation - UML sequence diagram

3.4.14 Inform about availability

Name	Inform about availability
Related goals	[G7]
Actors	Taxi driver
Entry condition	Nothing.
Flow of events	1. Taxi driver sets his new state by means of TMA. 2. The TMA sends the new state to TS system. 3. TS system check if the new state is reachable from the current state. 4. The TS system changes the taxi state. 5. A confirmation is sent to TMA.
Exit condition	The new taxi state is stored in TS.
Exceptions	• If the new state is invalid the state transition is prevented and an error message is shown to the taxi driver.
Special requirement	Nothing.

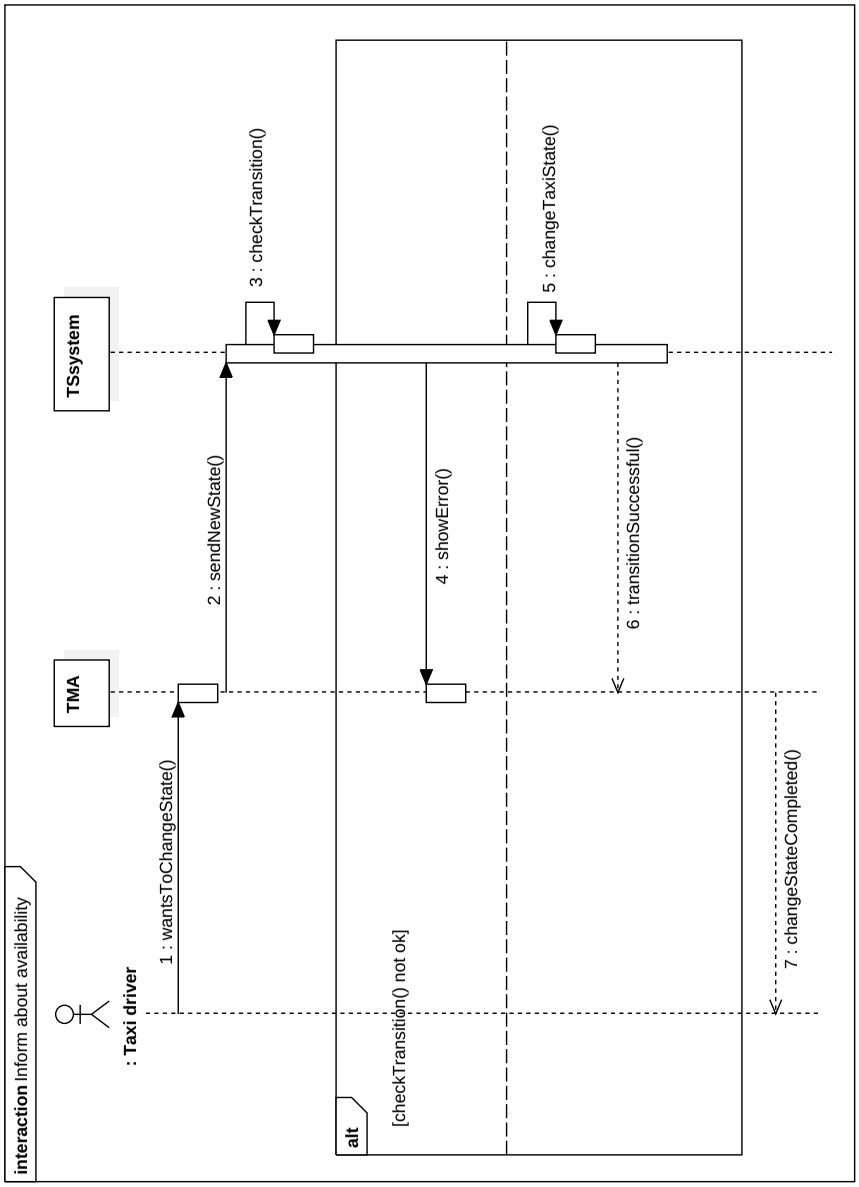


Figure 17: Inform about availability - UML sequence diagram

3.4.15 Insert call request

Name	Insert call request
Related goals	No goals related but needed for integration with old system.
Actors	Call center operator
Entry condition	A call center operator receives a call by a passenger.
Flow of events	1. The call center operator asks the passenger for address and number of passengers. 2. Call center operator uses PWA to create a request. 3. Call center operator informs the passenger about the number of the incoming taxi and the waiting time.
Exit condition	A new request is created.
Exceptions	• If, for some reason, the request submission fail, the call center operator report the error to the passenger.
Special requirement	Request and Visualize request info are performed.

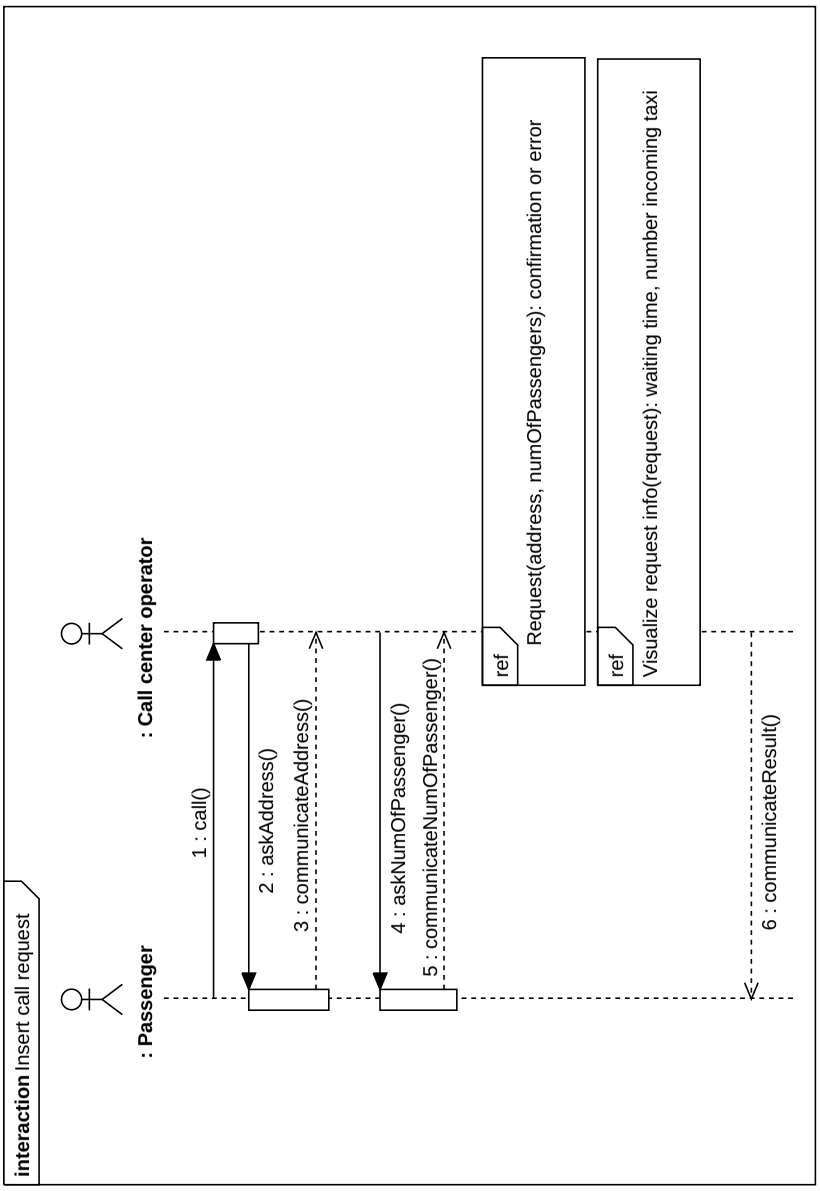


Figure 18: Insert call request - UML sequence diagram

3.5 Other UML diagrams

3.5.1 Class diagram

In this subsection a conceptual UML Class Diagram is shown.

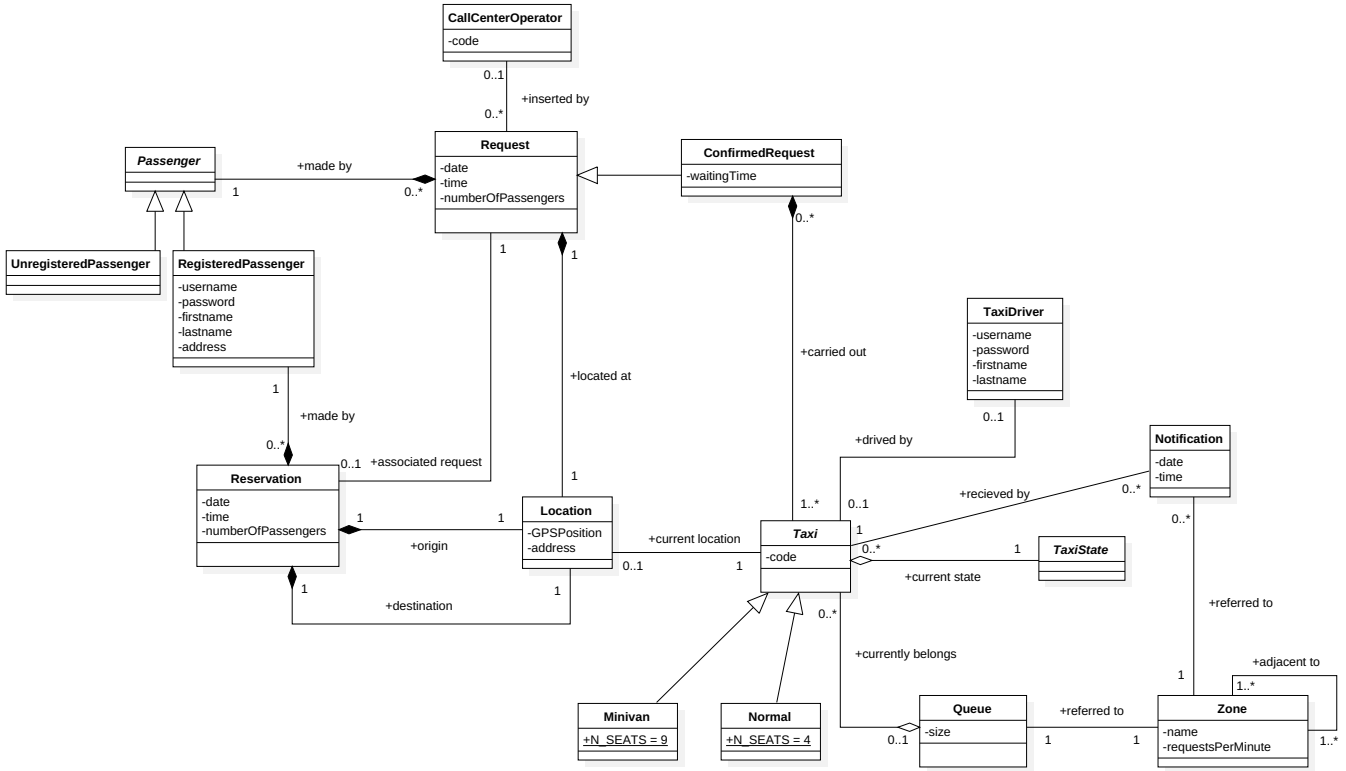


Figure 19: UML Class Diagram

3.5.2 State chart diagram

In this subsection a state chart diagram is shown to make clear how the state of a taxi changes according to the events occuring.

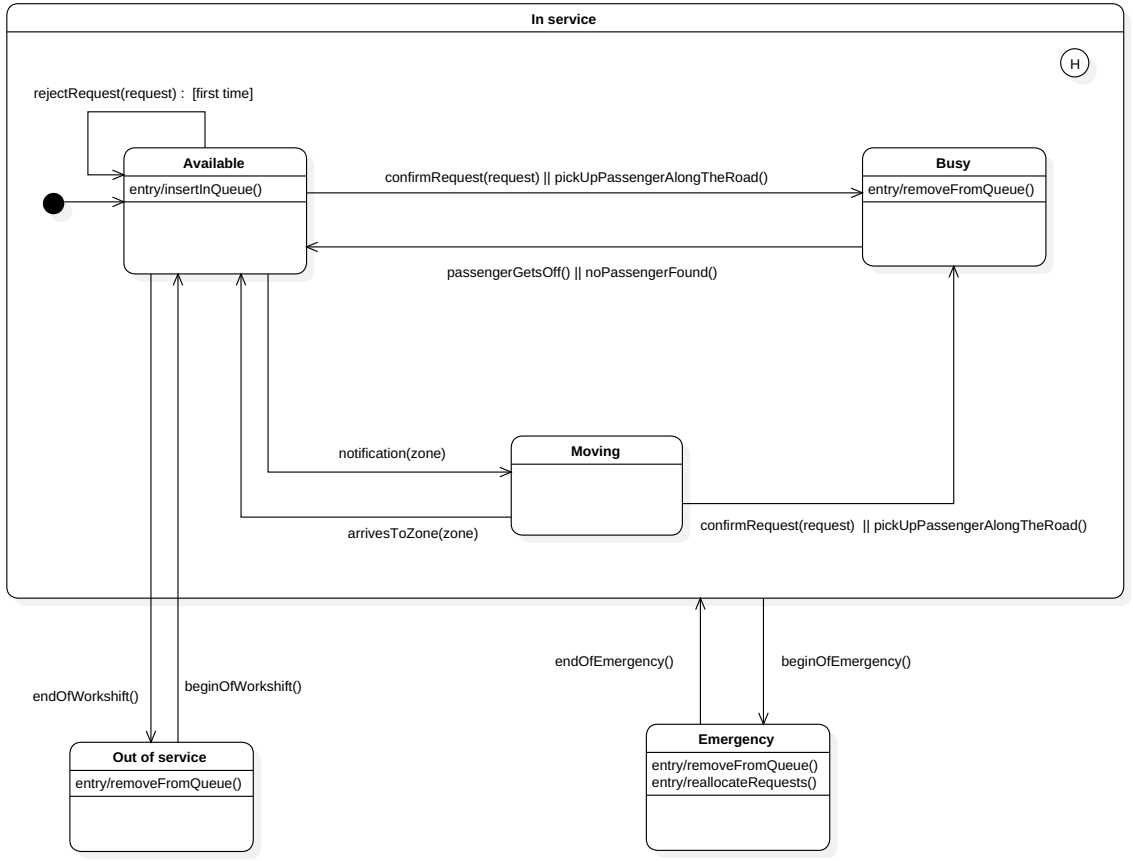


Figure 20: Taxi state - UML State Chart diagram

3.5.3 Activity diagram

Since in the previous section for the request evaluation only a normal flow has been represented, here we show an activity diagram in order to clarify how the general flow (normal flow and exceptional flow) of that use case works.

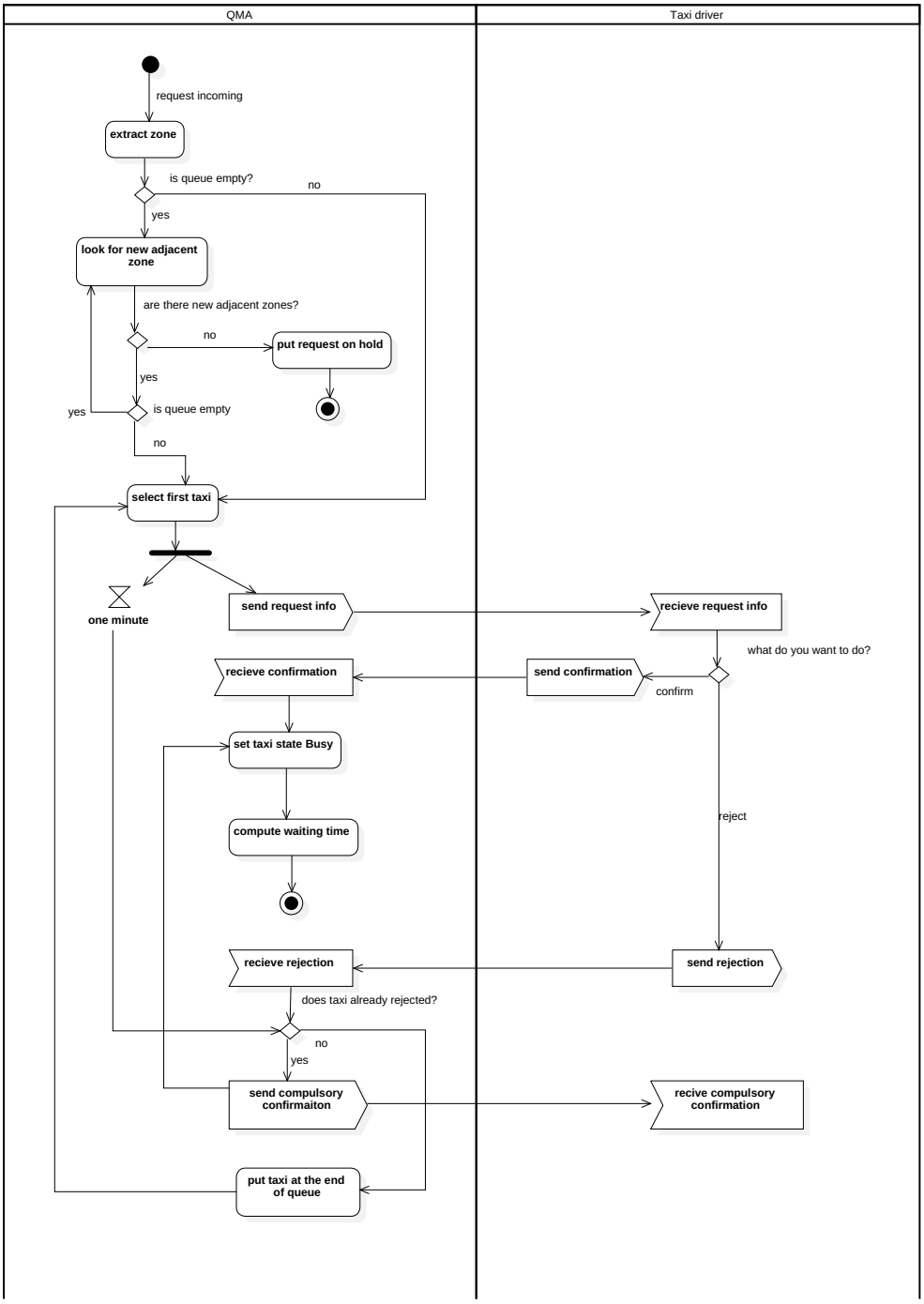


Figure 21: Request evaluation - UML Activity diagram

3.6 Non functional requirements

Since TS system is intended to increase the degree of usability of the taxi service and perform improvements in the available taxi queue management, according to stackeholders' expectations we identified the following non functional requirements.

3.6.1 Performance

- All internal elaborations carried out by QMA must have a response time of 100 ms in 99% of times.
- TMA, PMA and PWA must have a response time of 10 ms in 99% of times for local elaborations (not involving Internet connection).
- Remote interaction (via Internet) within TS system must least at most 2 seconds otherwise the connection is closed.
- TS system shall support at least 500 simultaneous connections.

3.6.2 Reliability

- TS system shall ensure that every operation coming from TMA, PMA and PWA is correctly processed.
- All the user inputs are verified by the local application before being sent to the TS system .
- TS system shall be a transactional system (both operations generated by users and internal operations carried out by TS system are transactions).

3.6.3 Availability

- The TS system must be available 99.9%.
- A hardware duplicate must be ready to be activated when maintenance is performed or failure occurs to ensure continuity of the service.
- A backup of data should be performed every week to prevent any loss of data in the TS system.

3.6.4 Security

- All sensitive data collected from users must be stored in safe devices accessible only by authorized users to prevent loss, manipulation or stealing.
- All data flowing between system and users must be encrypted to ensure confidentiality.
- Registered passenger's password must be changed every three months.

3.6.5 Maintainability

- TS system shall be developed in order to reduce the maintenance cost, trying to anticipate and preventing future interventions (corrective and adaptive maintenance) by means of standard patterns (in particular the architecture must follow the MVC pattern).
- TS system shall expose a set of APIs in order to allow future functionalists extensions (perfective maintenance).
- Thanks to hardware duplication, hardware maintenance interventions will be performed without service interruption.

3.6.6 Portability

- TMA and PMA shall be installable in both iOS, Android and Windows mobile.
- PWA shall be accessible by means of all up to date browsers having cookies enabled.

3.6.7 Documentation

- An accurate user guide must be available online.
- All the documents produced during the design and development of the TS shall be available for specialized personnel.

3.6.8 User interface and human factors

- 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.
- Every functionality shall be reached surfing no more than 4 pages.

4 Alloy model

In this section we provide an alloy model trying to capture the main features of the entities affected by the TS system. The model is built defining signatures according to the class diagram and specifying the constraints that comes out from the requirements.

4.1 General model

4.1.1 Signatures, facts and functions

```
module myTaxiService/model

//Auxiliary signatures
sig Date{}
sig Time{}

sig Location{
        zone: one Zone,
}

abstract sig Passenger{}
sig UnregisteredPassenger extends Passenger{}
sig RegisteredPassenger extends Passenger{}

abstract sig TaxiState{}
one sig OutOfService,Emergency extends TaxiState{}
one sig Available, Busy, Moving extends TaxiState{}

abstract sig Taxi {
        driver: lone TaxiDriver,
        state: one TaxiState,
        numberOfSeats: one Int,
}

sig MinivanTaxi extends Taxi {} {numberOfSeats = 9}
sig NormalTaxi extends Taxi {} {numberOfSeats = 4}

sig TaxiDriver{}

sig Request{
        passenger: one Passenger,
        date: one Date,
        time: one Time,
        numberOfPassengers: one Int,
        location: one Location,
}  {numberOfPassengers>=1}

sig ConfirmedRequest extends Request{
        waitingTime: one Time,
        taxis: some Taxi,
}

sig Reservation{
        passenger: one RegisteredPassenger,
        date: one Date,
        time: one Time,
        numberOfPassengers: one Int,
        origin: one Location,
```

```
        destination: one Location,
        associatedRequest: one Request,
} {numberOfPassengers>=1 and associatedRequest.passenger = passenger and
    associatedRequest.location = origin and associatedRequest.
    numberOfPassengers = numberOfPassengers}

sig Zone{
        requestsPerMinute: one Int,
}{ requestsPerMinute>0}

//A taxi driver drives one taxi at time
fact oneTaxiForEachTaxiDriver {
        all disj t1, t2 : Taxi | t1.driver != t2.driver
}

//Non out of service taxis must have a driver
fact allAvailableTaxiMustHaveADriver {
        all t: Taxi | t.state in (Available + Moving + Busy) implies one t.
            driver
}

//All busy taxi must have at least a request associated
fact allBusyTaxiMustHaveAtLeastARequestAssociated {
        all t: Taxi | some r: Request | t.state in Busy implies t in r.
            taxis
}

//Auxiliary predicate to check if two requests are simultaneous
pred atTheSameTime[r1,r2 : Request]{
        r1.time = r2.time and r1.date = r2.date
}

//Each passenger can make at most one request at time
fact noPassengerMakesMoreThanOneRequestAtTime {
        all disj r1,r2 : Request | atTheSameTime[r1,r2]
                implies r1.passenger != r2.passenger
}

//Each taxi can carry out at most one request at time
fact noTaxiCarriesOutMoreThanOneRequestAtTime {
        all disj r1, r2: ConfirmedRequest | atTheSameTime[r1,r2]
                implies r1.taxis != r2.taxis
}

//Each request can be associated to at most one reservation
fact eachRequestAssociatedToAtMostOneReservation {
        all r: Request | lone re: Reservation |
                re.associatedRequest = r
}

//Origin and destination for each request must be different
fact originAndDestinationDifferent {
        all r: Reservation | r.origin != r.destination
}

//In each request the number of seats must be sufficient wrt number of
    passengers
fact numberOfSeatsSufficient {
        all r: ConfirmedRequest | sum r.taxis.numberOfSeats
                >= r.numberOfPassengers
```

```
}

/*
This is syntactically and semantically correct but it is writter in second
    order logic so it cannot be executed
fact numberOfSeatsAreTheMinimumRequired {
        all r: ConfirmedRequest | no taxiSubset: set Taxi | taxiSubset in r
            .taxis and taxiSubset != r.taxis and                sum
            taxiSubset.numberOfSeats >= r.numberOfPassengers
}
*/

//The same fact rephrased in FOL: the number of taxis sent are the minimun
    requested to pick up all passengers
fact numberOfSeatsAreTheMinimumRequired {
        all r: ConfirmedRequest | no t: Taxi | t in r.taxis and sum r.taxis
            .numberOfSeats - t.numberOfSeats >= r.numberOfPassengers
}

//Returns the number of in service taxis
fun numberOfInServiceTaxis: Int {
        #state.Available + #state.Busy + #state.Moving
}

//At least 20% of total number of taxis must be in service
fact atLeast20PerCentOfTaxisAreNotOutOfService {
        mul[numberOfInServiceTaxis[],2] >= #Taxi
}
```

4.1.2 Predicates

```
//Just builds a world satisfying constraints
pred show{}

run show for 4 but 5 Int, exactly 1 Zone, 1 Reservation, 3 Request
```

Executing "Run show for 4 but 5 int, exactly 1 Zone, exactly 1 Reservation, exactly 3 Request" Solver=sat4j Bitwidth=5 MaxSeq=4 SkolemDepth=1 Symmetry=20 6397 vars. 446 primary vars. 16955 clauses. 16ms. Instance found. Predicate is consistent. 15ms.

```
//Predicate for sending a request: if the request is not already confirmed
    and it is not already in the set it is added to the set
pred sendRequest[setOfRequests,setOfRequests': set Request, request:
    Request] {
        no ((ConfirmedRequest + setOfRequests) & request)  implies
                setOfRequests' = setOfRequests + request
        else
                setOfRequests' = setOfRequests  }

run sendRequest for 10 but 5 Int, 1 Zone, 0 Reservation, exactly 4 Request,
     1 ConfirmedRequest, 2 Passenger, 3 Taxi
```

Executing "Run sendRequest for 10 but 5 Int, 1 Zone, 0 Reservation, exactly 4 Request, 1 ConfirmedRequest, 2 Passenger, 3 Taxi" Solver=sat4j Bitwidth=5 MaxSeq=10 SkolemDepth=1 Symmetry=20 14458 vars. 841 primary vars. 37281 clauses. 31ms. Instance found. Predicate is consistent. 63ms.

```
//Predicate for sending a reservation: if the reservation is not already in
    the set it is added  to the set and a corresponding request is
    generated
pred sendReservation[setOfReservations,setOfReservations': set Reservation,
    reservation: Reservation] {
        no (setOfReservations & reservation) implies (
                setOfReservations' = setOfReservations + reservation and
                one request: Request | reservation.associatedRequest =
                    request )
        else            setOfReservations' = setOfReservations
}

run sendReservation for 10 but 5 Int, 1 Zone, exactly 3 Reservation, 3
    Request, 3 Passenger, 3 Taxi
```

Executing "Run sendReservation for 10 but 5 Int, 1 Zone, exactly 3 Reservation, 3 Request, 3 Passenger, 3 Taxi" Solver=sat4j Bitwidth=5 MaxSeq=10 SkolemDepth=1 Symmetry=20 9021 vars. 706 primary vars. 22718 clauses. 32ms. Instance found. Predicate is consistent. 15ms.

```
//Predicate for deleting a reservation: if reservation is present it is
    removed and the corresponding request no longer exists
pred cancelReservation[setOfReservations,setOfReservations': set
    Reservation, reservation: Reservation]{
        one setOfReservations & reservation implies
                setOfReservations' = setOfReservations - reservation
        else
                setOfReservations' = setOfReservations
}

run cancelReservation for 10 but 5 Int, 1 Zone, exactly 2 Reservation, 2
    Request, 2 Passenger, 3 Taxi
```

Executing "Run cancelReservation for 10 but 5 Int, 1 Zone, exactly 2 Reservation, 2 Request, 2 Passenger, 3 Taxi" Solver=sat4j Bitwidth=5 MaxSeq=10 SkolemDepth=1 Symmetry=20 9023 vars. 706 primary vars. 22712 clauses. 0ms. Instance found. Predicate is consistent. 15ms.

4.1.3 Assertions

```
//Verifies whether at least one taxi is in service
assert ifOneTaxiExistsItIsInService {
        one Taxi implies Taxi.state in (Available + Busy + Moving)
}

check ifOneTaxiExistsItIsInService
```

Executing "Check ifOneTaxiExistsItIsInService for 10" Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20 49010 vars. 2308 primary vars. 122421 clauses. 172ms. No counterexample found. Assertion may be valid. 15ms.

```
//Verifies if number of seats is the minimun necessary
assert numberOfSeatsAreTheMinimunNecessary {
        all r: ConfirmedRequest | r.numberOfPassengers <= sum r.taxis.
            numberOfSeats and r.numberOfPassengers <= plus[sum r.taxis.
            numberOfSeats ,MinivanTaxi.numberOfPassengers]
}

check numberOfSeatsAreTheMinimunNecessary for 10
```

Executing "Check numberOfSeatsAreTheMinimunNecessary for 10" Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20 49755 vars. 2318 primary vars. 124998 clauses. 109ms. No counterexample found. Assertion may be valid. 313ms.

```
//Request carried out by the same taxi driver must be different in date or
    time
assert allRequestOfTheSameTaxiDriverDifferentInTime {
        all td: TaxiDriver | all disj r1,r2: ConfirmedRequest | td in (r1.
            taxis.driver & r2.taxis.driver) implies not atTheSameTime[r1,r2
            ]
}

check allRequestOfTheSameTaxiDriverDifferentInTime for 10
```

Executing "Check allRequestOfTheSameTaxiDriverDifferentInTime for 10" Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20 50129 vars. 2338 primary vars. 125356 clauses. 109ms. No counterexample found. Assertion may be valid. 219ms.

```
//Each reservation has a request associated with the same origin
assert eachReservationHasARequestWithSameOrigin {
        all r: Reservation | some re: Request | r.origin = re.location
}

check eachReservationHasARequestWithSameOrigin for 10
```

Executing "Check eachReservationHasARequestWithSameOrigin for 10" Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20 53009 vars. 2420 primary vars. 133925 clauses. 391ms. No counterexample found. Assertion may be valid. 224ms.

Figure 22 is a world generated by predicate **pred** `show`. It is shown in order to make the reader understand that the model is consistent, since it admits at least one instance satisfying the constraints. In the following pages other worlds corresponding to the execution of the predicates will be shown.

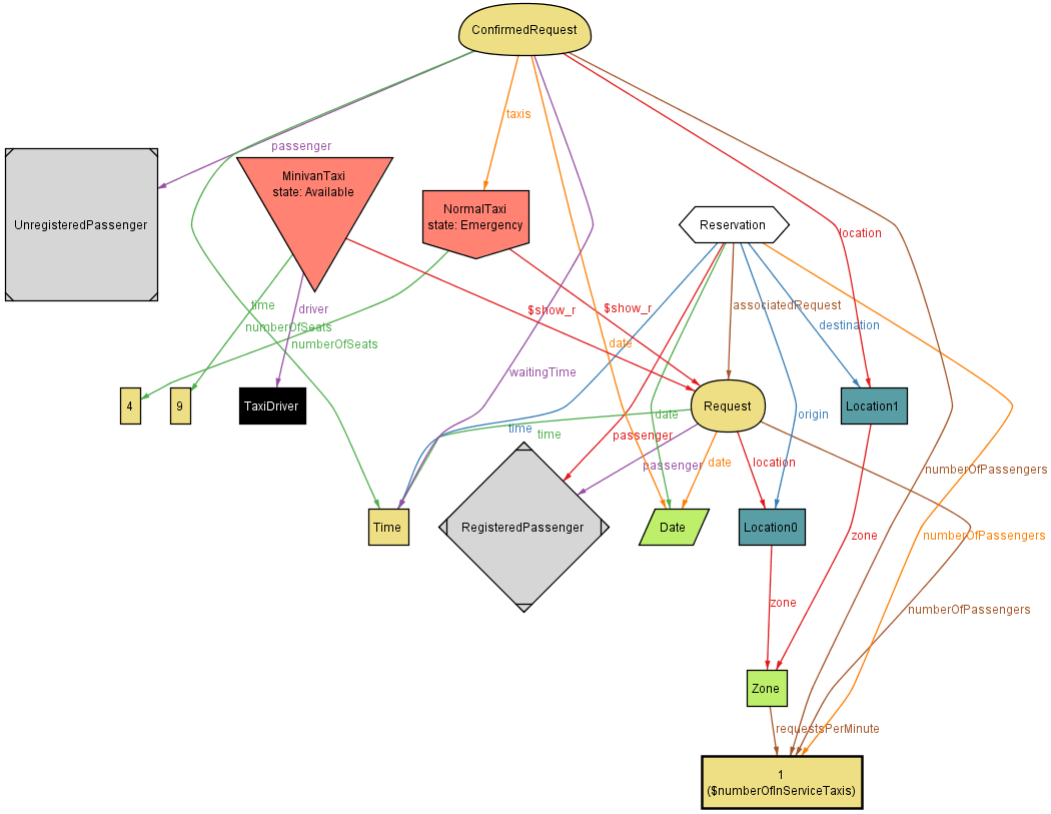


Figure 22: World generated by predicate **pred** `show`

Figure 23 is a world generated by predicate **pred** `sendRequest[setOfRequests,setOfRequests': set Request, request: Request]`. Before inserting Request2 the set of requests is composed of only ConfirmedRequest; after the insertion also Request2 is in the set.

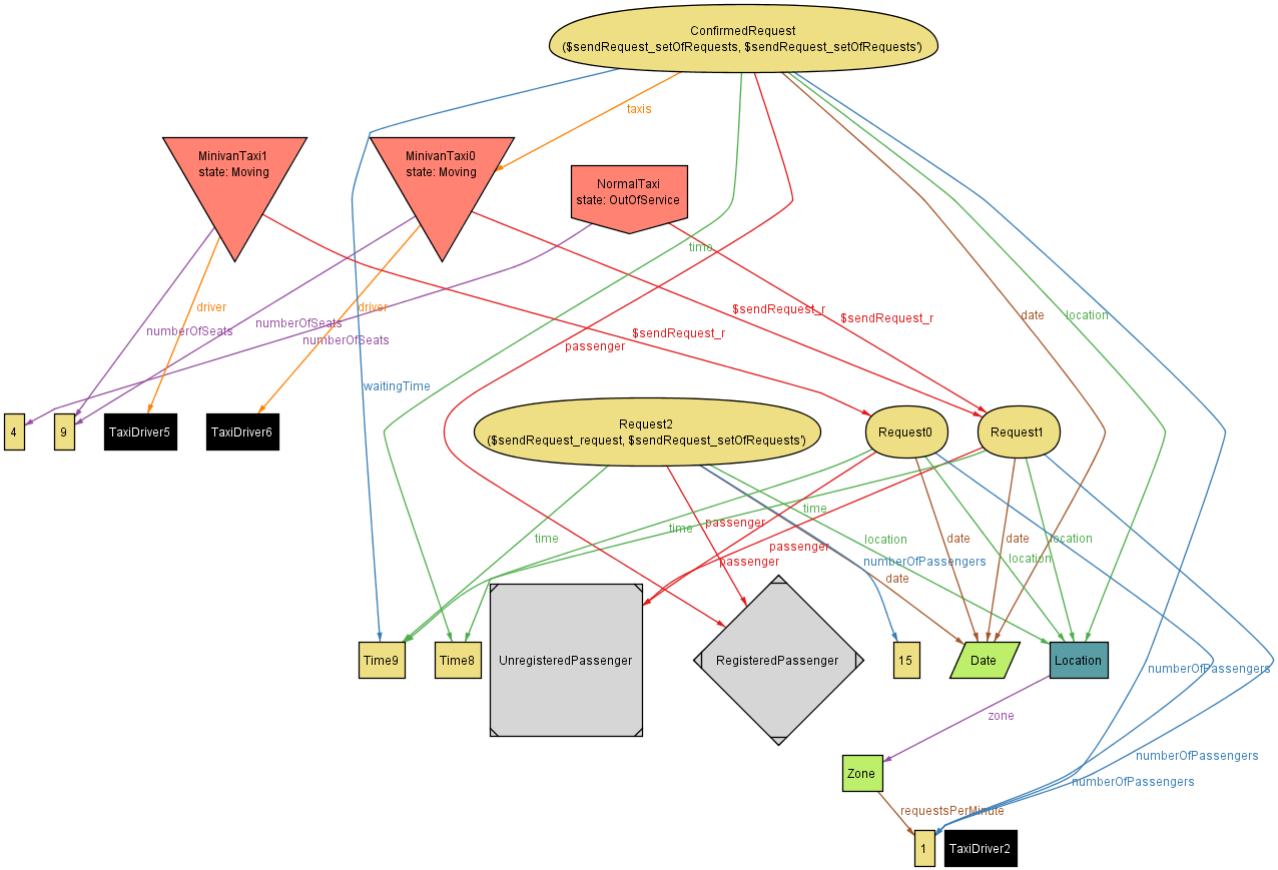


Figure 23: World generated by predicate **pred** `sendRequest`

Figure 23 is a world generated by predicate **pred** `sendReservation[setOfReservations,setOfReservations': set Reservation, reservation: Reservation]`. It is clear that before the execution the set of reservation contains only Reservation1 and after it will contain also Reservation2 (Reservation0 does not belong to any of them).

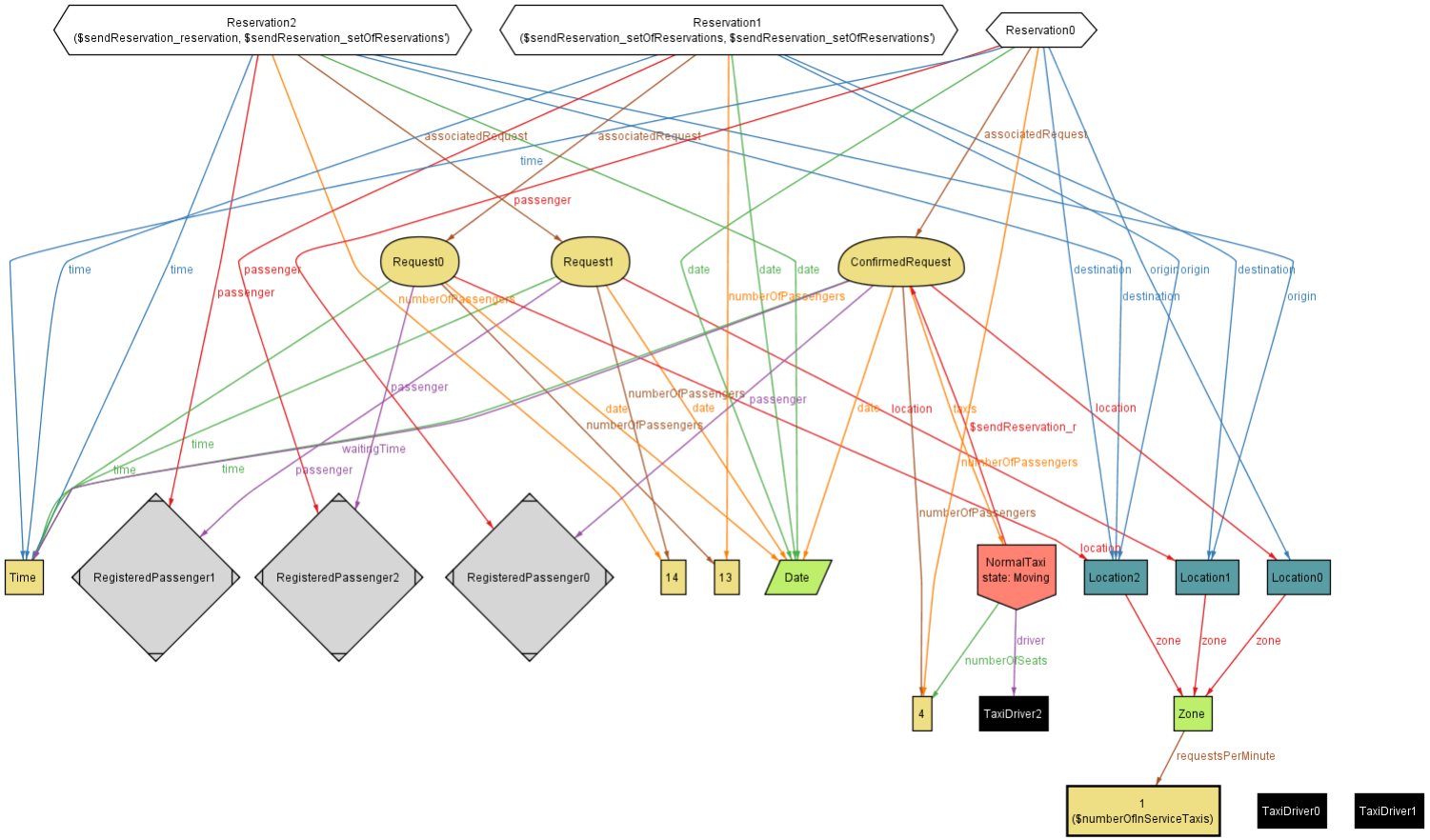


Figure 24: World generated by predicate **pred** `sendReservation`

Figure 23 is a world generated by predicate **pred** `cancelReservation[setOfReservations,setOfReservations': set Reservation, reservation: Reservation]` . As you can see, before the execution Reservation1 belonged to the set of reservation while reservation set is empty.

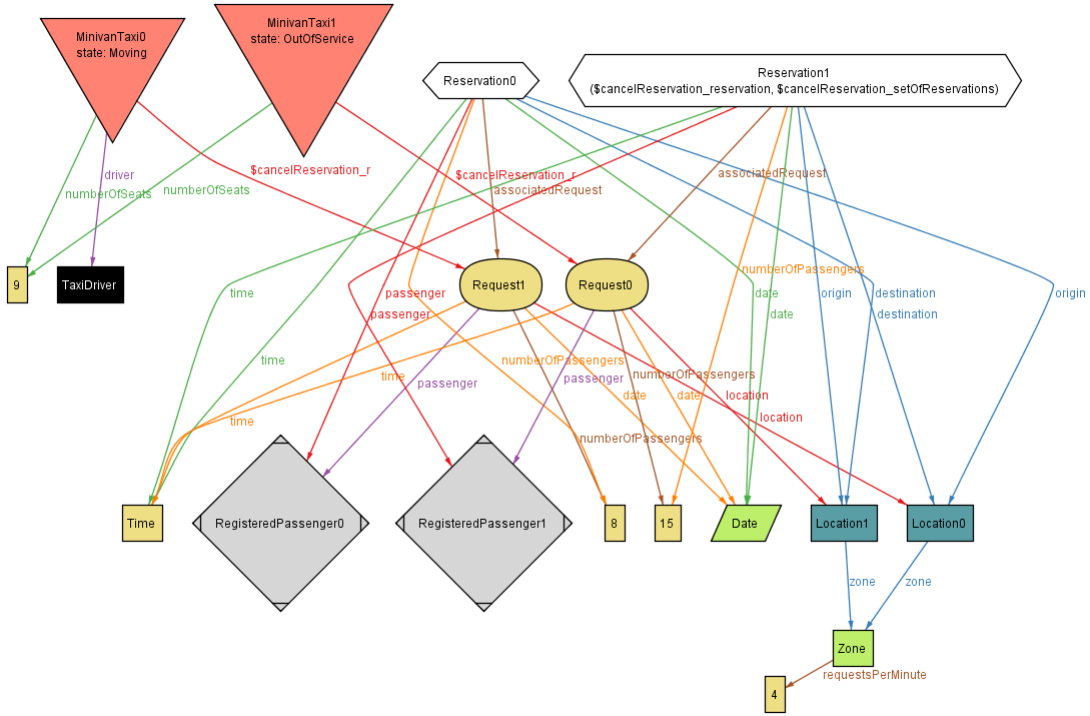


Figure 25: World generated by predicate **pred** `cancelReservation`

4.2 Queue management model

Since queue management is a relevant part of the TS system, in this subsection we model the structure of a queue and the adjacency relation between zones. We will focus on the important constraints imposed by the system but we will not model the dynamical behavior of the queues.

4.2.1 Signatures, facts and functions

```
module myTaxiService/queue

abstract sig TaxiState{}
one sig OutOfService,Emergency extends TaxiState{}
one sig Available, Busy, Moving extends TaxiState{}

sig Taxi {
        state: one TaxiState,
}

sig Zone{
        queue: one Queue,
        adjacentZones: some Zone,
}

//Queue definition
sig Queue {
        root: lone Node
}

sig Node {
        taxi: one Taxi,
        next: lone Node
}

//Structural properties
fact queueStructuralProperties {

        //Each node belongs to exactly one queue
        all n: Node | one q: Queue | n in q.root.*next

        //No cycles
        no n: Node | n in n.^next
}

//Each queue must belong to exactly one zone
fact eachQueueBelongsToExactlyOneZone {
        all q: Queue | one z: Zone | q in z.queue
}

//Adjacency relation between zones is simmetric but not reflexive
fact adjacencySimmetricButNotReflexive
{
        adjacentZones in ~adjacentZones
        no adjacentZones & iden
}

//Returns the set of taxis belonging to the queue q
fun getTaxisFromQueue[q: Queue] : set Taxi {
        q.root.*next.taxi
}
```

```
52
//Queues must store only available taxis
54 fact allTaxisInQueueAreAvailable {
        all q: Queue | getTaxisFromQueue[q].state in Available
56 }

58 //Each available taxi belongs to exactly one node
        fact eachTaxiBelongsToExactlyOneNode {
60      all t: Taxi | t.state in Available implies (one n: Node | n.taxi =
            t)
}
```

4.2.2 Predicates

```
1 //Builds a realistic world
pred showQueues{
3       some q: Queue | #getTaxisFromQueue[q]>3
        some t: Taxi | t.state in OutOfService
5       some t: Taxi | t.state in Busy
}
7
run showQueues for 10 but exactly 4 Zone, exactly 10 Taxi
```

> Executing "Run show for 5 but exactly 4 Zone, exactly 10 Taxi" Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20 3031 vars. 176 primary vars. 6646 clauses. 15ms. Instance found. Predicate is consistent. 16ms.

4.2.3 Assertions

```
//No available taxi are not present any queue
2 assert noAvailableTaxiNotInQueue {
        no t: Taxi | t.state in Available and (no q:Queue | t in
            getTaxisFromQueue[q])
4 }

6 check noAvailableTaxiNotInQueue for 10
```

> Executing "Check noAvailableTaxiNotInQueue for 10" Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20 14039 vars. 580 primary vars. 36827 clauses. 63ms. No counterexample found. Assertion may be valid. 31ms.

```
//There are no taxi that appear in more than one queue
2 assert noTaxiSharedBetweenQueus {
        all disj q1,q2: Queue | no getTaxisFromQueue[q1] &
            getTaxisFromQueue[q2]
4 }

6 check noTaxiSharedBetweenQueus for 10
```

> Executing "Check noTaxiSharedBetweenQueus for 10" Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20 14642 vars. 590 primary vars. 37985 clauses. 47ms. No counterexample found. Assertion may be valid. 531ms.

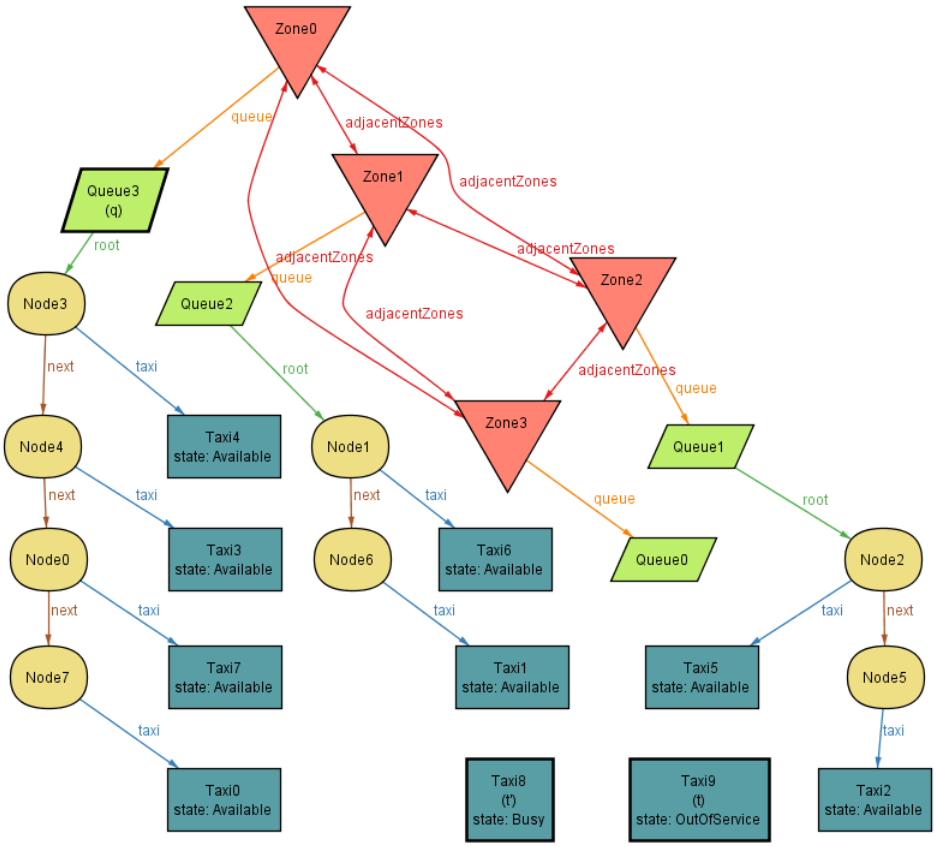


Figure 26: World generated by the predicate **pred** `showQueues`

4.3 Considerations about Alloy

The process of modeling a complex system is always prone to many different kinds of errors. Even though you have correctly understood the requirements several conditions often neglected because they seem to be obvious, but they are not. Alloy is a very precious tool that allows requirement engineer to understand and overcome those deficiencies in the model.

A i* modeling

i* modeling

The i* framework was developed for modelling and reasoning about organizational environments and their information systems [2]. It consists of two main modelling components. TheStrategic Dependency (SD) model is used to describe the dependency relationships among various actors in an organizational context. The Strategic Rationale (SR) model is used to describe stakeholder interests and concerns, and how they might be addressed by various configurations of systems and environments [1]. We think that such a model can be a useful instrument for requirement engineering because it is able to highlight relationships between different levels of abstraction showing how *tasks* cooperates to fullfil *goals* and how non functional requirements (known ad *soft goals* in the model) are affacted. In this section we provide a simple model (Strategic Rationale model) of a part of the TS system by using i* notation.

For further details and a exhaustive description of the notation refer to:

[1] E. Yu, Modelling Strategic Relationships for Process Reengineering, Ph.D. thesis, also Tech. Report DKBS-TR- 94-6, Dept. of Computer Science, University of Toronto, 1995.

[2] E. Yu 'Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering' Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97) Jan. 6-8, 1997, Washington D.C., USA. pp. 226-235.

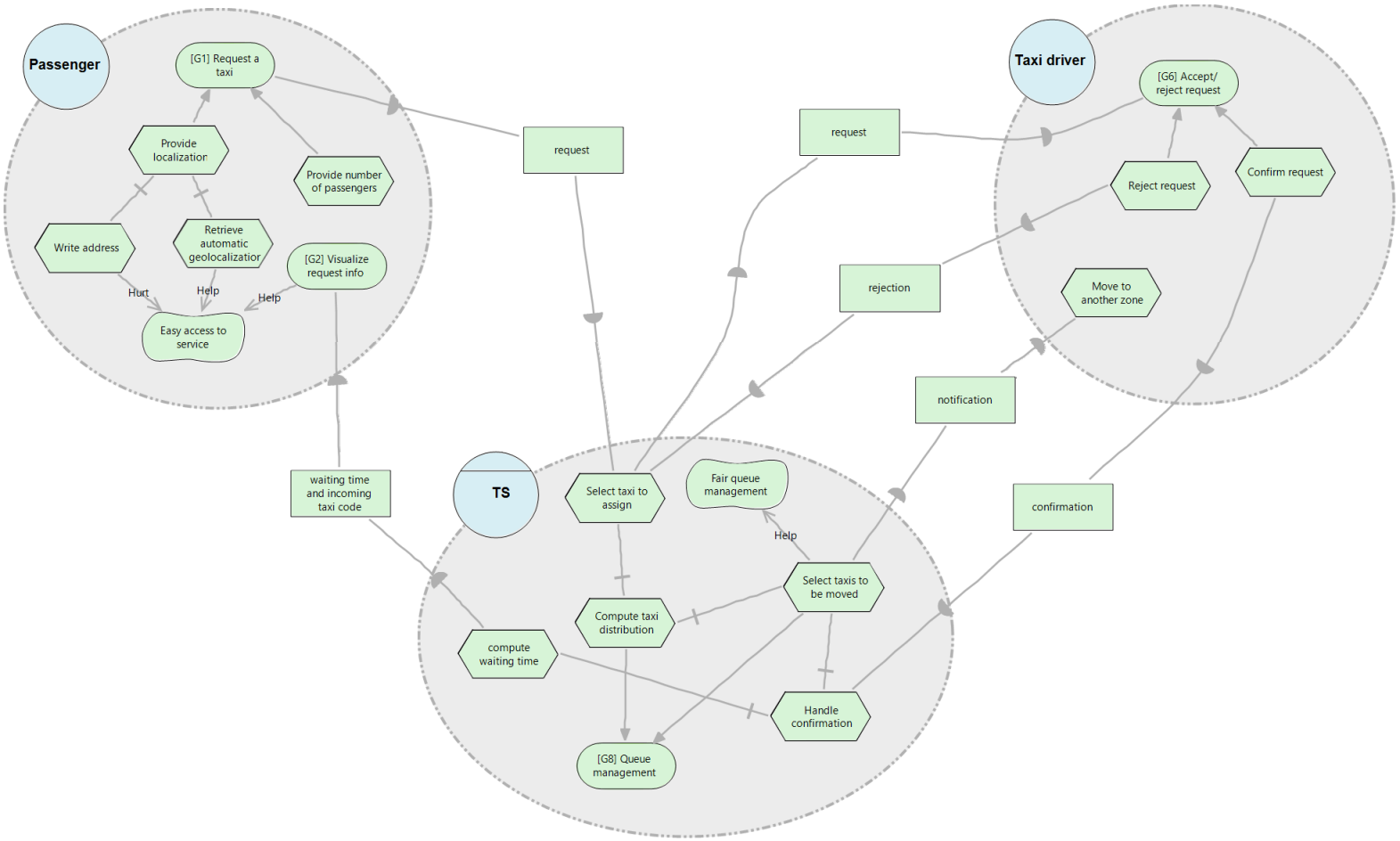


Figure 27: i* model

B Appendix

Used tools

1. LyX visual editor for LaTeX (`http://www.lyx.org/`) to write this document.
2. Star UML (`http://staruml.io/`) for use case diagram, class diagram, sequence diagram, activity diagram and state chart diagram.
3. Alloy Analyser 4.2 (`http://alloy.mit.edu/alloy/`) to build the model and prove its consistency.
4. Balsamiq Mockup (`http://balsamiq.com/products/mockups/`) for user inferface mockup generation.
5. OpenOME (`http://www.cs.toronto.edu/km/openome/OpenOME.html`), an open-source requirements engineering tool for the i* model.

Hours of works

Time spent by each group member:

- Alberto Maria Metelli: 33 h
- Riccardo Mologni: 33 h

Revision history

Version	*Date*	*Revision description*	*Revision notes*
0.1	1-11-2015	Initial draft	-
1.0	6-11-2015	Final draft	-
1.1	10-11-2015	Revision on final draft	Fixed section 3.4.6
1.2	04-12-2015	Revision before DD release	Fixed reference documents and class diagram.
2.0	22-2-2016	Final release	Fixed introduction and some terminology.

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^{TM}-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.

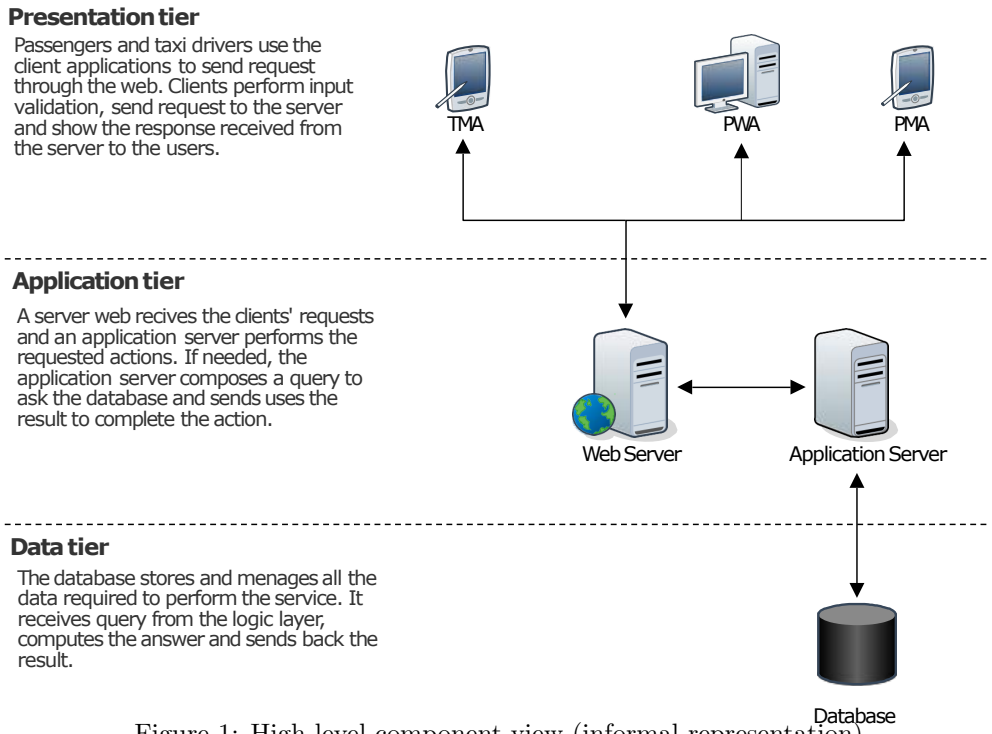


Figure 1: High level component view (informal representation)

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.

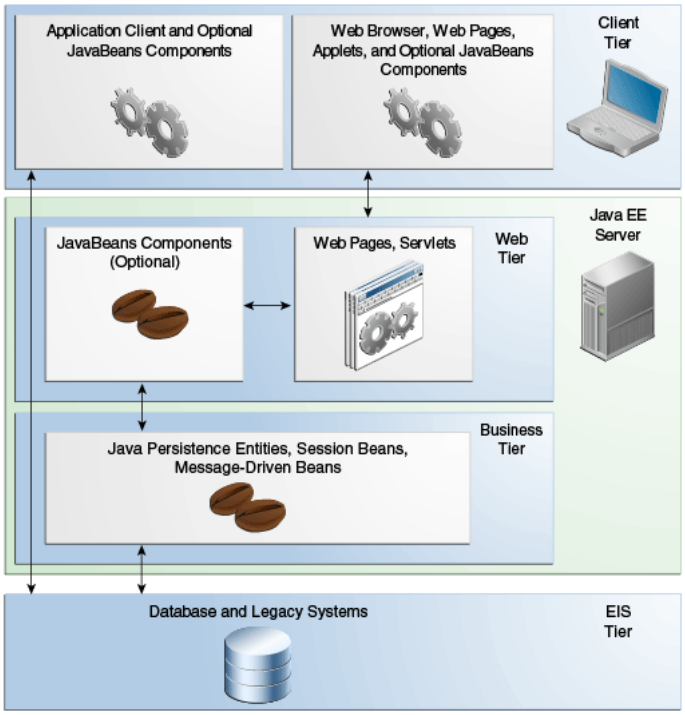


Figure 2: JEE architecture

[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].

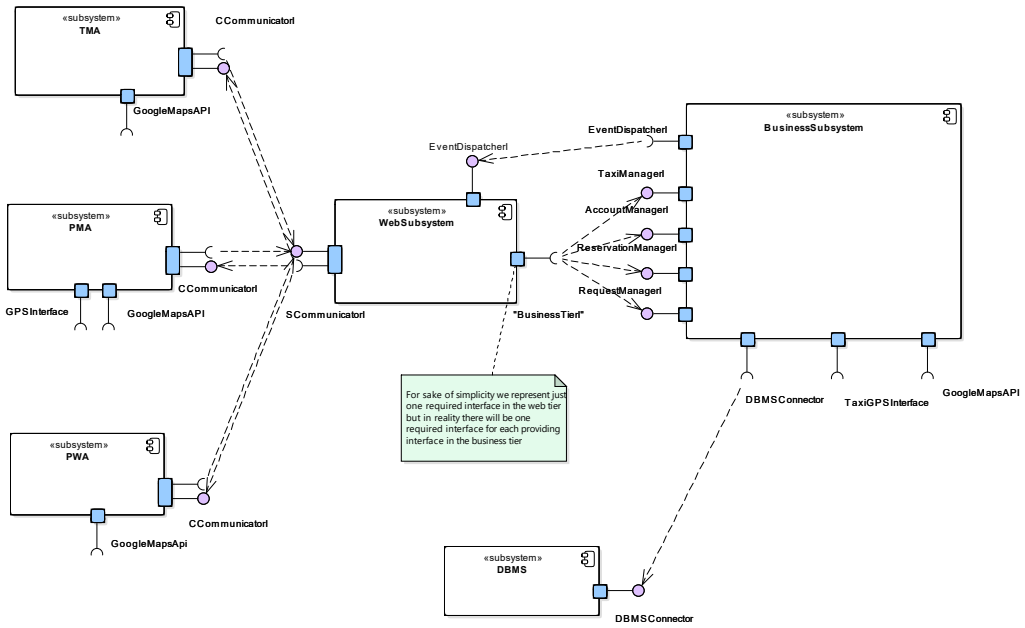


Figure 3: UML "high level" component diagram

[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

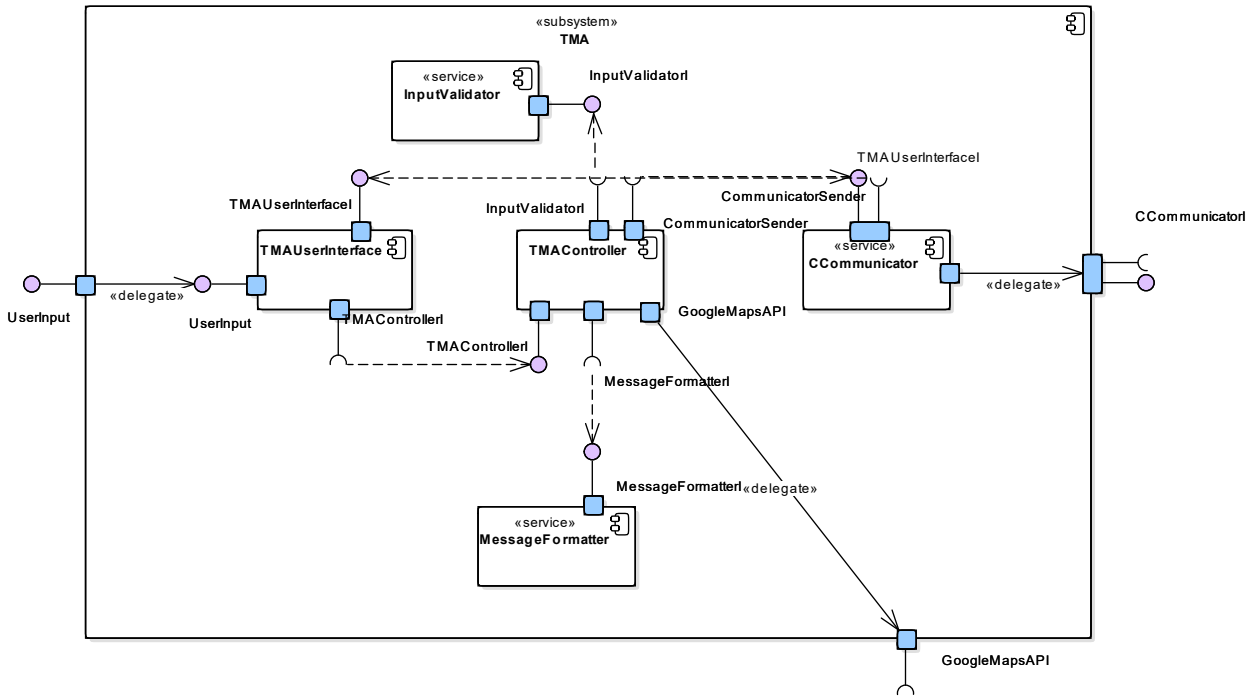


Figure 4: UML component diagram - TMA

- *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

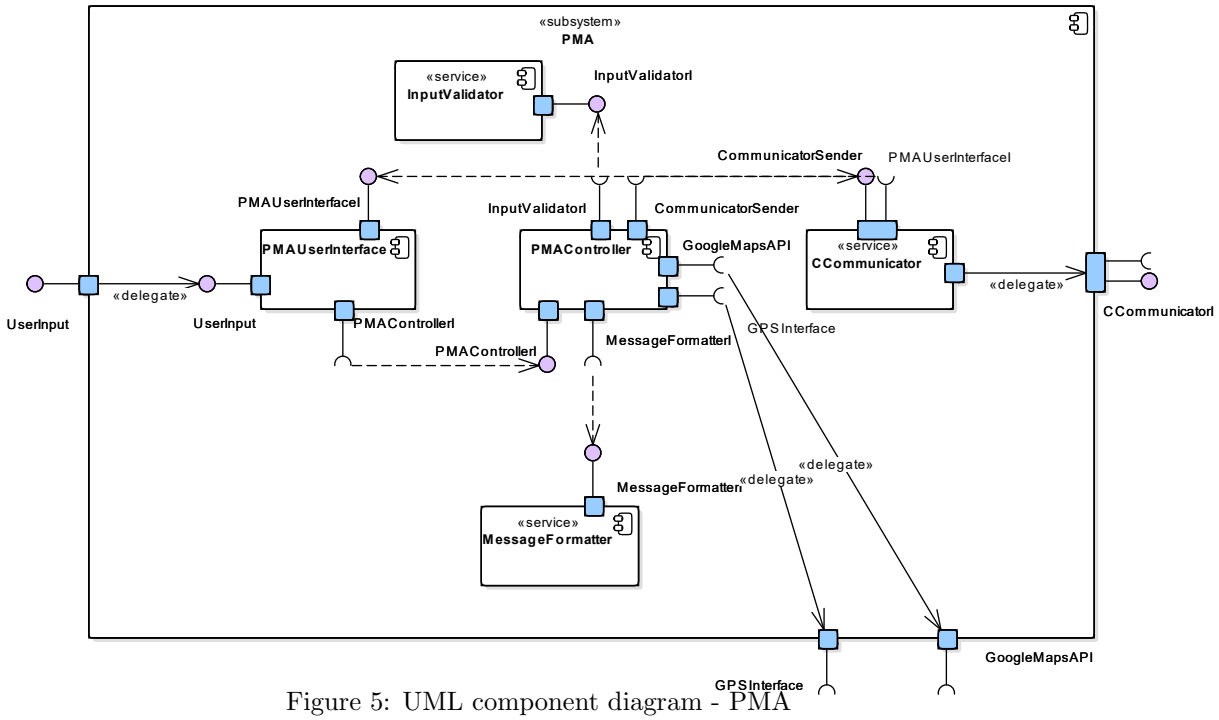


Figure 5: UML component diagram - PMA

- *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

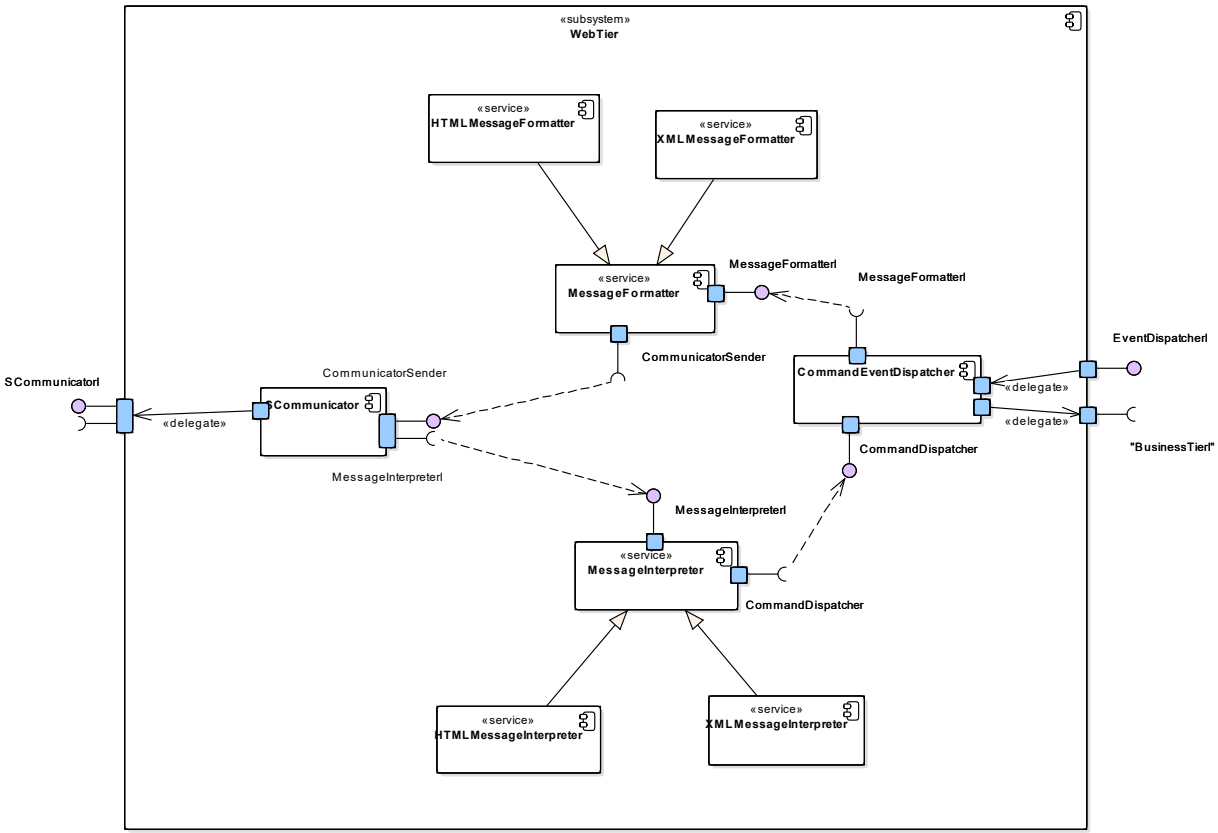


Figure 6: UML component diagram - WebSubsystem

- *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

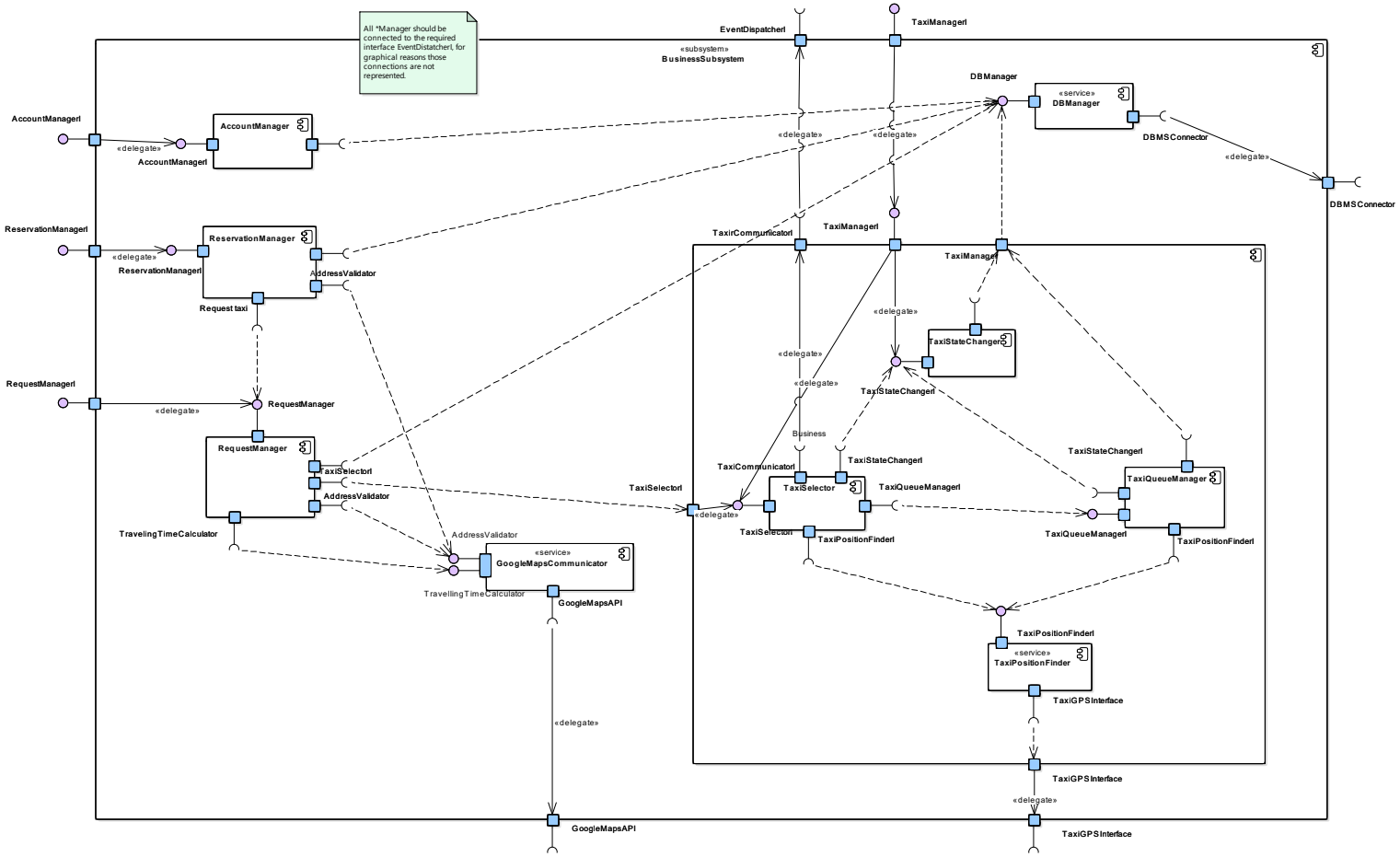


Figure 7: UML component diagram - BusinessSubsystem

- *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.

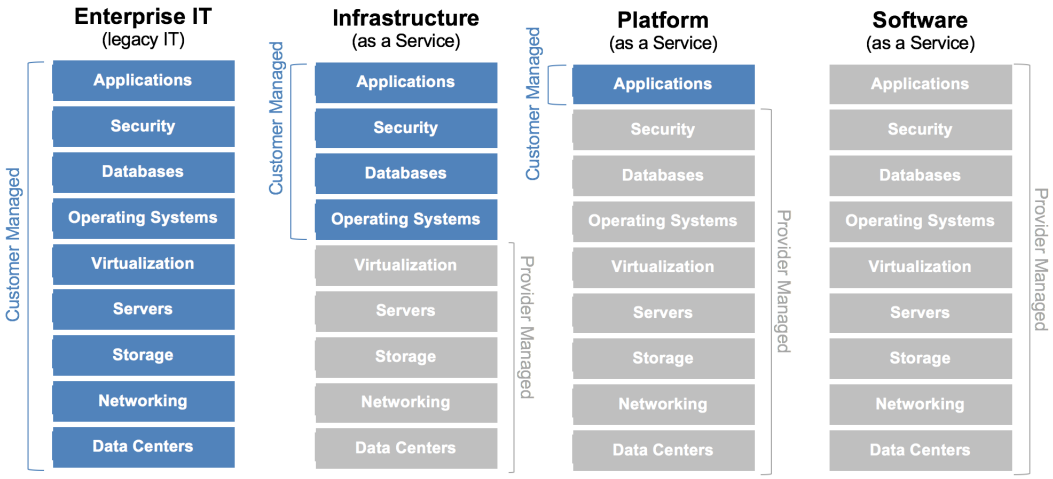


Figure 8: Cloud computing levels

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.

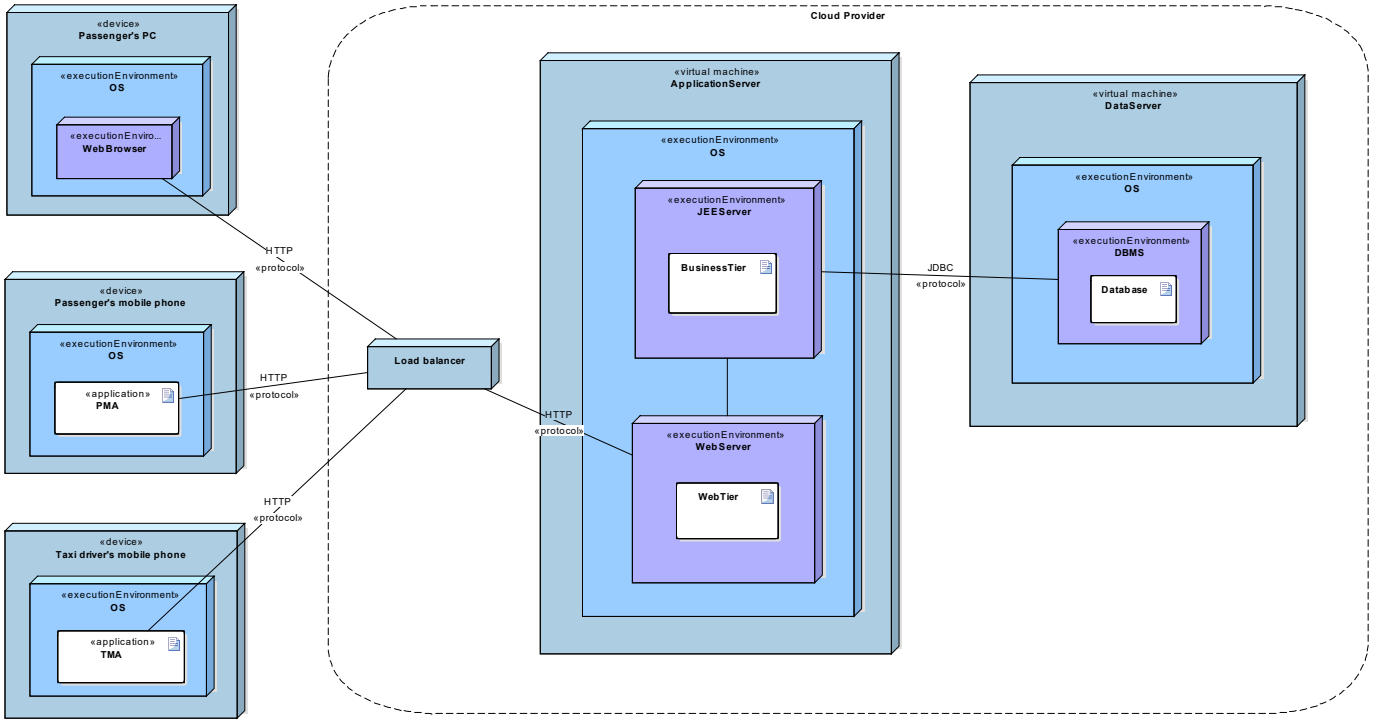


Figure 9: UML deployment diagram

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

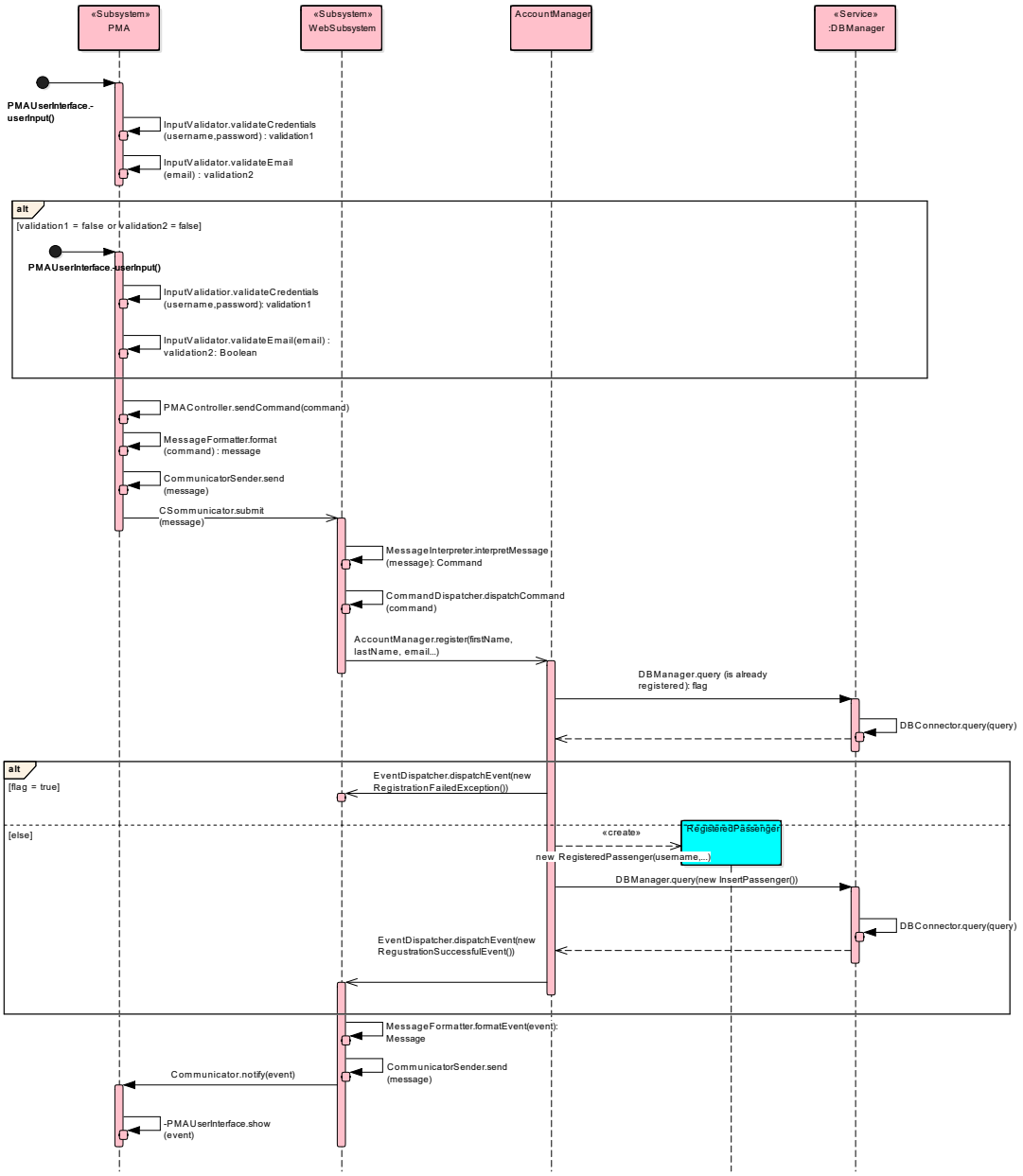


Figure 10: UML Sequence diagram - Registration

2.6.2 Login

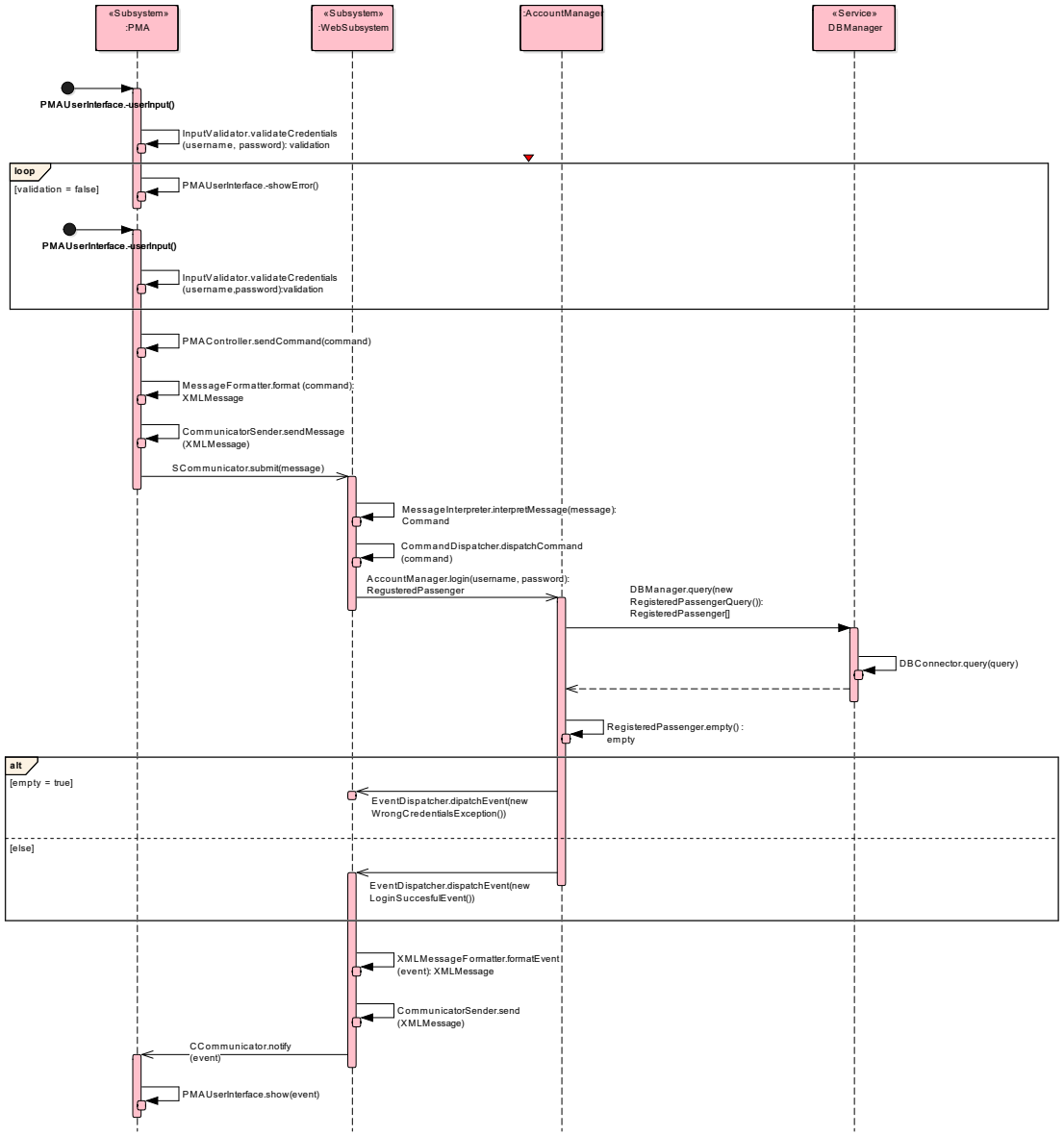


Figure 11: UML Sequence diagram - Login

2.6.3 Request using PMA

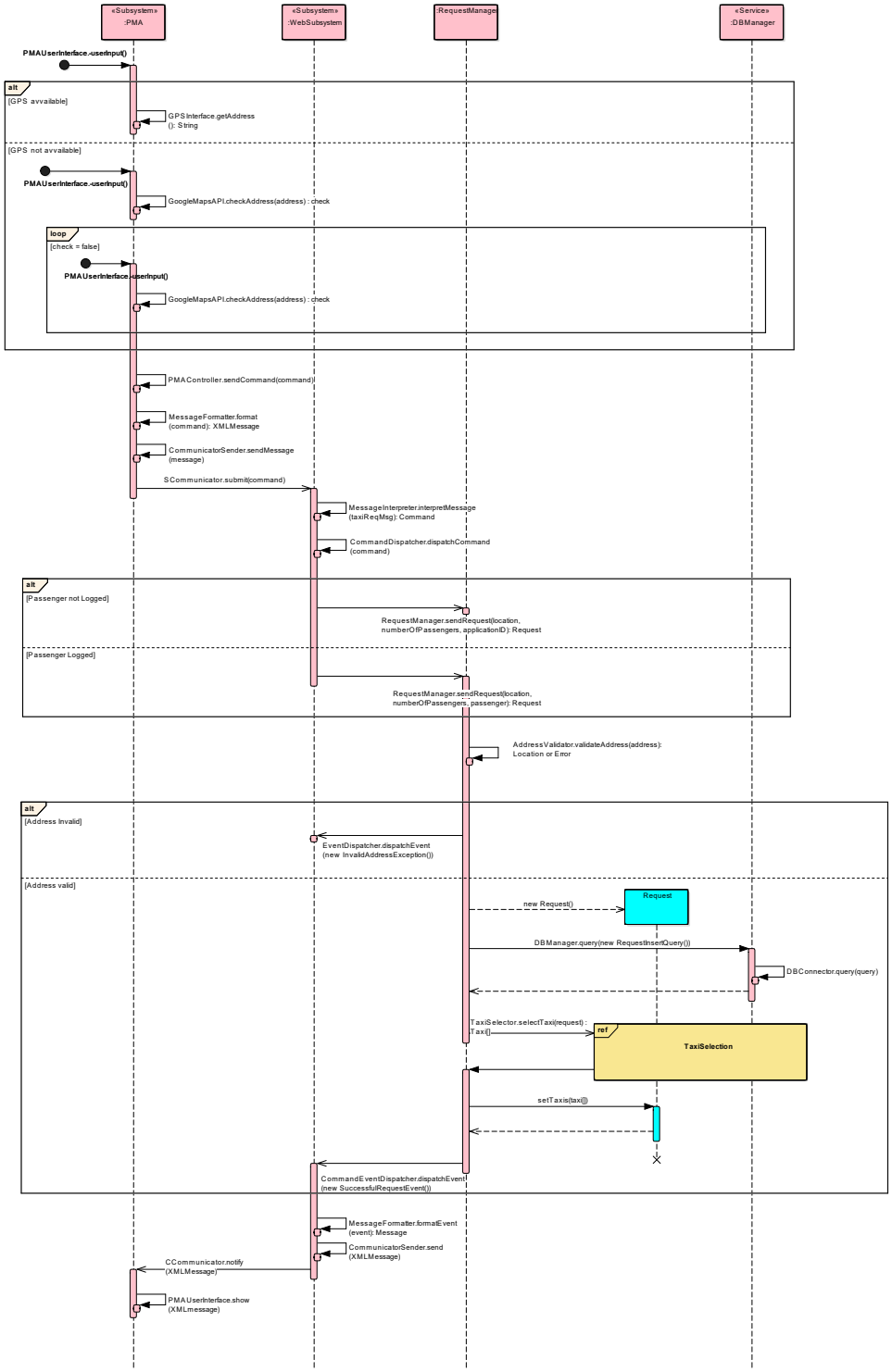


Figure 12: UML Sequence diagram - Request using PMA

2.6.4 Reservation using PWA

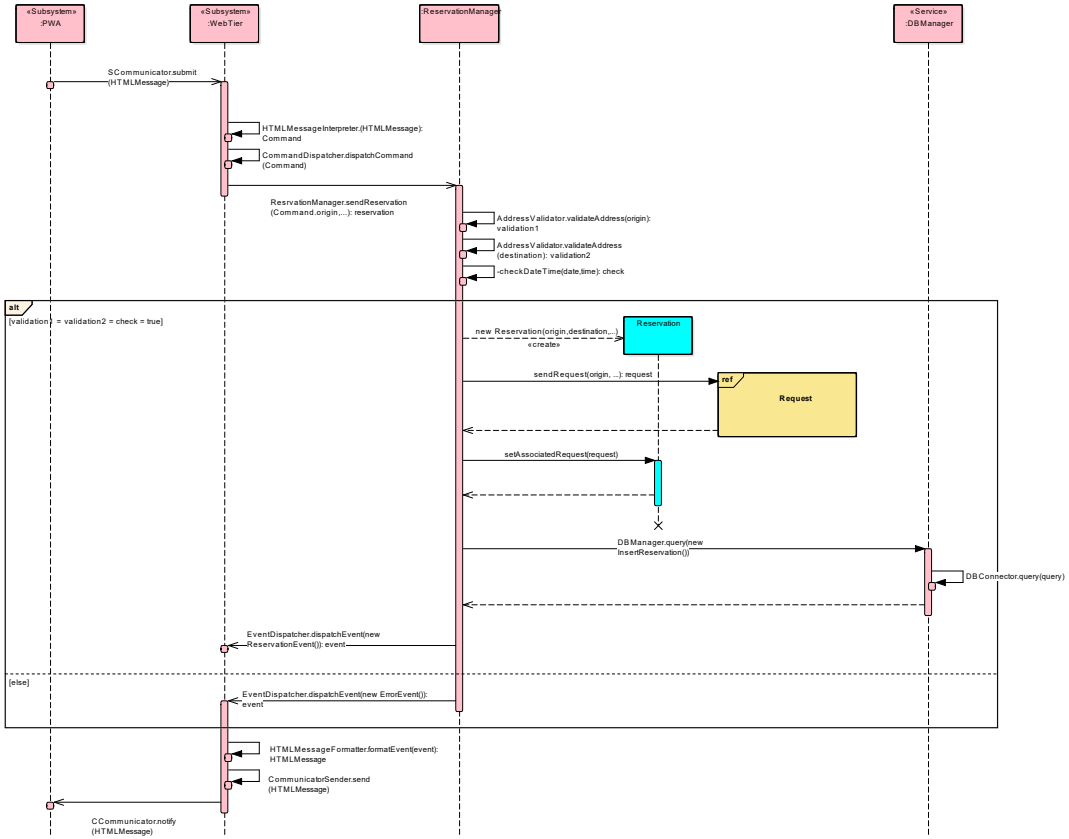


Figure 13: UML Sequence diagram - Request using PWA

2.6.5 Cancel reservation using PMA

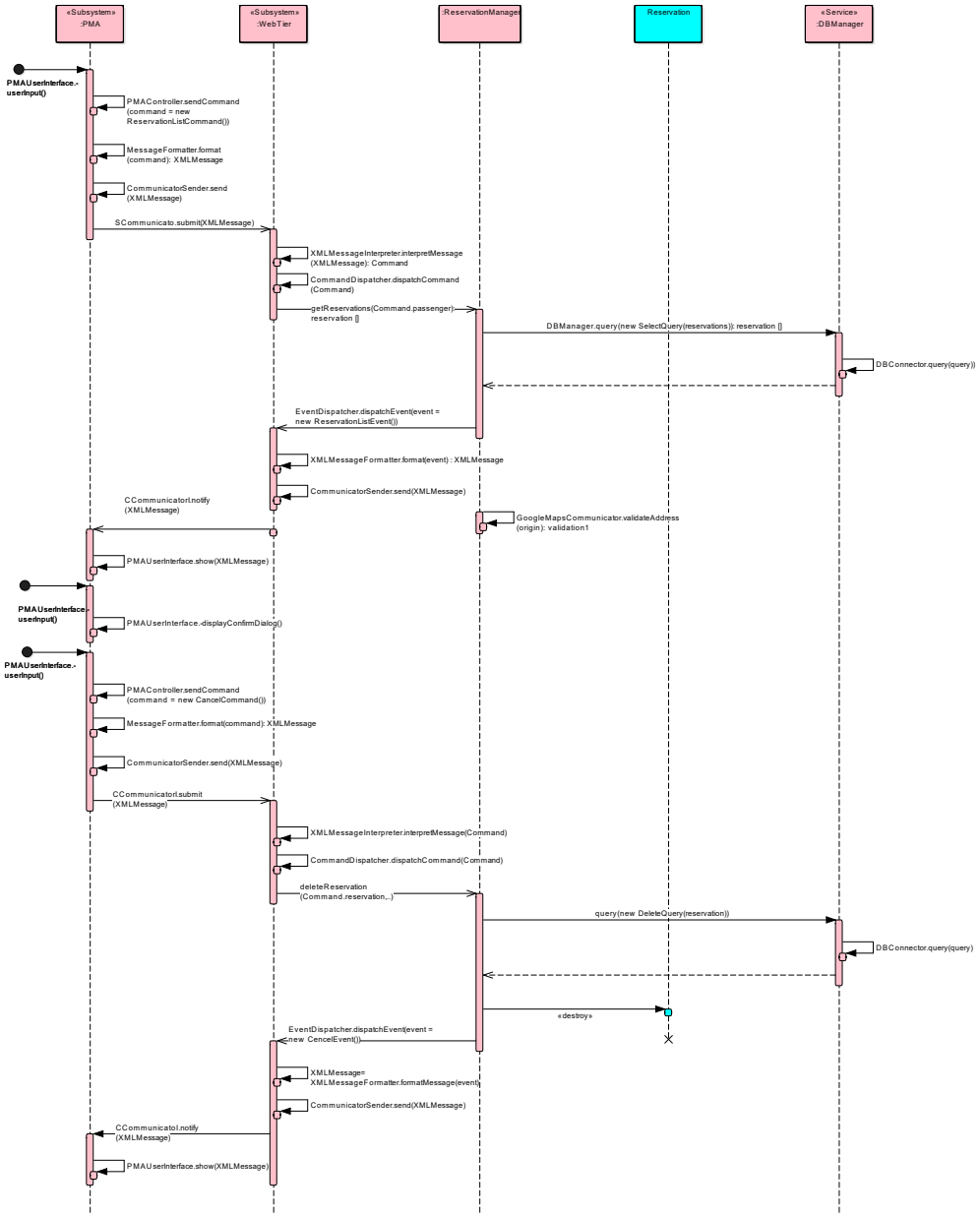


Figure 14: UML Sequence diagram - Cancel reservation using PWA

2.6.6 Taxi selection

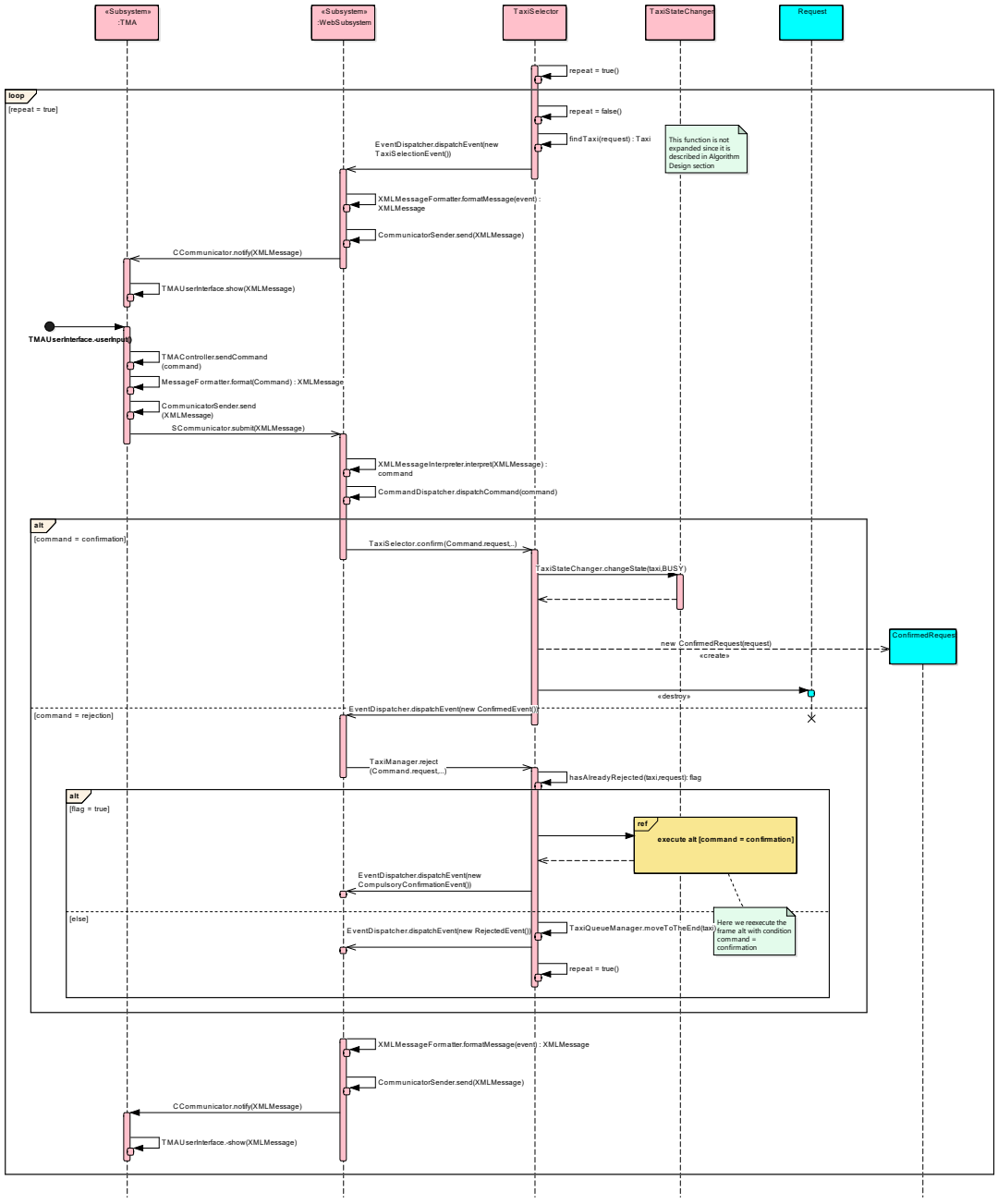


Figure 15: UML Sequence diagram - Taxi selection

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

```
1: function computePosition()
2:     taxis ← TaxiQueueManager.getAvailableTaxis()
3:     for all t ∈ taxis do
4:         location ← TaxiPositionFinder.getTaxiPosition(t)
5:         newZone ← location.getZone()
6:         oldZone ← t.getLocation().getZone()
7:         if zone ≠ oldZone then
8:             TaxiQueueManager.move(t, oldZone, newZone)
9:         end if
10:    end for
11: 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 $G = (Z, A)$.

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

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

```
1: function SELECTTAXISTOMOVE(G', K, D):(X, cost)                ▷ Minimum cost flow
2:                                                  ▷ Find an initial maximum feasible flow
3:     X ← EDMONDS-KARP(G', K)
4:     cost ← 0
5:     for all (i, j) ∈ G'.A' do
6:         cost ← cost + x_ij d_ij
7:     end for
8:                                                  ▷ Until there is a negative cost cycle...
9:     optimal ←false
10:    while not optimal do
11:                                                 ▷ ...compute the residual network...
12:        (Ḡ, D̄, K̄) ←RESIDUALNETWORK(G, K, D, X)
13:                                                 ▷ ...find a negatite cycle...
14:        C ←FINDNEGATIVECYCLE(Ḡ, D̄, K̄)
15:        if C = {} then
16:            optimal ←true
17:        else
18:                                                 ▷ ...update the flow
19:            δ ← min_{(i,j)∈C} k̄_ij
20:            for all (i, j) ∈ G'.A' do
21:                if (i, j) ∈ C then
22:                    x_ij ← x_ij + δ
23:                    cost ← cost + δd̄_ij
24:                end if
25:                if (j, i) ∈ C then
26:                    x_ij ← x_ij − δ
27:                    cost ← cost + δd̄_ij
28:                end if
29:            end for
30:        end if
31:    end while
32:    return (X, cost)
33: 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 $O(|Z||A'|^2k_{max}d_{max})$, 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.

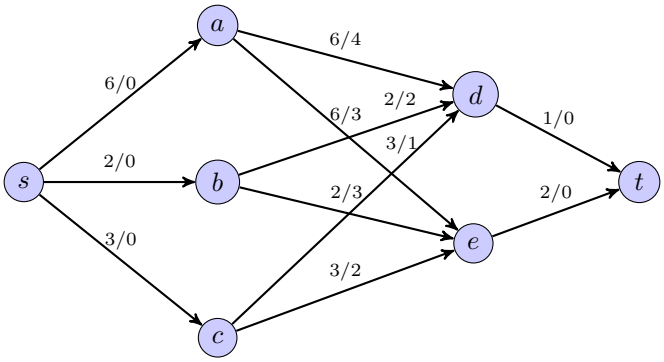
[15]From R. K. Ahuja, T. L. Magnati, J. B. Orlin, Network Flows, 1993

3.1.3.1 Some observations

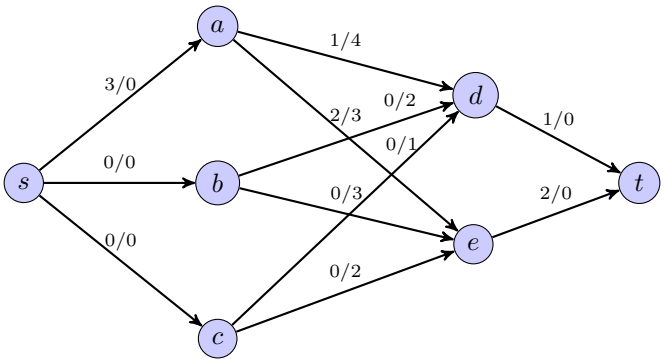
- The algorithm returns the **number** of taxis to be moved from each zone to each zone, not **which** taxis to move. We will assume to move the taxis that are located at the tail of the queue.
- We should require that the algorithm ensures a final value of q_i strictly grater than $t_{i,min}$ to avoid 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

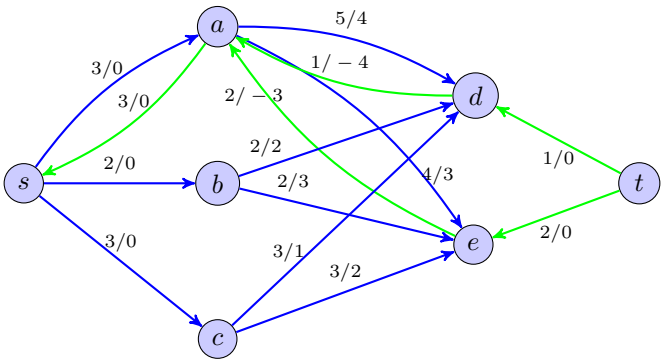
```
1: function RELOCATETAXIS
2:     Z ← TaxiQueueManager.getZones()
3:     numberOfAvailableTaxis ← TaxiQueueManager.getNumberOfAvailableTaxis()
4:                                                  ▷ Compute the closure of the graph
5:     (G+, D) ←GRAPHTRANSITIVECLOSURE(Z)
6:                                                  ▷ Build the network G'
7:     G' ← (Z ⋃ {s, t}, {})
8:     for all (i, j) ∈ G+.A+ do
9:         if not i.isLackingOfTaxis(numberOfAvailableTaxis) and
   j.isLackingOfTaxis(numberOfAvailableTaxis) then
10:            G'.A' ← G'.A' + {(i, j)}
11:            k_ij ← TaxiQueueManager.getNumberOfTaxis(i) − t_i,min
12:        end if
13:    end for
14:    for all i ∈ G'.N − {s, t} do
15:        if not i.isLackingOfTaxis(numberOfAvailableTaxis) then
16:            G'.A' ← G'.A + {(s, i)}
17:            k_si ← TaxiQueueManager.getNumberOfTaxis(i) − t_i,min
18:            d_si ← 0
19:        else
20:            G'.A' ← G'.A + {(i, t)}
21:            k_it ← t_i,min − TaxiQueueManager.getNumberOfTaxis(i)
22:            d_it ← 0
23:        end if
24:    end for
25:    X ←SELECTTAXISTOMOVE(G', K, D)
26:                                                  ▷ Send notification to taxis
27:    for all x_ij > 0 do
28:        for all 1 <= n <= x_ij do
29:            taxi ← TaxiQueueManager.getLast(i)
30:            TMASENDNOTIFICATION(taxi, j)
31:        end for
32:    end for
33: end function
```


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

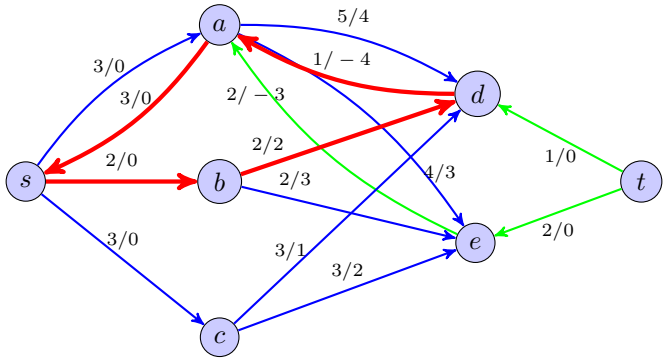
(a)


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

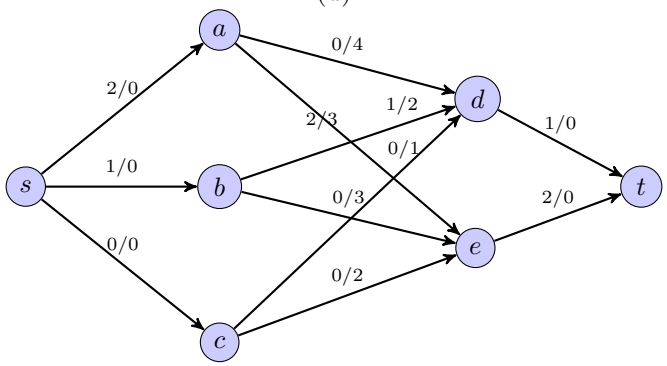
(b)


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

(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

```
1:  function findTaxi(request)
2:      visitedZones ←[false]
3:      zone ← request.getLocation().getZone()
4:      if TaxiQueueManager.getNumberOfTaxis(zone) = 0 then
5:          visitedZones[zone] ←(true)
6:          adjZones ← zone.getAdjacentZones()
7:          found ← false

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

11:             if TaxiQueueManager.getNumberOfTaxis(z) = 0 then
12:                 for all h ∈ z.getAdjacentZones() s.t. not visitedZones[h] do
13:                     adjZones.add(h)
14:                 end for
15:             else
16:                 zone ← z
17:                 found ←true
18:             end if

19:         end while

20:     end if

21:     if not found then
22:         putOnHold(request)
23:     else
24:         taxi ← TaxiQueueManager.getFirst(zone)
25:         TMASendRequest(taxi, request)
26:         waitForAnswer(1 minute)
27:     end if
28: 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

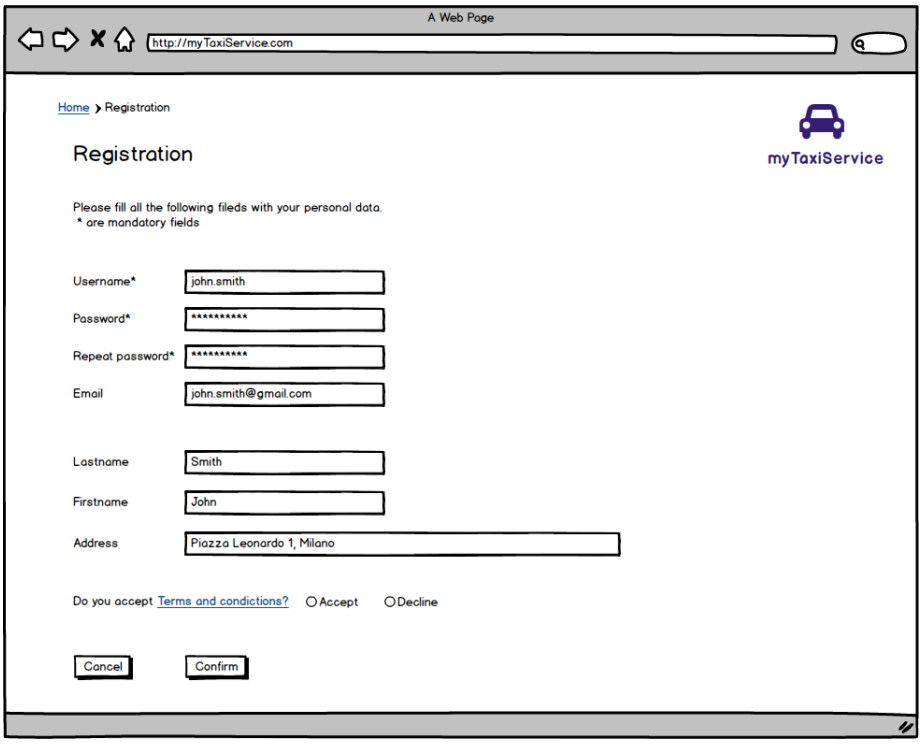


Figure 17: Registration, initial page, web application

Figure 18: Request, page one, web application

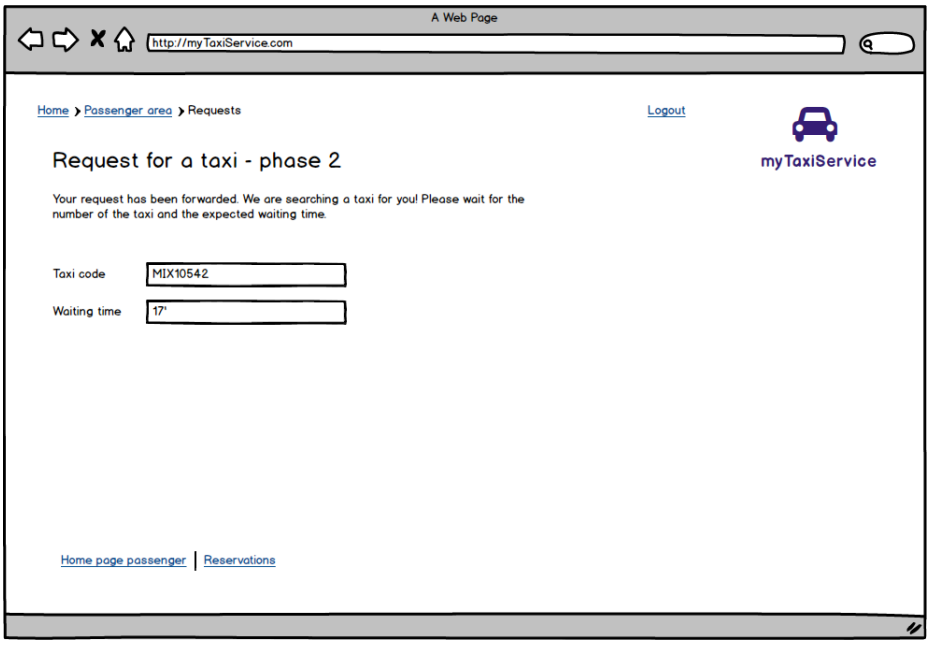


Figure 19: Request, page two, web application

4.1.2 PMA

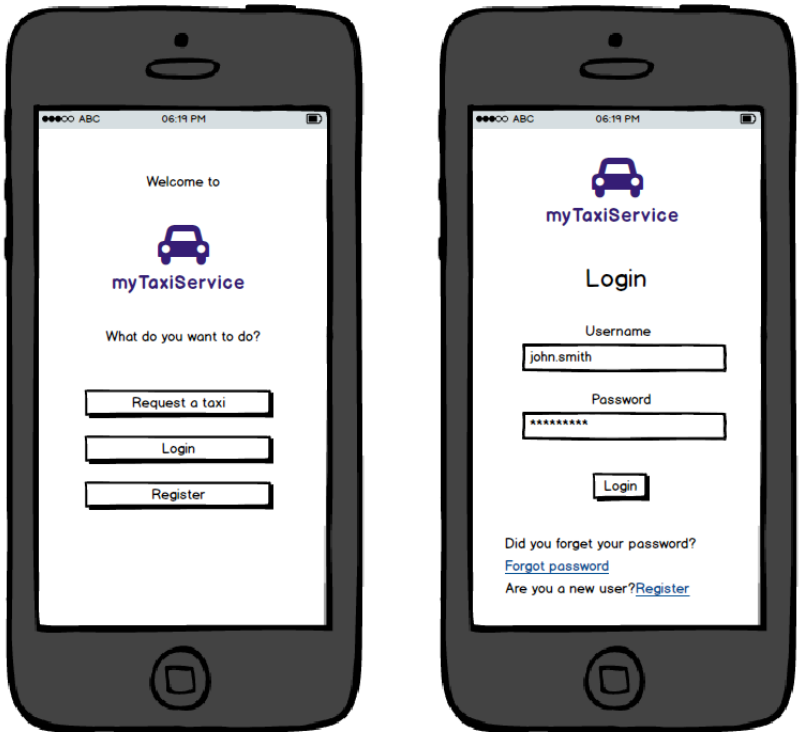


Figure 20: Initial page, mobile application - Login, mobile application

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

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

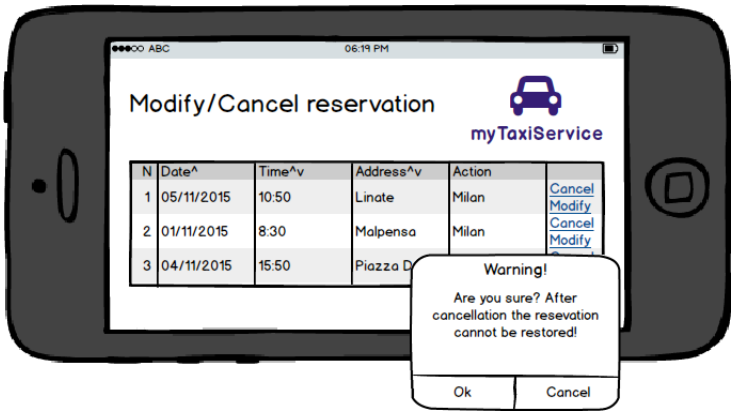


Figure 23: Cancellation confirmation, mobile application

4.1.3 TMA

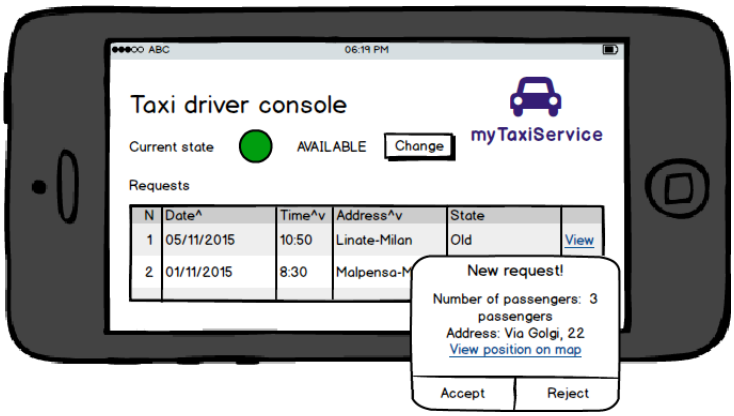


Figure 24: An incoming request

Figure 25: Taxi driver sees the position of the passenger

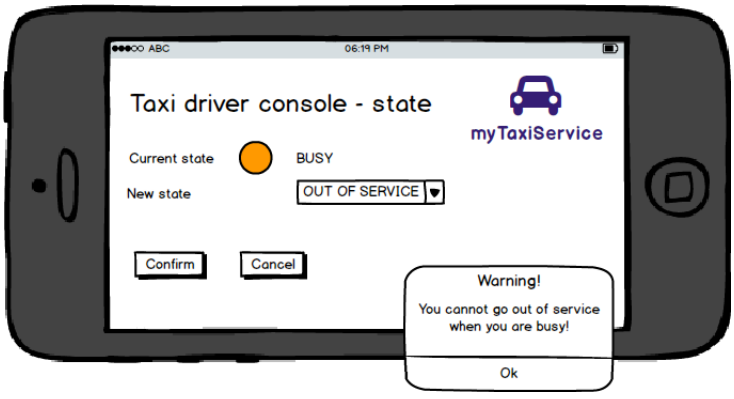


Figure 26: Taxi driver tries go out of service when busy

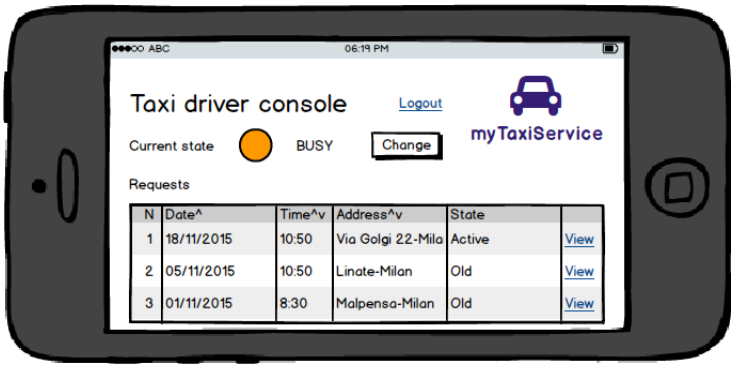


Figure 27: Taxi carrying out a request

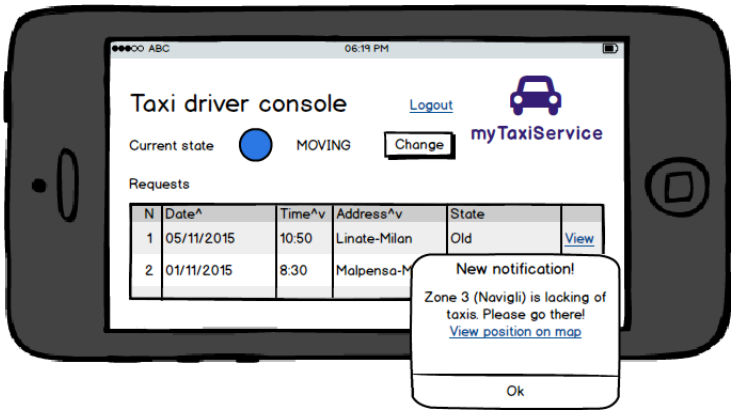


Figure 28: A move notification incoming

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

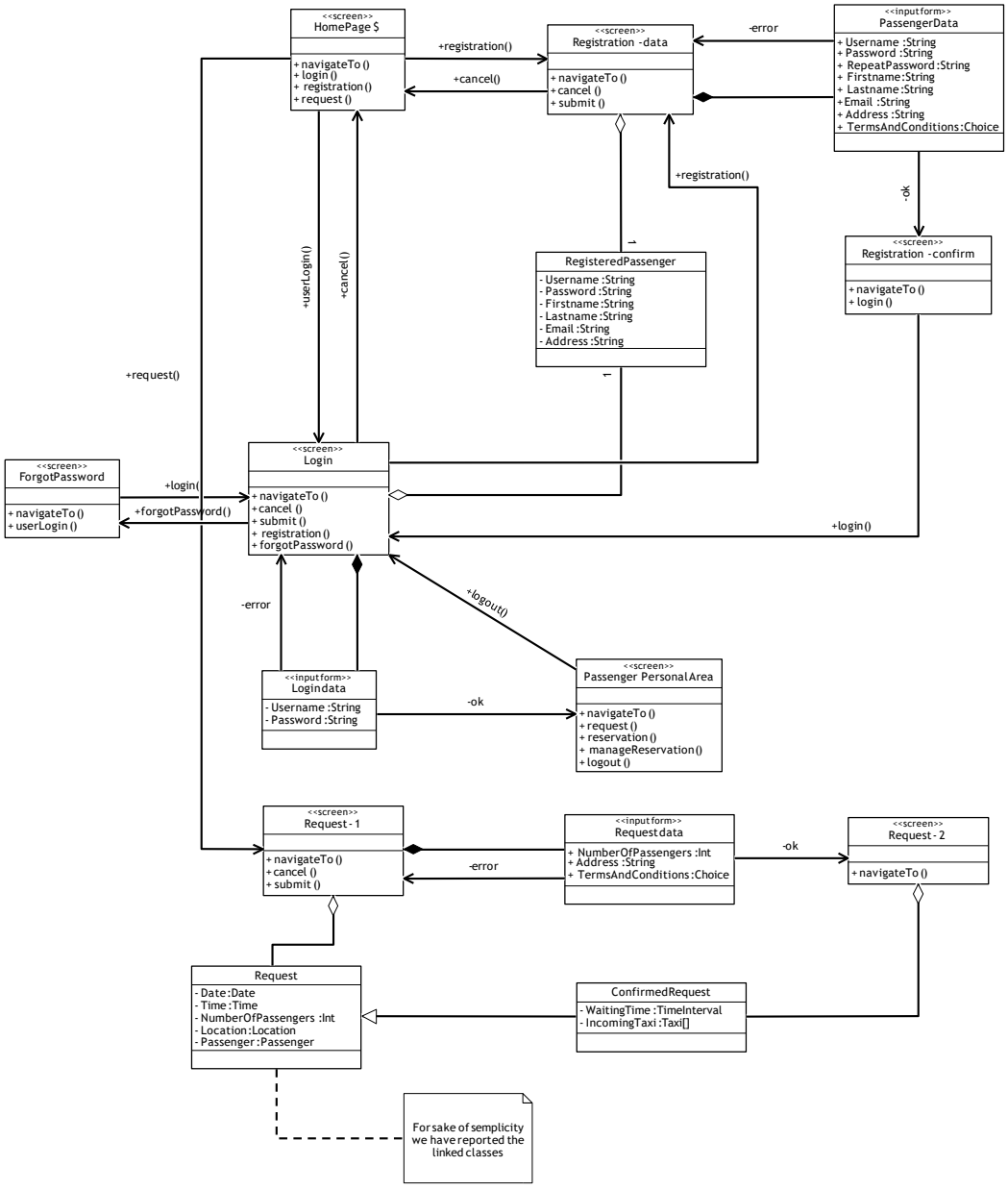


Figure 29: UX diagram - unregistered passenger

4.2.2 Registered passenger

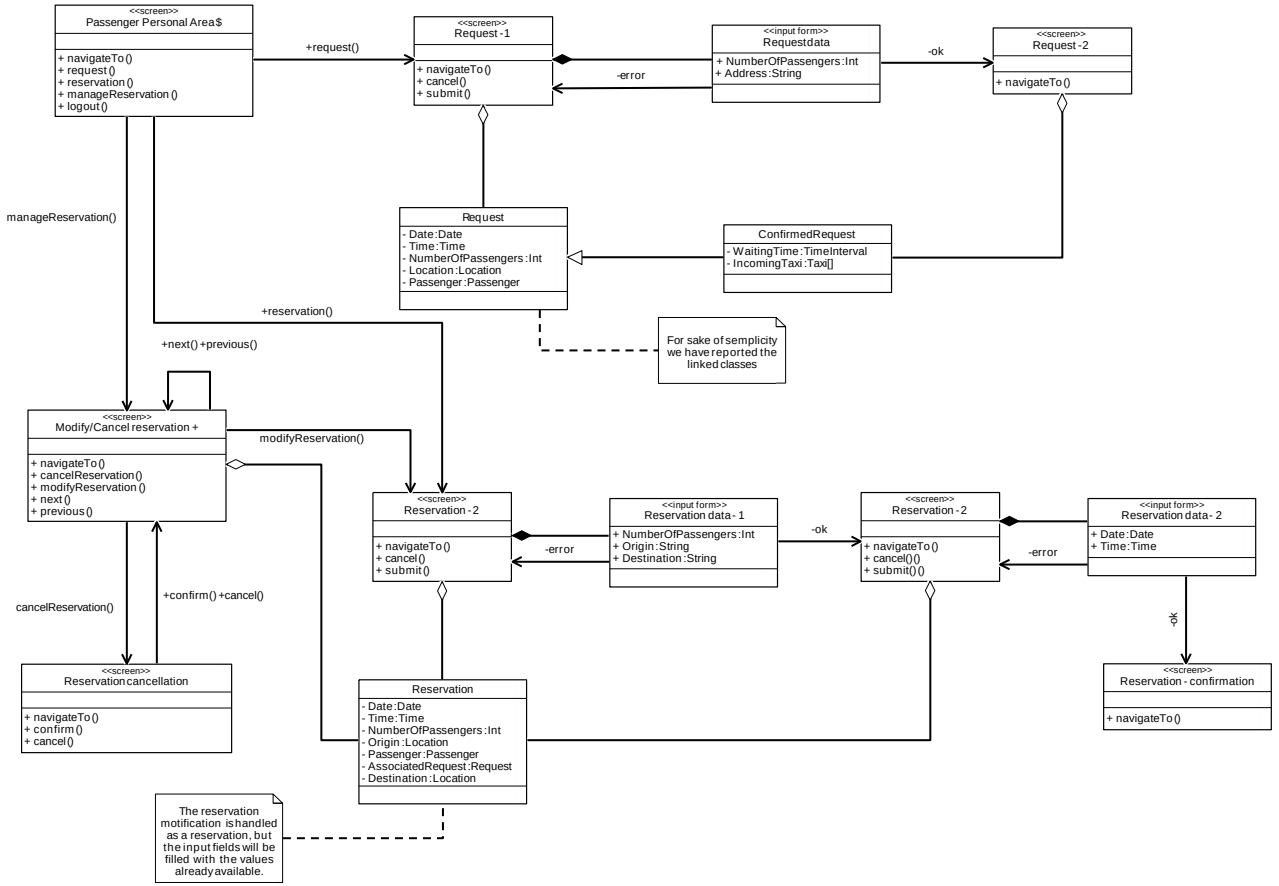


Figure 30: UX diagram - registered passenger

4.2.3 Taxi driver

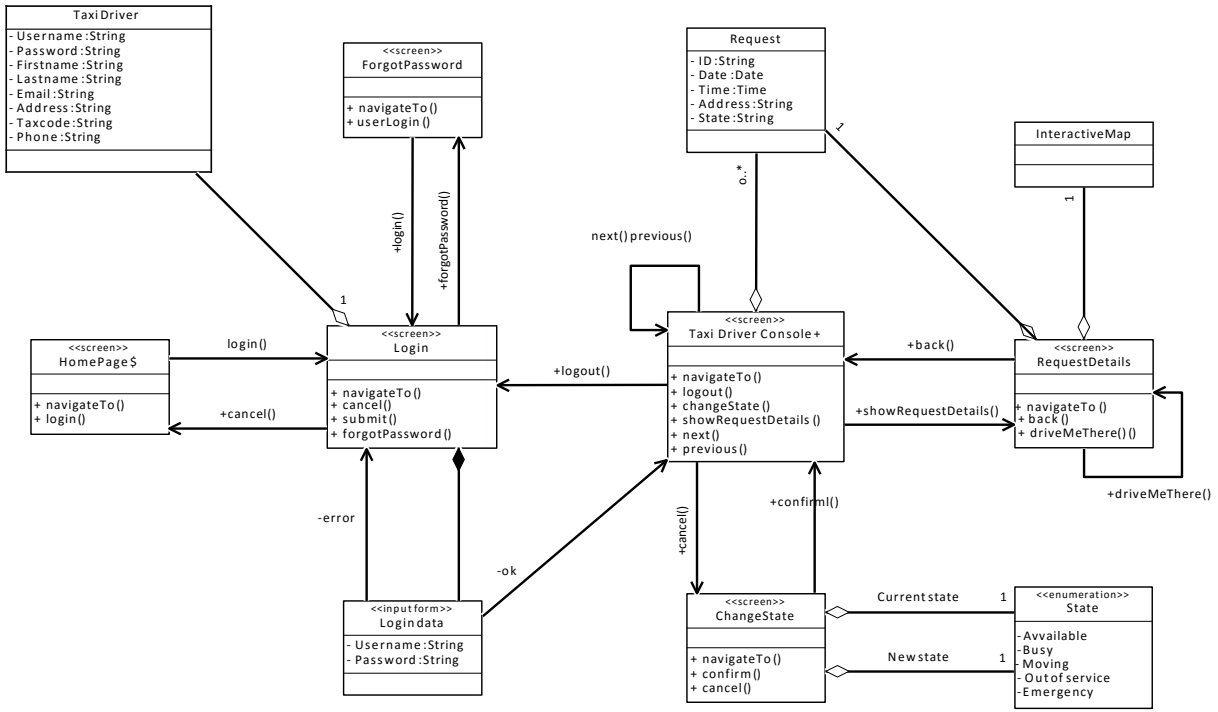


Figure 31: UX diagram - taxi driver

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.

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

ITPD
Integration Test Plan 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 ITPD (*Integration Test Plan Document*) is to give a detailed description of the plan for the integration test of the *myTaxyService* application. The integration test, as a part of the V&V (Verification and Validation) process, is executed after the unit test of the involved modules, which constitutes its main entry condition, and before the system and acceptance test. It is aimed to exercise the interaction between modules and the corresponding interfaces in order to check the compatibility against functional, performance and reliability requirements. The preferred approach for integration testing is *black-box* since single modules are supposed to be already tested isolated and no simple coverage criteria can be defined. Applications are often composed of many modules and the relations among them can be very intricate therefore an *integration test plan* is a key feature of the project plan and has to be compliant to architecture define in the DD and to the build plan, that describes the sequence in which modules will be compounded. The Integration Test Plan Document is intended to be the main reference for this process and it is mainly addressed to the integration test team.

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.

1.3.3 Abbreviations

- Sn n-th subsystem.
- SnIm m-th integration test of the n-th subsystem.
- SnIm-Tk k-the test case of m-th integration test of the n-th subsystem.
- SIm m-th integration test of the full system.
- SIm-Tk k-the test case of m-th integration test of the n-th subsystem.
- EIm m-th integration test with external subsystems.

1.4 Reference documents

[1] The assignment of *myTaxiService*.

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

[3] DD (Design Document) of the *myTaxiService*.

[4] Software Engineering 2 course slides.

[5] Arquillian documentation. `https://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html_single/`

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 ITPD, a very high level description of the main functionalists of the *myTaxiService* system and the resources used to draw up this document.
- The second section constitutes the core of the test plan. This section is devoted to the description of the integration test strategy: the preconditions required to start the integration test will be presented, the main rationals behind the chosen strategy will be discussed; the elements to be integrated will be listed with reference to the ones presented in the DD. Eventually the sequence of integration steps will be clearly illustrated distinguishing between the different levels of abstraction adopted to perform the test.
- The third section contains the definition of the test sets and test cases. Each test will be presented with reference to the sequence explained in the second section, together with the hypothesis about the initial state of the system and the expected results.
- The fourth section is dedicated to the test tools. We will suggest a set of commercial tools suitable to perform the integration test with their main characteristics and the reason why they are appropriate in this context.
- The fifth section is devoted to the description of the program stubs, drivers and the specific test data to accomplish the integration test in some particular scenarios such as external system integration.
- The appendix contains a brief description of the tools used to produce this documents and the number of hours each group member has worked towards the fulfillment of this deadline and the revision history.

2 Integration strategy

This section is devoted to the explanation of the main choices related to the integration testing plan mainly concerning the integration testing strategies.

2.1 Entry criteria

An entry criterion for the integration test is a precondition that is supposed to hold when the integration test is initialized, if one of them is not verified it can compromise or make even impossible the entire process. According the the *software lifecycle*, we identified the following as entry criteria.

1. The implementation phase is terminated therefore the project is code-complete.
2. Unit testing or at least sanity checking[1] for every module/component is complete (every test has been run).
3. All High prioritized bugs fixed and closed (every test has been passed).
4. The internal documentation has been updated to reflect the current state of the product.
5. The ITPD is complete and been approved by QA group.
6. The integration teasing environment setup is completed and stable.

2.2 Elements to be integrated

In this subsection we propose the structure of the designed system, starting from what was described in the DD, in order to clarify the steps needed for the integration testing plan. We do not treat here the problem of integration with external system since a part of section 5 is devoted to it, we assume that part of integration has already been performed.

2.2.1 Preliminary considerations

The integration test plan should be driven by the conceptual system decomposition proposed in the DD, so we construct the plan starting from the components represented there. Although each component can be easily mapped into a programmatic class, being a cohesive and coherent group of functionalists, it is very likely that not all classes needed for the implementation of the application appear as component, most of them will be probably auxiliary class; therefore we assume that during the *unit test*, which constitutes an entry condition for the integration test, the integration *within* the component is performed. If this assumption holds we can proceed to the integration test starting from the components as depicted in the DD.

2.2.2 Levels of integration

Considered the distributed nature and the clear modular structure of myTaxiService application a two level approach of integration testing should be suitable. In particular, the integration phase will be realized at:

[1] *Sanity checking,* sometimes called sanity test, consists in checking that a module/compoent does not contain elementary mistakes or impossibilities, or is not based on invalid assumptions. It is typically a more shallow verification approach with respect to unit testing since just evident mistakes can be manifested. To clarify the distinction, checking that the result of a multiplication of negative numbers is positive is typical of a sanity checking while comparing the result against the one provided by an oracole is typical of unit testing.

- *component level*: each component will be integrated and tested against every dependent component in the contest of the subsystem to which it belongs;
- *subsystems level*: once each subsystem is entirely integrated, all of them will be integrated and tested.

This approach is here just mentioned to understand the following representation of the hierarchies[2] of components/subsystems and will be further discussed in the dedicated section 2.3.

2.2.3 Subsystems

According to the DD the following diagram[3] describes the hierarchy of subsystems to be integrated. The decomposition clearly reflects the JEE architecture, for a detailed description refer to [3].

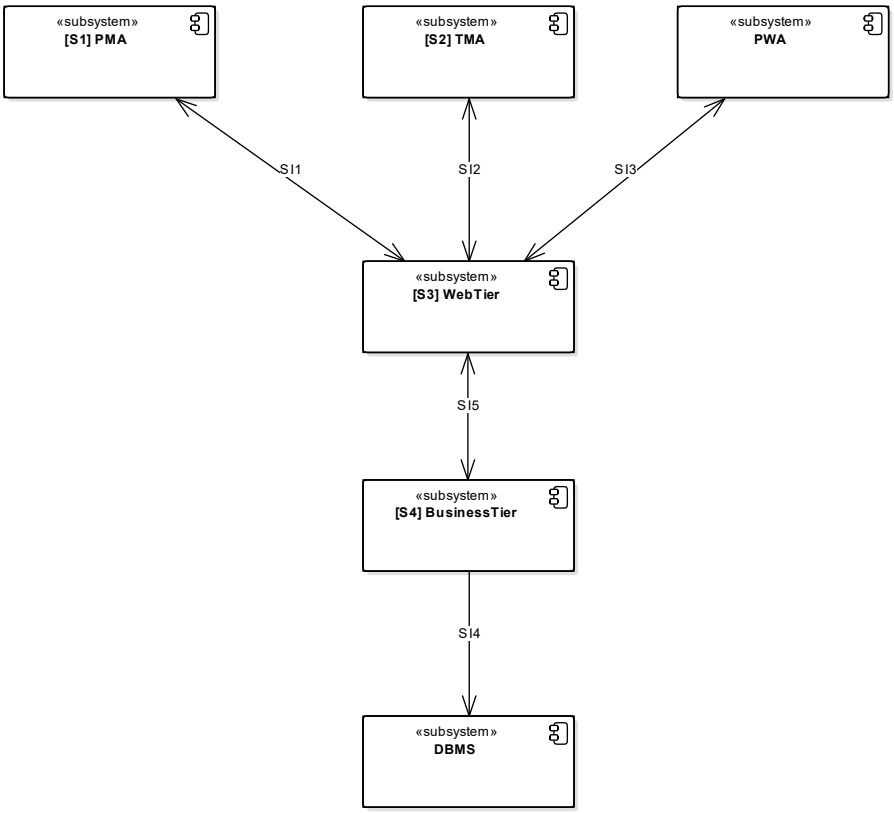


Figure 1: Integration testing plan - subsystem level hierarchy

For each subsystem we present the internal hierarchy of components, since PWA is made of the browser no internal integration is needed, also for the DBMS the internal structure is not known so it is not further specified.

[2] A hierarchy of the dependencies of components is a *DAG* (Directed Acyclic Graph) therefore a bottom-up strategy is actually an plan where integration happens in reverse topological order while in the top-down strategy integration occurs in topological order. If a component A calls a method of another component B (namely A *depends* on B) then B belongs to the layer right below in the hierarchy with respect to A.

[3] The notation used here is a simplified version of UML Component Diagram where just dependencies among modules are represented, while interfaces are not.

2.2.4 PMA

Notice that the hierarchy contains a cycle between PMAUserInterface, PMAController and CCommunicator (so it is not properly speaking a hierarchy) and this is due to the usage of the pattern Observer-Observable where the CCommuncator constitutes the model (actually the link to the remote model)[4]. To break the cycle in order to perform the integration more stubs will be needed.

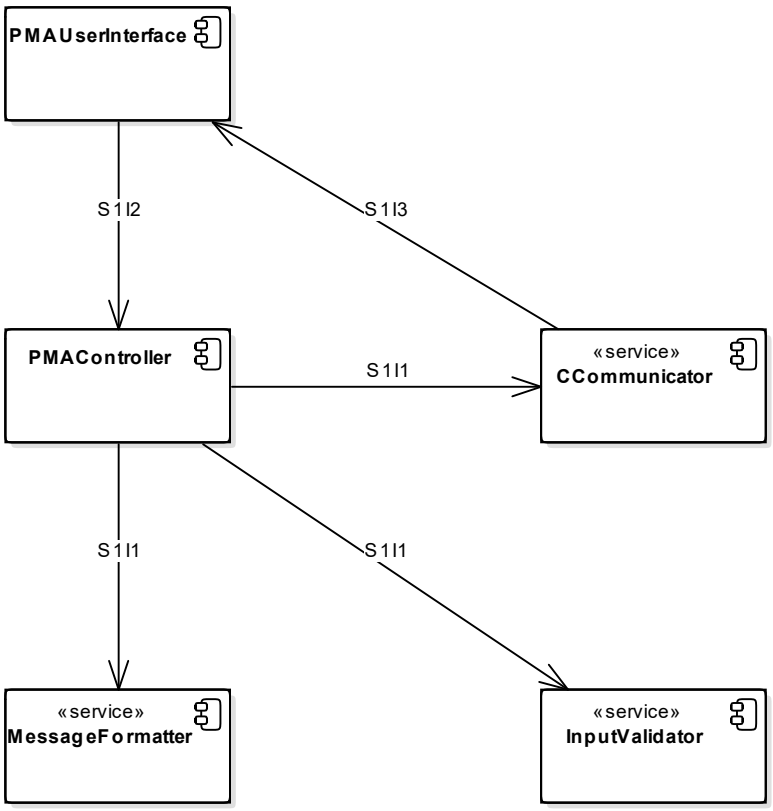


Figure 2: Integration testing plan - PMA

[4]From an implementative point of view that cyclic dependency does not exist since CCommunicator does not accesses directly to PMAUserInterface but just sees the interface Observer.

2.2.5 TMA

Since TMA shares the same structure of PMA the same considerations explained above are valid here.

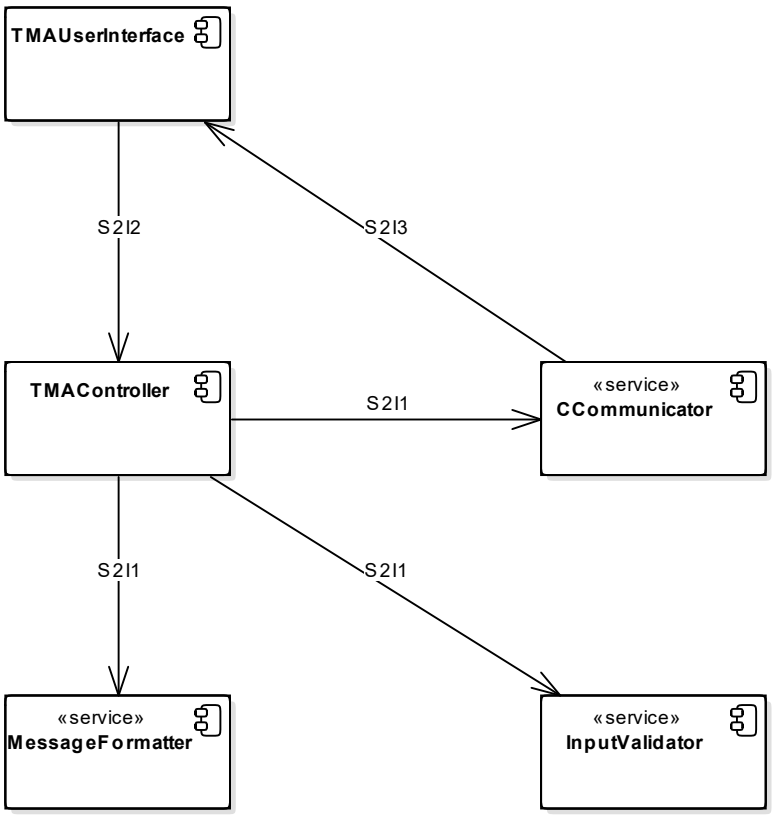


Figure 3: Integration testing plan - TMA

2.2.6 Web subsystem

Notice that the dependency graph is a cycle, this is due to the fact that SCommunicator and CommandEventDispatcher manage the bidirectional flow of messages between clients and Business subsystem. However the interfaces involved in the exchange of messages in the two direction are different (SCommunicator → CommandDispatcher and EventDispatcher → CommunicatorSender).

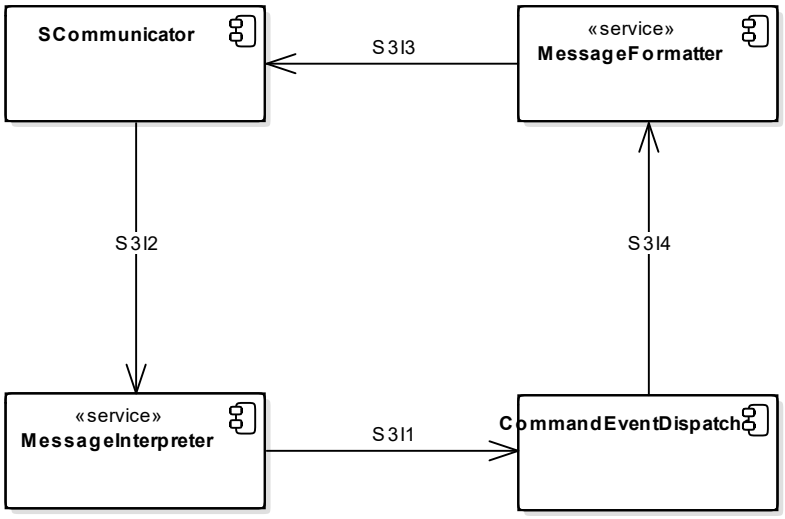


Figure 4: Integration testing plan - TMA

2.2.7 Business subsystem

As it is represented in the DD the business subsystem is in turn composed of an internal macro-component TaxiManager, therefore the process of intergration will be performed integrating TaxiManager before and than completing the integration with the other components. So we present two different hierarchies.

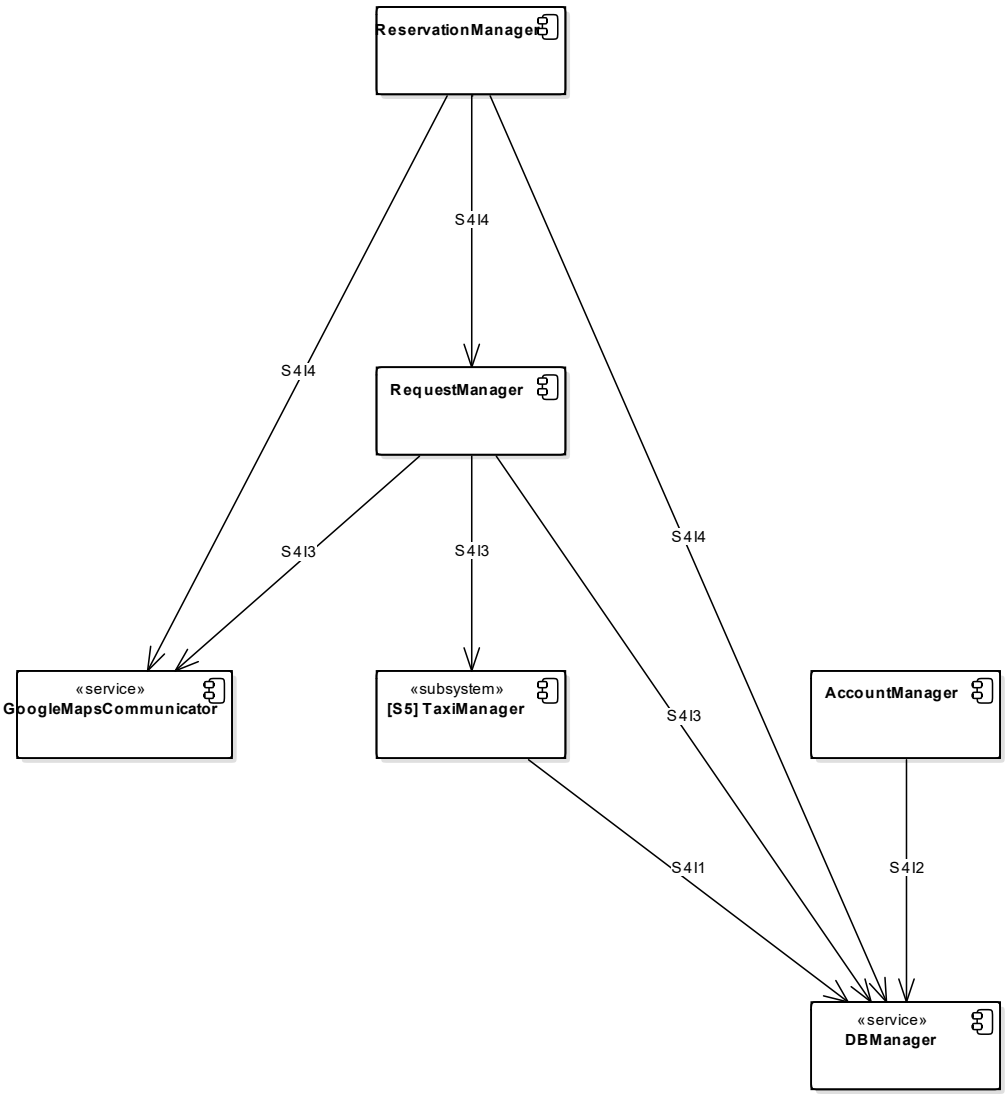


Figure 5: Integration testing plan - Business subsystem

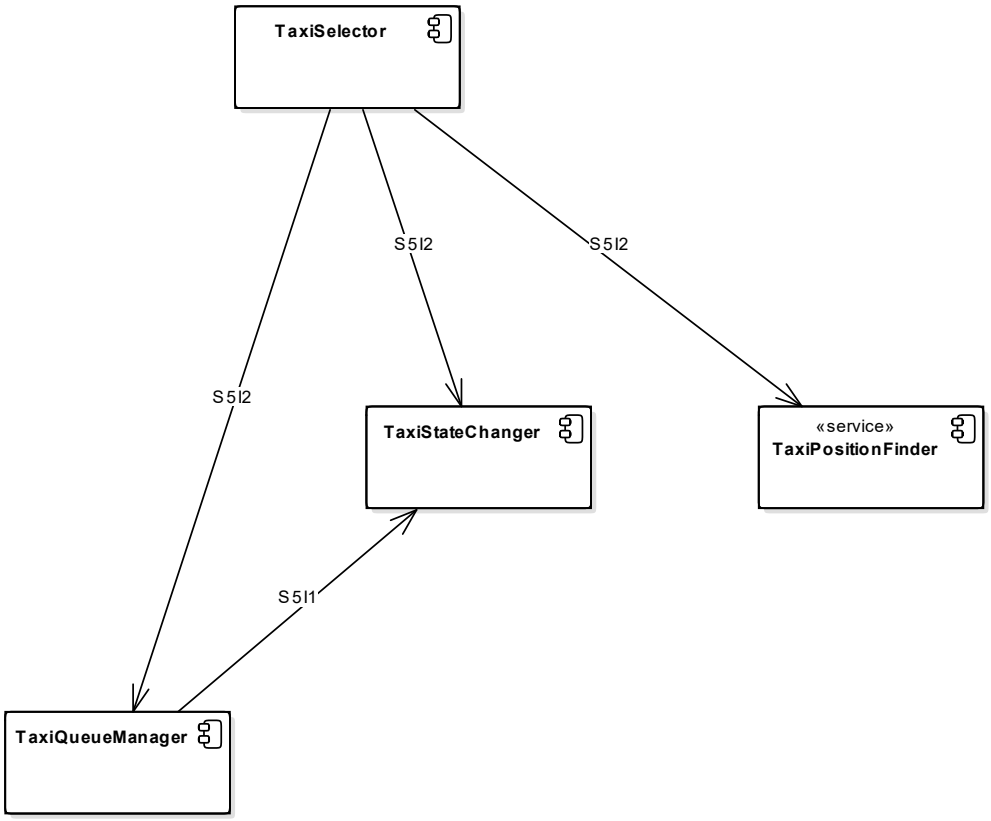


Figure 6: Integration testing plan - Taxi manager

2.3 Integration testing strategy

In this section, starting from the scenario depicted in the previous ones, we illustrate the integration strategy we have chosen providing the rational behind that choice with reference to the software architecture.

2.3.1 Preliminary considerations

Integration process in a client/server application is often a big issue. Coherently with the design and conquer principle, as already mentioned in section 2.2.1, we clearly distinguish between the integration stages at *component level* and at *subsystem level*. This first choice has several advantages.

- At the end of the first stage (integration at component level) it is guaranteed that all subsystems are correctly working, obviously this activity requires the usage of proper drivers and stubs to simulate the environment of the other subsystems.
- The integration at component level for each subsystem can be performed in parallel since they are considered isolated, reducing the project overall time and allowing working of differentiated integration test groups.
- Different integration testing strategies can be selected for the two stages according to the different needs.

2.3.2 Selected integration strategies

Considered the different needs and characteristics of the two levels of abstractions we think a proper solution should adopt different integration testing strategies.

- *Bottom-up* strategy for the integration at component level within each subsystem. The components in the lowest layer of the hierarchy are tested individually, then components belonging to the layer above are integrated and tested until the root of the hierarchy is reached. The main advantages of this approach are the following:
  - only test drivers are used to set up the testing environment and pass the test case, no test stubs are needed;
  - it is suitable for object oriented design methodologies because it starts from the low levels of hierarchies going up to the more abstract elements;
  - favors the evaluation of the performance requirements.
- *Sandwich* strategy for the integration at subsystem level. It combines top-down strategy with bottom-up strategy, favoring parallel testing. The system is viewed as having three layers: a target layer in the middle, a layer above the target and a layer below the target. Testing converges at the target layer. If there are more than three layers, as in myTaxiService, we can exploit heuristics to minimize the number of stubs and drivers; for us it is clear that we may converge towards the Business subsystem. Sandwich strategy brings the following advantages:
  - top and bottom layer tests can be done in parallel;
  - is suitable for large projects having several subprojects.

Notice that the sandwich strategy per se does not prescribe an individual testing strategy of any subsystems (which should be performed simultaneously with the subsystem integration), but since we distinguished between component level and subsystem level integration testing this is automatically implied. This strategy is often referred as *modified sandwich* strategy that, however, does not define the specific approach to be adopted within each subsystem, for us bottom-up.

2.4 Sequence of Component/Function Integration

In this section, according to the hierarchies and integration testing strategies described above we will provide the sequence of integration of components and subsystems. We will exploit the UML activity diagram in order to make the process more clear and highlight the possible parallelizations.

2.4.1 Software Integration Sequence (component level)

At component level the integration testing strategies can be applied in a fully parallelized environment, since we assume to perform integration testing within each subsystem in an isolated way; drivers and stubs for the interacting component will be necessary. Now for each subsystem we provide the sequence in which integration testing will be performed. Drivers will be necessary at each step (refer to section 4 for the detailed description) since we are proceeding bottom-up but also "external" stubs are necessary[5].

[5]Typically Bottom-up integration testing does not require any stub, however since we adopt this strategy at level of components we need to model the other subsystems, those models will be called *external stubs.* They can be real stubs or just the actual subsystems if already integrated.

2.4.2 PMA [S1]

All leaves components are supposed to be individually tested with a suitable drivers.

S1I1 PMAController → InputValidator

S1I1 PMAController → MessageFormatter

S1I1 PMAController → CCommunicator

S1I2 PMAUserInterface → PMAController

S1I3 CCommunicator → PMAUserInterface

For test S1I1 PMAController → CCommunicator an external stub for the Web subsystem is needed while, since there is a cycle between PMAController, PMAUserInterface and CCommunicator a stub for CCommunicator is needed when performing integration test S1I2 (we can actually use the same CCommunicator since it is already unit tested and integrated).

2.4.3 TMA [S2]

All leaves components are supposed to be individually tested with a suitable drivers.

S2I1 TMAController → InputValidator

S2I1 TMAController → MessageFormatter

S2I1 TMAController → CCommunicator

S2I2 TMAUserInterface → PMAController

S2I3 CCommunicator → PMAUserInterface

For test S2I1 TMAController → CCommunicator an external stub for the Web subsystem is needed while, since there is a cycle between TMAController, TMAUserInterface and CCommunicator a stub for CCommunicator is needed when performing integration test S2I2 (we can actually use the same CCommunicator since it is already unit tested and integrated).

2.4.4 Web subsystem [S3]

S3I1 MessageInterpreter → CommandEventDispatcher

S3I2 SCommunicator → MessageInterpreter

S3I3 MessageFormatter → SCommunicator

S3I4 CommandEventDispatcher → MessageFormatter

When performing all those integration test an external stub for the Business Subsystem and the client is needed.

2.4.5 TaxiManager [S5]

All leaves components are supposed to be individually tested with a suitable drivers.

S5I1 TaxiQueueManager → TaxiStateChanger

S5I2 TaxiSelector → TaxiStateChanger

S5I2 TaxiSelector → TaxiPositionFinder

S5I2 TaxiSelector → TaxiStateChanger

When performing all those integration test an external stub for the Business Subsystem is needed, since TaxiManager is one of its subcomponents.

2.4.6 Business subsystem [S4]

S4I1 TaxiManager → DBManager

S4I2 AccountManager → DBManager

S4I3 RequestManager → GoogleMapsCommunicator

S4I3 RequestManager → TaxiManager

S4I3 RequestManager → DBManager

S4I4 ReservationManager → GoogleMapsCommunicator

S4I4 ReservationManager → DBManager

S4I4 ReservationManager → RequestManager

When performing all those integration test an external stub for the DBMS is needed.

2.4.7 Subsystem Integration Sequence (subsystem level)

At subsystem level the integration is performed based on the sandwich strategy. We list the integration steps to be followed; notice that thanks to the modified sandwich strategy SI1, SI2, SI3 and SI4 are to be performed in parallel. Moreover, SI1, SI2 and SI3 go top-down so they need a stub of Business subsystem while SI4 goes bottom-up so it needs a driver of Web subsystem.

SI1 PMA ↔ Web subsystem

SI2 TMA ↔ Web subsystem

SI3 PWA ↔ Web subsystem

SI4 Business subsystem → DBMS

SI5 Web subsystem ↔ Business subsystem

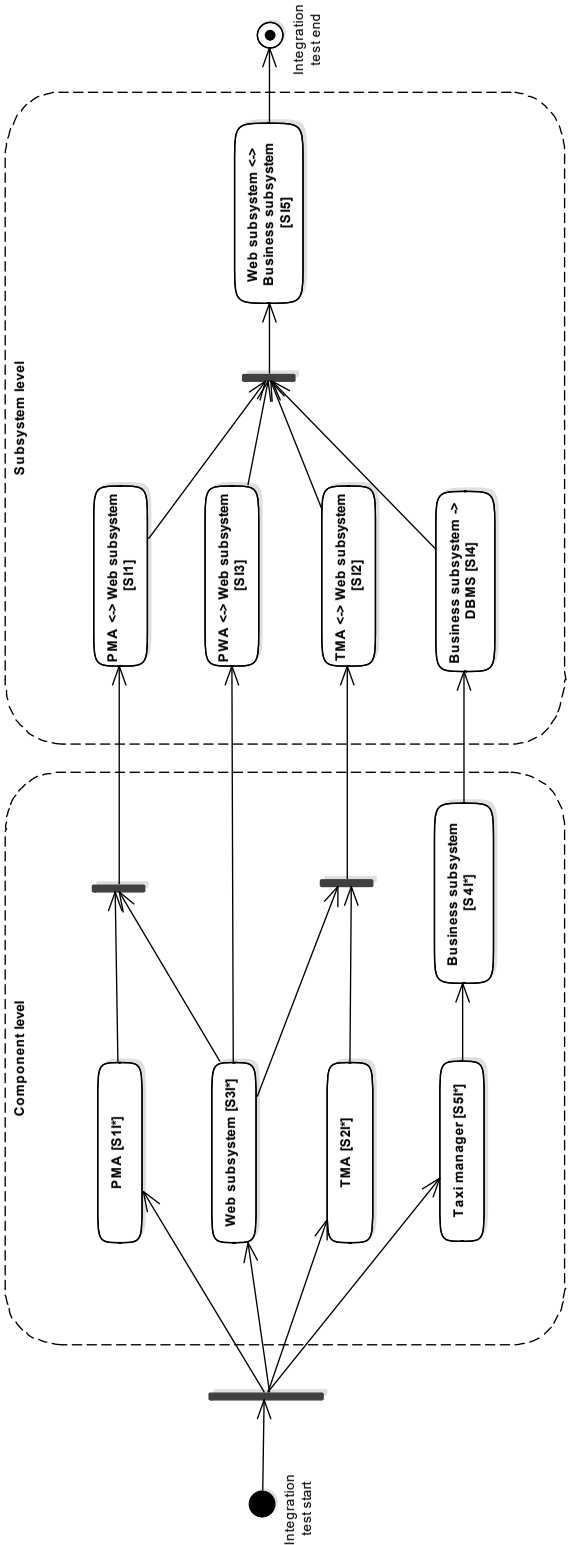


Figure 7: Integration test plan - activity diagram

3 Individual steps and test description

In this section, following the two steps depicted in the previous sections, we present the test cases we have identified according to the integration strategy chosen. For each test case we report the involved items, either components or subsystems, the input/output specification, the functional/non functional requirements tested with reference to the RASD and the DD and the suggested technique to be used.

3.1 Component level integration test

3.1.1 [S1] PMA

Test case identifier	**S1I1-T1**
Test items	PMAController MessageFormatter
Input specification	Create typical PMAController input
Output specification	Check if the correct methods are called in the MessageFormatter
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on PMAController, format is called on MessageFormatter.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S1I1-T2**
Test items	PMAController InputValidator
Input specification	Create typical PMAController input
Output specification	Check if the correct methods are called in the InputValidator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand with a register command is called on PMAController validateEmail and validateCredentials are called on InputValidator. 2. Check that when sendCommand with a login command is called on PMAController validateCredentials is called on InputValidator. 3. Check that when sendCommand with a reservation command is called on PMAController validateDate and validateTime are called on InputValidator.
Tested non functional requirements	1. (Reliability) Check that all the user input are verified by the local application before being sent to the server.
Testing technique	Automated

Test case identifier	**S1I1-T3**
Test items	PMAController CCommunicator (CommunicatorSender)
Input specification	Create typical PMAController input
Output specification	Check if the correct methods are called in the CCommunicator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on PMAController send is called on CCommunicator (CommunicatorSender interface).
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S1I2-T1**
Test items	PMAUserInterface PMAController
Input specification	Create typical PMAUserInterfaceinput
Output specification	Check if the correct methods are called in the PMAController
Environmental needs	S1I1 succeeded
Tested functional requirements	1. Check that when an action by the user is performed on the user interface the method sendCommand is called on PMAController.
Tested non functional requirements	1. (UI) Check that every functionality shall be reached surfing no more than 4 pages. 2. (Reliability) Check that the response time is smaller that 10 ms in 99% of times for local elaborations (not involving Internet connection).
Testing technique	Manual

Test case identifier	**S1I3-T1**
Test items	CCommunicator (CCommunicator) PMAUserInterface
Input specification	Create typical CCommunicator input
Output specification	Check if the correct methods are called in the PMAUserInterface
Environmental needs	S1I2 succeeded
Tested functional requirements	1. Check that when notify method is called on CCommunicator (CCommunicatorI interface) the method show is called on PMAUserInterface
Tested non functional requirements	-
Testing technique	Automated

3.1.2 [S2] TMA

Test case identifier	**S2I1-T1**
Test items	TMAController MessageFormatter
Input specification	Create typical TMAController input
Output specification	Check if the correct methods are called in the MessageFormatter
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on TMAController format is called on MessageFormatter.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S2I1-T2**
Test items	TMAController InputValidator
Input specification	Create typical TMAController input
Output specification	Check if the correct methods are called in the InputValidator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand with a login command is called on TMAController validateCredentials is called on InputValidator. 2. Check that when sendCommand with an 'input' command is called on TMAController validateDate and validateTime are called on InputValidator.
Tested non functional requirements	1. (Reliability) Check that all the user input are verified by the local application before being sent to the server.
Testing technique	Automated

Test case identifier	**S2I1-T3**
Test items	TMAController CCommunicator (CommunicatorSender)
Input specification	Create typical TMAController input
Output specification	Check if the correct methods are called in the CCommunicator
Environmental needs	-
Tested functional requirements	1. Check that when sendCommand is called on TMAController send is called on CCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S2I2-T1**
Test items	TMAUserInterface TMAController
Input specification	Create typical TMAUserInterface input
Output specification	Check if the correct methods are called in the TMAController
Environmental needs	S2I1 succeeded
Tested functional requirements	1. Check that when an action by the user is performed on the user interface the method sendMessage is called on TMAController.
Tested non functional requirements	1. (UI) Check that every functionality shall be reached surfing no more than 4 pages. 2. (Reliability) Check that the response time is smaller that 10 ms in 99% of times for local elaborations (not involving Internet connection).
Testing technique	Manual

Test case identifier	**S2I3-T1**
Test items	CCommunicator (CCommunicatorI) PMAUserInterface
Input specification	Create typical CCommunicator input
Output specification	Check if the correct methods are called in the PMAUserInterface
Environmental needs	S2I2 succeeded
Tested functional requirements	1. Check that when notify method is called on CCommunicator the method show is called on TMAUserInterface
Tested non functional requirements	-
Testing technique	Automated

3.1.3 [S3] WebTier

Test case identifier	**S3I1-T1**
Test items	MessageInterpreter CommandEventDispatcher (CommandDispatcher)
Input specification	Create typical MessageInterpreter input
Output specification	Check if the correct methods are called in the CommandEventDispatcher
Environmental needs	-
Tested functional requirements	1. Check that when interpretMessage is called on MessageInterpreter dispatchCommand is called on CommandEventDispatcher
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S3I2-T1**
Test items	SCommunicator (SCommunicatorI) MessageInterpreter
Input specification	Create typical SCommunicator input
Output specification	Check if the correct methods are called in the MessageInterpreter
Environmental needs	S3I1 succeeded
Tested functional requirements	1. Check that when submit is called on SCommunicator interpretMessage is called on MessageInterpreter
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S3I3-T1**
Test items	MessageFormatter SCommunicator (CommunicatorSender)
Input specification	Create typical MessageFormatter input
Output specification	Check if the correct methods are called in the SCommunicator
Environmental needs	S3I2 succeeded
Tested functional requirements	1. Check that when formatMessage is called on MessageFormatter send is called on SCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S3I4-T1**
Test items	CommandEventDispatcher (EventDispatcher) MessageFormatter
Input specification	Create typical CommandEventDispatcher input
Output specification	Check if the correct methods are called in the MessageFormatter
Environmental needs	S3I3 succeeded
Tested functional requirements	1. Check that when dispatchEvent is called on CommandEventDispatcher formatMessage is called on MessageFormatter.
Tested non functional requirements	-
Testing technique	Automated

3.1.4 [S5] TaxiManager

Test case identifier	**S5I1-T1**
Test items	TaxiQueueManager TaxiStateChanger
Input specification	Create typical TaxiQueueManager input
Output specification	Check if the correct methods are called in the TaxiStateChanger
Environmental needs	-
Tested functional requirements	1. Check that when move is called on TaxiQueueManager changeState is called on TaxiStateChanger. 2. Check that when moveToTheEnd is called on TaxiQueueManager changeState is called on TaxiStateChanger.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S5I2-T1**
Test items	TaxiSelector TaxiStateChanger
Input specification	Create typical TaxiSelector input
Output specification	Check if the correct methods are called in the TaxiStateChanger
Environmental needs	S5I1 succeeded
Tested functional requirements	1. Check that when confirm is called on TaxiSelector changeState is called on TaxiStateChanger.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S5I2-T2**
Test items	TaxiSelector TaxiPositionFinder
Input specification	Create typical TaxiSelector input
Output specification	Check if the correct methods are called in the TaxiPositionFinder
Environmental needs	S5I1 succeeded
Tested functional requirements	1. Check that when selectTaxi is called on TaxiSelector getTaxiPosition is called on TaxiPositionFinder.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S5I2-T3**
Test items	TaxiSelector TaxiQueueManager
Input specification	Create typical TaxiSelector input
Output specification	Check if the correct methods are called in the TaxiQueueManager
Environmental needs	S5I1 succeeded
Tested functional requirements	1. Check that when reject is called on TaxiSelector moveToTheEnd is called on TaxiQueueManager.
Tested non functional requirements	-
Testing technique	Automated

3.1.5 [S4] BusinessTier

Test case identifier	**S4I1-T1**
Test items	TaxiManager DBManager
Input specification	Create typical TaxiManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	Taxi manager integrated and tested
Tested functional requirements	1. Check that when all methods involving DB access are called on TaxiManager, method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I2-T1**
Test items	AccountManager DBManager
Input specification	Create typical AccountManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	-
Tested functional requirements	1. Check that when login, forgotPassword and register are called on AccountManager method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I3-T1**
Test items	RequestManager DBManager
Input specification	Create typical RequestManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	-
Tested functional requirements	1. Check that when sendRequest, getWaitingTime and getIncomingTaxiCode are called on RequestManager method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I3-T2**
Test items	RequestManager GoogleMapsCommunicator
Input specification	Create typical RequestManager input
Output specification	Check if the correct methods are called in the GoogleMapsCommunicator
Environmental needs	-
Tested functional requirements	1. Check that when sendRequest is called on RequestManager, validateAddress is called on GoogleMapsCommunicator. 2. Check that when getWaitingTime is called on RequestManager, getTravellingTime is called on GoogleMapsCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I3-T3**
Test items	RequestManager TaxiManager
Input specification	Create typical RequestManager input
Output specification	Check if the correct methods are called in the TaxiManager
Environmental needs	S4I1 succeeded
Tested functional requirements	1. Check that when sendRequest is called on RequestManager, selectTaxi is called on TaxiManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I4-T1**
Test items	ReservationManager GoogleMapsCommunicator
Input specification	Create typical ReservationManager input
Output specification	Check if the correct methods are called in the GoogleMapsCommunicator
Environmental needs	S4I3 succeeded
Tested functional requirements	1. Check that when sendReservation is called on ReservationManager, validateAddress is called on GoogleMapsCommunicator.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I4-T2**
Test items	ReservationManager RequestManager
Input specification	Create typical ReservationManager input
Output specification	Check if the correct methods are called in the RequestManager
Environmental needs	S4I3 succeeded
Tested functional requirements	1. Check that when sendReservation is called on ReservationManager, sendRequest is called on RequestManager.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**S4I4-T3**
Test items	ReservationManager DBManager
Input specification	Create typical ReservationManager input
Output specification	Check if the correct methods are called in the DBManager
Environmental needs	S4I3 succeeded
Tested functional requirements	1. Check that sendReservation, deleteReservation, modifyReservation, getReservation and getReservations are called on RequestManager method query is called on DBManager.
Tested non functional requirements	-
Testing technique	Automated

3.2 Subsystem level integration test

Test case identifier	**SI1-T1**
Test items	PMA → Web subsystem
Input specification	Create typical PMA input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	PMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when a user input is performed a message is received by Web subsystem through the network.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Manual

Test case identifier	**SI1-T2**
Test items	Web subsystem →PMA
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in PMA
Environmental needs	PMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when an event is generated as input of Web subsystem a message is recieved by PMA.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Automated

Test case identifier	**SI2-T1**
Test items	TMA → Web subsystem
Input specification	Create typical TMA input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	TMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when a user input is performed a message is recieved by Web subsystem through the network.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Manual

Test case identifier	**SI2-T2**
Test items	Web subsystem →TMA
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in TMA
Environmental needs	TMA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when an event is generated as input of Web subsystem a message is recieved by TMA.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Automated

Test case identifier	**SI3-T1**
Test items	PWA → Web subsystem
Input specification	Create typical PWA input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	PWA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when a user input is performed a message is recieved by Web subsystem through the network.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Manual

Test case identifier	**SI3-T2**
Test items	Web subsystem →PWA
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in PWA
Environmental needs	PWA and Web subsystem integrated and tested
Tested functional requirements	1. Check that when an event is generated as input of Web subsystem a message is received by PWA.
Tested non functional requirements	1. Check that remote interaction (via Internet) least at most 2 seconds and if not the connection is closed.
Testing technique	Automated

Test case identifier	**SI4-T1**
Test items	Business subsystem → DBMS
Input specification	Create typical Business subsystem input
Output specification	Check if the correct methods are called in DBMS
Environmental needs	Business subsystem integrated and tested
Tested functional requirements	1. Check that when a command which requires DB access is received by Business subsystem a query is performed on the DBMS.
Tested non functional requirements	1. Check that the system is a transactional system (both operations generated by users and internal operations carried out by system are transactions). 2. Check that the elaboration is carried out in less then 100 ms in 99% of times.
Testing technique	Automated

Test case identifier	**SI5-T1**
Test items	Web subsystem→ Business subsystem
Input specification	Create typical Web subsystem input
Output specification	Check if the correct methods are called in Business subsystem
Environmental needs	SI1, SI2, SI3 and SI4 succeeded
Tested functional requirements	1. Check that when a message is received by web subsystem a proper command is called on business subsystem.
Tested non functional requirements	-
Testing technique	Automated

Test case identifier	**SI5-T2**
Test items	Business subsystem → Web subsystem
Input specification	Create typical Business subsystem input
Output specification	Check if the correct methods are called in Web subsystem
Environmental needs	SI1, SI2, SI3 and SI4 succeeded
Tested functional requirements	1. Check that when an event is generated as a result of a previous command it is dispatched to the web subsystem.
Tested non functional requirements	-
Testing technique	Automated

4 Tools and test equipment required

In this section we present some tools that can be useful for the integration testing described in the previous sections. Those do not represent a constraint for the integration test execution phase but just a suggestion for the testing team.

4.1 Overview

Whatever approach is chosen for the integration testing, either *manual testing* or *automated testing*, it has to be at least compatible, preferably suitable, and simple for the architecture chosen in the implementative phase. As stated in the DD, the suggested architecture is JEE so we will describe possible approaches to integration testing advised for JEE.

- *Manual integration testing* refers to the testing process performed by hand.
- *Automated integration testing* refers to the testing process performed using the facilities of ad hoc testing tools, we will describe Arquillian that the main one adopted for JEE applications.

Automated testing should be preferred for reliability, precision and time saving (as it is done in unit testing) however for integration testing it is usually not enough. Sometimes automated tests cannot spot all forms of unexpected error conditions which can manifest only by a direct intervention of the developer who knows the dependency structure of the application.

4.2 Possible approaches

Mainly taken from [5].

4.2.1 Manual integration testing

Manual integration testing consists in the process of discovering defects in the interaction between components/subsystems by means of exercising the software with proper input from test cases and comparing the output with the expected output. Manual testing is typically slower and less reliable then automated testing however the "100% Automation is not possible" especially in integration testing so manual testing plays an important role in this context. Often there are not automatic tools to test the user interface therefore manual testing can be exploited.

4.2.2 Automated integration testing: Arquillian

Integration testing is very important in Java EE. The reason is two-fold:

- business components often interact with resources or sub-system provided by the container;
- many declarative services get applied to the business component at runtime.

Therefore to do integration tests on a JEE application it requires that you run them inside a JEE container.

Fortunately this problem is solved by an open source project called Arquillian (`http://www.jboss.org/arquillian`) which boots the container, allows the injection of managed beans and EJB beans into unit test classes. Arquillian is a *container-oriented testing framework* developed in Java Enterprise that brings your test to the runtime rather than requiring you to manage the runtime from the test. This strategy eliminates setup code and allows the test to behave more like

the components it's testing. The end result is that integration testing becomes no more complex than unit testing.

Arquillian combines a unit testing framework (JUnit or TestNG), ShrinkWrap, and one or more supported target containers (Java EE container, servlet container, Java SE CDI environment, etc) to provide a simple, flexible and pluggable integration testing environment. Beside Arquillian functionalities JUnit features (like assertions, annotations...) can be still exploited in order to write ad-hoc integration testing procedures. Other tools mainly intended to be used as unit test facilities can be exploited to build the necessary stub, one for all Mockito.

Figure 8: Arquillian architecture.

4.3 Suggested process

In order to exploit as much as possible the advantages, in terms of time and quality of results, of the automated integration testing we suggest to exploit manual testing only when strictly necessary, for instance in case of user interface integration testing. For all the other cases generating test data would be possible therefore automated tasting can be used. In section 3, for each test case, the suggested technique is specified.

5 Program stubs and test data required

5.1 Program drivers and stubs

In sections 2 and 3 we discussed the integration test strategies and the elements, either components or subsystems, to be integrated; we mentioned the need to use driver and stubs in order to set up the environment in which perform the test activity. In this section we present in details this infrastructure with reference to the test cases previously defined.

5.1.1 Component level integration testing

The strategy adopted for component level integration testing is *bottom-up* therefore drivers are necessary at each step of integration, in addition some stubs has to be used to simulate the functionalists of the other subsystems.

5.1.1.1 Drivers

Needed for each test case.

5.1.1.2 Stubs

Test id	*Items*	*Stubs*	*Interfaces*
S1I1	PMAController → CCommunicator	Web subsystem	SCommunicatorI
S1I2	PMAUserInterface → PMAController	CCommunicator	CCommunicatorI
S2I1	PMAController → CCommunicator	Web subsystem	SCommunicatorI
S2I2	PMAUserInterface → PMAController	CCommunicator	CCommunicatorI
S3I1	MessageInterpreter → CommandEventDispatcher	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
S3I3	MessageFormatter → SCommunicator	PMA, PMA	CCommunicatorI
S5I1	TaxiQueueManager → TaxiStateChanger	Business subsystem	DBManager
S5I2	TaxiSelector → TaxiStateChanger	Business subsystem	DBManager
S5I2	TaxiSelector → TaxiStateChanger	Business subsystem	DBManager
S4I1	TaxiManager → DBManager	DBMS	DBMSConnector
S4I2	AccountManager → DBManager	DBMS	DBMSConnector
S4I3	RequestManager → DBManager	DBMS	DBMSConnector
S4I4	ReservationManager → DBManager	DBMS	DBMSConnector

5.1.2 Subsystem level integration testing

The strategy adopted for component level integration testing is *sandwich* therefore both stubs and drivers are necessary at each step of integration.

5.1.2.1 Drivers

Test id	*Items*	*Driver*	*Functionality*
SI1	PMA → Web subsystem	-	-
SI1	Web subsystem → PMA	Business subsystem	Call methods of Web subsystem
SI2	TMA → Web subsystem	-	-
SI2	Web subsystem → TMA	Business subsystem	Call methods of Web subsystem
SI3	PWA →Web subsystem	-	-
SI3	Web subsystem → PWA	Business subsystem	Call methods of Web subsystem
SI4	Business subsystem → DBMS	Web subsystem	Call methods of Business subsystem
SI5	Web subsystem → Business subsystem	-	-
SI5	Business subsystem → Web subsystem	-	-

5.1.2.2 Stubs

Test id	*Items*	*Stubs*	*Interfaces*
SI1	PMA → Web subsystem	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
SI1	Web subsystem → PMA	-	-
SI2	TMA → Web subsystem	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
SI2	Web subsystem → TMA	-	-
SI3	PWA → Web subsystem	Business subsystem	RequestManager, ReservationManager, AccountManager, TaxiManager
SI3	Web subsystem → PWA	-	-
SI4	Business subsystem → DBMS	-	-
SI5	Web subsystem → Business subsystem	-	-
SI5	Business subsystem → Web subsystem	-	-

5.2 Test data requirements (external system integration)

In this subsection we briefly discuss the problem of integration testing with external subsystems, in particular we will focus on the integration with the GPS system in the smartphone of the passenger, the GoogleMapsAPI for address retrieval and the GPS system installed on taxis. For simplicity we assume those integration testing are performed before the integration testing of the system.

EI1 PMAController → GPSInterface

EI2 PMAController → GoogleMapsAPI

EI3 TMAController → GoogleMapsAPI

EI4 GoogleMapsCommunicator → GoogleMapsAPI

EI5 TaxiPositionFinder → TaxiGPSInterface

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.

Hours of works

Time spent by each group member:

- Alberto Maria Metelli: 12h
- Riccardo Mologni: 12h

Revision history

Version	*Date*	*Revision description*	*Revision notes*
0.1	15-1-2016	Initial draft	-
1.0	21-1-2016	Final draft	-
2.0	22-2-2016	Final release	Fixed introduction and some terminology.

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

PP
Project plan

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

Project management represents a necessary condition for the success of any software project. Considered the intrinsic complexity of a software project, the difficulty in assessing the quality of the software and the organizational, economical, social and technical issues to be tackled in any enterprise environment, project management cannot be avoided. It comprises all the activities aimed to ensuring the delivering of the software on schedule and in accordance with context and requirements; in particular project planning, reporting, risk management and people management. The *PP* (*Project Planning*) is intended to be a trace of the myTaxiService project process. For academic reasons, this document is delivered as last assignment but it refers to both activities to be completed before the project initiation and activities to be performed during and at the end of the project. In particular this document is focused on *project planning* (estimation and scheduling of the process development, assignment of resources), *risk management* (definitions, strategies to tackle risk) and *cost estimation*. In an enterprise environment the first two are typically performed before the project starts while the latter is executed either after the requirement engineering or after the design phase.

This document is intended to be read by stakeholders in order to show the devised organization of resources and time and to have a general overview of the effort needed to carry out the project useful for the evaluation of the founding.

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.
- SLOC: Source Lines Of Code.
- FP(s): Function Point(s).

Other acronyms are explained in the corresponding sections.

1.4 Reference documents

[1] The assignment of the *myTaxiService*.

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

[3] DD (Design Document) of the *myTaxiService*.

[4] ITPD (Integration Testing Plan Document) of the *myTaxiService*.

[5] Software Engineering 2 course slides.

[6] COCOMO II Model Definition Manual `http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf`

[7] Function Point Languages Table Version 5.0 `http://www.qsm.com/resources/function-point-language`

1.5 Document Structure

This document is composed of six sections and an appendix.

- The first section, this one, is intended to define the goal of the document, a very high level description of the main functionalists of the *myTaxiService* system and the resources used to draw up this document.
- The second section describes some preliminary results for cost estimation. A brief theoretical introduction about Function Points method will be provided and then it will be applied to the myTaxiService system in order to estimate the complexity of the project and derive the expected number of SLOC.
- The third section is devoted to cost estimation. A general method, called COCOMO, will be presented and applied to the specific case of myTaxiService in order to estimate the effort of the project, the estimated time needed for the fulfillment and the number of people required.
- In the fourth section we discuss project planning and in particular, according to the activities identified we propose a possible schedule, according to both the real deadlines and some reasonable considerations. We also provide a graphical representation of the schedule by means of a Gantt chart.
- The fifth section is still devoted to project planning but the focus is on the resource allocation. Some general consideration will be given and the allocation strategy explained, we will also use a resource chart to clarify the results.
- The sixth section is devoted to risk management. The main risks affecting the myTaxiService project will be stated according to the traditional classification. Some consideration about the strategy adopted to tackle them will be given.
- 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 history.

2 Function points

2.1 Introduction to cost estimation

Estimating the cost of a software project is a non trivial task. Many aspects contribute to determine the effort put in a software project and relationships among the characteristics of the project team (number of people, type of organization, ...), the features of the project itself (complexity) and the environmental influences are typically intricate and difficult to be estimated with an acceptable degree of approximation. Therefore often we rely on the *experience-based techniques* where the estimation is made by an experienced manager on the basis on the past projects and the application domain, sometimes this task is performed by a team of experts (community-based estimation). If no experts are available, or the domain of the application in too specific only *algorithmic cost modeling* is allowed. Those techniques are based on a formulated approach to compute the project effort based on estimates of product attributes. We will adopt the *Function Points* technique that was proposed by Allan Albrecht at IBM in 1965 and then COCOMOII that was developed around '80 based on the statistical analysis performed by Barry Boehn on the basis on many real projects coming from various domains.

2.2 Function points approach

There is an underlying hypothesis behind function points: the dimension of software can be characterized by *abstraction*. Therefore after the architectural design (early in the project life cycle), when the product model is almost clear, a rough evaluation of the size of the software can be performed. Albrecht's method identifies and counts the number of function types within the software (actually its model), those types constitutes the "external representation" of an application, that is, its functionality as defined from an abstract point of view. The types of functionalists are the following:

- *Internal Logical File* (ILF): homogeneous set of data used and managed by the application.
- *External Interface File* (EIF): homogeneous set of data used by the application but generated and maintained by other applications.
- *External Input*: elementary operation to elaborate data coming form the external environment.
- *External Output:* elementary operation that generates data for the external environment; it usually includes the elaboration of data from logic files.
- *External Inquiry*: elementary operation that involves input and output, without significant elaboration of data from logic files.

The measure of the size of the software is given by the weighed average of the function points (number of each function type listed above), according to the following predefined table.

Function type	**Weight**		
	Simple	*Average*	*Complex*
EI (External Inputs)	3	4	6
EO (External Outputs)	4	5	7
EQ (External Inquiries)	3	4	6
ILF (Internal Logical Files)	7	10	15
ELF (External Logical Files)	5	7	10

The resulting sum is called UFP (Unadjusted Function Points). This value can be further manipulated with the "adjustment" formula to get an estimation of the effort, however the result is usually little significant, therefore it is suggested to use the UFP in combination with other effort estimation algorithms like COCOMO II.

2.3 Function point count

In this section we present the function point count; for each type of functionality we list the ones we have identified within the myTaxiService system with the corresponding complexity and a rational for our choice. According to the Albrecht definition, this process is performed by abstraction therefore the main resource is the RASD in particular the high level class diagram for the logical files and the use cases for the inputs, outputs and inquiries.

2.3.1 ILF (Internal logical files)

The application includes a number of ILFs that will be used to store information about:

- *passengers*, in particular username, password, lastname, firstname, email and address;
- *taxi drivers*, in particular username, password, firstname and lastname;
- *taxis*, in particular the plate number, the code, the number of seats and the current state;
- *requests*, in particular the date and time in which the request is sent, the number of passengers, the location (geographical coordinates and the address) of the passenger, possibly the waiting time and the taxis assigned to the request itself;
- *reservations*, in particular the date and time in which the request is sent, the number of passengers, the location (geographical coordinates and the address) of both origin and destination and the corresponding attached request;
- *zones*, in particular the name of the zone, the estimation of the requests per minute and the adjacency relation among zones;
- *queues*: in particular the proper size of the queue (and also the minimum and maximum number of taxis allowed) and the taxis belonging to the queue with the corresponding position.

Each of these entities has a simple structure with a limited number of fields except for the queue which requires a quite articulated structure to manage positions and sizes of queues. Thus, we select medium complexity for the latter and simple for the other ones.

$$ILF = 6 \cdot 7 + 1 \cdot 10 = 52FPs$$

2.3.2 ELF (External logical files)

myTaxiService has to interact with external systems, in particular with the GPS and the GoogleMaps API therefore we can identify the following ELFs:

- *Address validation* requires the interaction with the GoogleMaps API in order to check whether a string typed by the user corresponds to a valid address.
- *Coordinate/Address translation* requires the interaction with the GoogleMaps API in order to convert a coordinate retrieved by the GPS into an address and viceversa.

- *Travelling time* requires the interaction with the GoogleMaps API in order to compute the waiting time.
- *Passenger's geographic coordinates* requires the interaction with the GPS system installed on the passenger smartphone or with the browser geolocalization to retrieve the passenger's position.
- *Taxi geographic coordinates* requires the interaction with the GPS installed on each taxi in order to find the position of the taxi itself.

All those data items are very compact and they share the same simple structure, so a simple complexity should be appropriate.

$$ELF = 5 \cdot 5 = 25FPs$$

2.3.3 EI (External inputs)

Since myTaxiService is an application characterized of having a high degree of interaction with the final user, we can identify the following EI.

- *Registration*: this function requires the exchange of a relevant amount of information (username, password, lastname, firstname, email and address), in addition some checks has to be performed (like check that the username is not already used), so we can consider medium complexity with a contribution of 4 FPs.
- *Login/Logout*: this function is standard and requires exchange of basic structured information and simple operations, so it can be considered simple with a contribution of 3 FPs.
- *Request*: this function requires the insertion of some data (like address), the interaction with external systems (like GPS and GoogleMaps API) and with the DBMS, therefore it can be considered complex with a contribution of 6 FPs.
- *Taxi selection*: this function requires non trivial elaborations related to the algorithm used to select the taxis to fulfill a request/reservation; it requires interaction with the DBMS and external systems therefore it can be considered complex with a contribution of 6 FPs.
- *Taxi queue management*: this function requires the execution of the algorithm described in the design document for the taxi movement, in addition it requires to interact with external systems and the DBMS therefore it can be considered complex with a contribution of 6 FPs.
- *Reservation*: this function requires interaction with the DBMS and with external systems, it has also to instantiate a new request associated to the reservation and to perform validity checks on the inserted data, therefore it can be considered complex with a contribution of 6 FPs.
- *Modify reservation*: this function can be considered an extension of Reservation, adding a small new functionality, therefore it can be considered simple with a contribution of 3 FPs.
- *Cancel reservation*: this function can be considered an extension of Reservation, adding a small new functionality, therefore it can be considered simple with a contribution of 3 FPs.
- *Request evaluation*: this function requires to interact with the TMA and the analysis of the taxi queues, so it can be considered medium complexity with a contribution of 4 FPs.
- *Inform about availability*: this function requires to check some conditions to validate the state changing of the taxi driver, it can be considered medium complexity with a contribution of 4 FPs.

- *Insert phone request*: this function is just reduced to Request therefore it can be considered simple with a contribution of 3 FPs.

$$EI = 4 \cdot 3 + 3 \cdot 4 + 4 \cdot 6 = 48FPs$$

2.3.4 EO External output

The only external outputs generated by the myTaxiService are:

- *Movement notification:* that function is performed by the system to communicate to the taxi driver the notification of the movement, since it requires the evaluation of the taxi queues it can be considered medium complexity with a contribution of 5 FPs.
- *Request notification:* this function is performed in order to communicate to the taxi driver a request to be carried out, it requires only to send information about the address of the passenger therefore it can be considered simple with a contribution of 4 FPs.

$$EI = 1 \cdot 4 + 1 \cdot 5 = 9FPs$$

2.3.5 EQ External inquiries

We identified the following EQs.

- *Visualize request info*: this function allows passenger, both registered or not, to visualize the information about the last request including waiting time and the number of incoming taxi.
- *Visualize previous reservations*: this function allows registered passengers to visualize the previous sent reservations.
- *Visualize previous requests (taxi driver)*: this function allows the taxi driver to visualize the previous requests carried out.

All these functions requires the interaction with the DBMS therefore they can be considered medium complexity.

$$EI = 3 \cdot 4 = 12FPs$$

2.3.6 UFP Unadjusted Function Points count

Here we summarize the number of function points identified in every category and we provide the UFP (Unadjusted Function Points count) which could possibly be adjusted to take into account organizational aspects and get an estimation of the effort, however this approach is typically very imprecise therefore we will use UFP in combination with the COCOMO approach.

Function type	**FPs**
ILF (Internal Logical Files)	52
ELF (External Logical Files)	25
EI (External Inputs)	48
EO (External Outputs)	9
EQ (External Inquiries)	12

Thus, we get at the end the value of UFP

$$UFP = ILF + ELF + EI + EO + EQ = 146FPs$$

Here we provide an histogram representing the number of function points for each category.

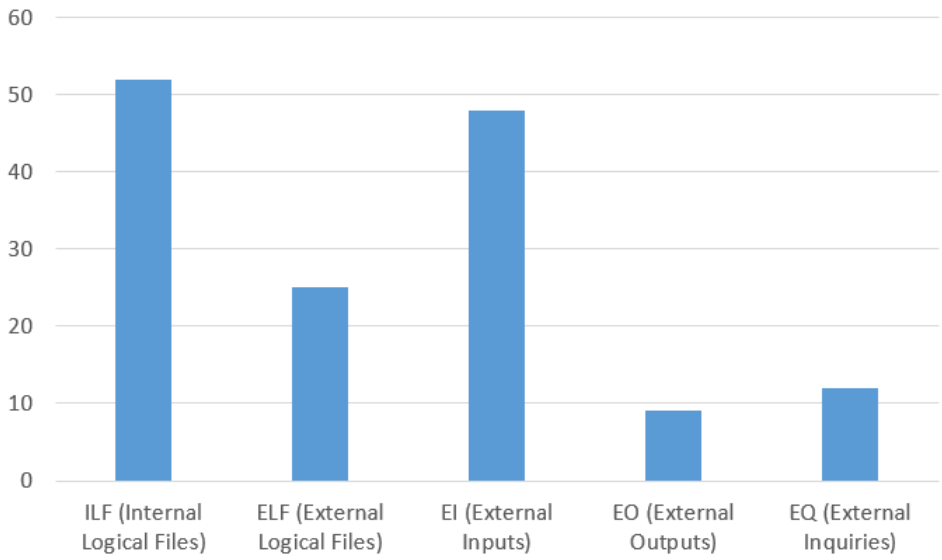


Figure 1: Function Points Histogram

2.4 Lines of code count

The Unadjusted Function Points (UFP) can be used to provide an estimation of the number of lines of code (SLOC) of the final project according to the specific implementation language chosen. Instead of referring to the "traditional" table proposed by Jones 1996 for the conversion factor, which by the way does not include a value for JEE, we adopt the more recent Function Points Language Table proposed in [7]. myTaxiService is intended to be implemented with JEE, the table provides a range for the conversion factor SLOC/FP

Language	*Average*	*Median*	*Low*	*High*
J2EE	46	49	15	67

Not having precise information about the complexity of the implementation we think an average value should be proper, so $SLOC/FP = 46$ and we get the value of SLOC.

$$SLOC = SLOC/FP \cdot UFP = 46 \cdot 146 = 6716$$

Notice that this value does not capture, for the most part, the client side of the application and the web server. Since both PMA, TMA and web server do not perform meaningful functions at requirement level but just auxiliary functions they are not highlighted in the function points count, however from an implementation point of view they might require a relevant effort.

3 COCOMO II

In this section, after a brief introduction to COCOMO II, we present the application of the method to myTaxiService system in order to get the effort estimation and the expected duration of the project.

3.1 COCOMOII approach

Mainly taken from [6].

COCOMO II is an updated version of COCOMO 81 which is based on a linear regression model of the function relating the size of the project and the effort. However the original version was affected by old time and non realistic assumptions on the *software lifecycle*, in particular it assumed that the development process followed the traditional waterfall scheme, that requirements were supposed to be stable during the project duration and that the documentation was written incrementally. COCOMO II is able to overcome those limitation and provide a reasonable estimation of both the effort and the duration of the project. In the following subsection we will describe the model.

3.1.1 Effort Equation

In COCOMO II *effort* is expressed as *Person-Months* (PM). [1]The *Effort equation* allows to compute the effort as a function of the *size* of software development expressed in KSLOC (thousand of SLOC)[2] that can be obtained by the Function Points method, a constant, A, an exponent, E, and a number of values called *effort multipliers* (EM). The number of effort multipliers depends on the model.

$$Effort = PM = A \cdot Size^{E} \cdot \prod_{i=1}^{n} EM_i$$

where $A = 2.94$ for COCOMO II. We now present how to compute E and each EM_i[3].

3.1.2 Software Scale Drivers

The exponent E in the effort equation is an aggregation of five *scale factors* (SF, also referred to *scale drivers* SD) is related to economic, organizational and technical aspects of the environment in which the project is going to be developed. In particular if account for the relative economies or diseconomies of scale encountered for software projects of different sizes [Banker et al. 1994]. [4]

The value of E is a contribution of the following scale factors.

PREC *Precedentedness*: reflects the previous experience of the organization with this type of project. Very low means no previous experience, Extra high means that the organization is completely familiar with this application domain.

[1]A person month is the amount of time one person spends working on the software development project for one month. COCOMO II treats the number of person-hours per person-month, PH/PM, as an adjustable factor with a nominal value of 152 hours per Person-Month.

[2]The SLOC in COCOMO represent only the line of codes that are going to be delivered and they have to be computed as *logical* SLOC.

[3]Sometimes the overall contribution of the EM_i is called EAF (Effort Adjustment Factor) which is trivially defined as $EAF = \prod_{i=i}^{n} EM_i$.

[4]If E<1.0, the project exhibits economies of scale. If the product's size is doubled, the project effort is less than doubled. The project's productivity increases as the product size is increased. Some project economies of scale can be achieved via project-specific tools (e.g., simulations, testbeds). For small projects, fixed start-up costs such as tool tailoring and setup of standards and administrative reports are often a source of economies of scale. If E=1.0, the economies and diseconomies of scale are in balance. This linear model is often used for cost estimation of small projects. If E>1.0, the project exhibits diseconomies of scale.

FLEX *Development Flexibility*: reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client only sets general goals.

RESL *Architecture / Risk Resolution*: reflects the extent of risk analysis carried out. Very low means little analysis, Extra high means a complete a thorough risk analysis.

TEAM *Team Cohesion*: reflects how well the development team know each other and work together. Very low means very difficult interactions, Extra high means an integrated and effective team with no communication problems.

PMAT *Process Maturity*: reflects the process maturity of the organization. The computation of this value depends on the CMM (Capability Maturity Model)[5] Questionnaire but an estimate can be achieved by subtracting the CMM process maturity level from 5.

The following table provides the rating levels for the COCOMO II scale factors. The selection of scale factors is based on the rationale that they are a significant source of exponential variation on a project's effort or productivity variation. Each scale factors has a range of rating levels, from Very Low to Extra High and each rating level has a weight.

Scale Factor	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
PREC	thoroughly unprecedented **6.20**	largely unprecedented **4.96**	somewhat unprecedented **3.72**	generally familiar **2.48**	largely familiar **1.24**	thoroughly familiar **0.00**
FLEX	rigorous **5.07**	occasional relaxation **4.05**	some relaxation **3.04**	general conformity **2.03**	some conformity **1.01**	general goals **0.00**
RESL	little (20%) **7.07**	some (40%) **5.65**	often (60%) **4.24**	generally (75%) **2.83**	mostly (90%) **1.41**	full (100%) **0.00**
TEAM	very difficult interactions **5.48**	some difficult interactions **4.38**	basically cooperative **3.29**	interactions largely cooperative **2.19**	highly cooperative **1.10**	seamless interactions **0.00**
PMAT	SW-CMM Level 1 Lower **7.80**	SW-CMM Level 1 Upper **6.24**	SW-CMM Level 2 **4.68**	SW-CMM Level 3 **3.12**	SW-CMM Level 4 **1.56**	SW-CMM Level 5 **0.00**

To have a complete description of the meaning of each scale factor, please refer to [6] where also assessment methods are proposed.

Once the value of each scale factor SF_j is determined the value of the exponent E can be computed using the following formula.

$$E = B + 0.01 \cdot \sum_{j=i}^{5} SF_j$$

[5]CMM is a development model created to represent the "maturity" of software development process as the degree of formality and optimization of processes, from ad hoc practices, to formally defined steps, to managed result metrics, to active optimization of the processes. Nowadays the extension CMMI (Capability Maturity Model Integration) is used.

where $B = 0.91$ for COCOMO II. Notice that if all scale factors are judged as Nominal the resulting exponent value is $E = 1.0997$.

3.1.3 Software Cost Drivers

Cost drivers are used to capture characteristics of the software development that affect the effort to complete the project. All COCOMO II cost drivers have qualitative rating levels that express the impact of the driver on development effort. These ratings can range from Extra Low to Extra High. Each rating level of every multiplicative cost driver has a value, called an *effort multiplier* (EM), associated with it. The EM value assigned to a multiplicative cost driver's nominal rating is 1.00. All those are intended to be evaluated in post-architecture/early-design phase and they are organized in categories, for the complete description refer to [6].

- *Product* factors account for variation in the effort required to develop software caused by characteristics of the product under development. A product that is complex, has high reliability requirements, or works with a large testing database will require more effort to complete.

RELY	Required Software Reliability
DATA	Data Base Size
CPLEX	Product Complexity
RUSE	Developed for Reusability
DOCU	Documentation Match to Lifecycle Needs

- *Personnel* factors have the strongest influence in determining the amount of effort required to develop a software product. The Personnel Factors are for rating the development team's capability and experience – not the individual. These ratings are most likely to change during the course of a project reflecting the gaining of experience or the rotation of people onto and off the project.

ACAP	Analyst Capability
PCAP	Programmer Capability
PCON	Personnel Continuity
AEXP	Application Experience
PEXP	Platform Experience
LTEX	Language and Toolset Experience

- *Platform* factors refers to the target-machine complex of hardware and infrastructure software (previously called the virtual machine). The factors have been revised to reflect this as described in this section. Some additional platform factors were considered, such as distribution, parallelism, embeddedness, and real-time operations.

TIME	Time Constraint
STOR	Storage constraint
PVOL	Platform Volatility

- *Project* factors account for influences on the estimated effort such as use of modern software tools, location of the development team, and compression of the project schedule.

TOOL	Use of Software Tools
SITE	Multisite Development
SCED	Required Development Schedule

3.1.4 Schedule estimation

The *Schedule equation* allows to determine the *Time to Develop*, TDEV, expressed in number of months required to complete the software project.

$$Duration = TDEV = C \cdot (PM)^{(D+0.2(E-B))}$$

where $C = 3.67$, $D = 0.28$, $B = 0.91$.

C is a TDEV coefficient that can be calibrated, PM is the estimated effort, D is a TDEV scaling base exponent that can also be calibrated. E is the effort scaling exponent derived as the sum of project scale factors and B as the calibrated scale factor base-exponent. If we also know the average monthly salary S of the personnel we can estimate the personnel cost.[6]

$$Cost = PM \cdot S$$

3.1.5 Number of people estimation

Using PM and TDEV calculated before we can give an estimation of the number of required people.

$$N = \frac{PM}{TDEV}$$

3.2 COCOMO II calculation

In this subsection we perform the COCOMO II estimation starting from the SLOC determined in the previous stage thanks to Function Points approach. Given the academic context and the fact that implementation was not performed we will try to both perform the calculation with

- all SFs and CDs rated as nominal and
- SFs and CDs rated according our opinion with respect to a reasonable implementation phase.

Then we compere the results obtained by the two different approaches. We also assume that each person involved in the project is payed 2500$ per month. To easily compute the PM and TDEV we refer to the online tool `http://csse.usc.edu/tools/COCOMOII.php`.

3.2.1 COCOMO II with all SFs and CDs nominal

If we assume to set all SFs and CDs to nominal we get the parameters $E = 1.0997$ and $EAF = 1$ therefore knowing that $Size = KSLOC = 6.716$ we get the following results.

$$Effort = PM = 2.94 \cdot (6.716)^{1.0997} \cdot 1 = 23.9 \text{ person} - \text{month}$$

$$Duration = TDEV = 3.67 \cdot (23.9)^{0.31794} = 10.5 \text{ months}$$

$$Cost = 23.9 \cdot 2500\$ = 59684\ \$$$

[6] Notice that this is very different from the overall cost of the project since we do not take into account overheads (buildings, heating, lighting, ...).

$$N = \frac{23.9}{10.5} = 2.276 \text{ person}$$

We report some tables and graphs taken from the web site.

Acquisition Phase Distribution

Phase	*Effort (Person-months)*	*Schedule (Months)*	*Average Staff*	*Cost (Dollars)*
Inception	1.4	1.3	1.1	3581
Elaboration	5.7	3.9	1.5	14324
Construction	18.1	6.5	2.8	45360
Transition	2.9	1.3	2.2	7162

Software Effort Distribution for RUP/MBASE (Person-Months)

Phase/Activity	Inception	Elaboration	Construction	Transition
Management	0.2	0.7	1.8	0.4
Environment/CM	0.1	0.5	0.9	0.1
Requirements	0.5	1.0	1.5	0.1
Design	0.3	2.1	2.9	0.1
Implementation	0.1	0.7	6.2	0.5
Assessment	0.1	0.6	4.4	0.7
Deployment	0.0	0.2	0.5	0.6

Staffing profile

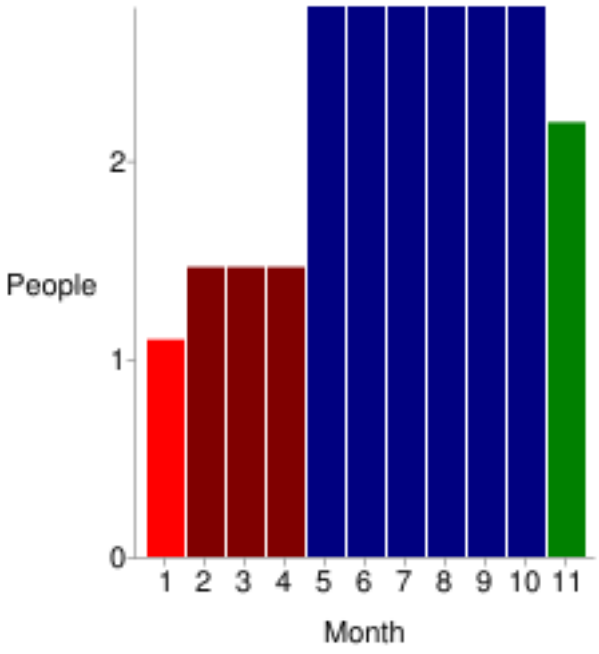


Figure 2: COCOMO II histogram - all nominal

From now on we will discuss how to rate Scale Drivers and Cost Drivers for myTaxiService.

3.2.2 Software Scale Drivers

In the following table we describe for each scale driver the motivations behind the rating level choice, we refer to the descriptors stated in [6].

Scale Factor	*Motivation*	*Rating level*	*Weight*
PREC	• Organizational understanding of product objectives: Thorough • Experience in working with related software systems: Moderate • Concurrent development of associated new hardware and operational procedures: Moderate • Need for innovative data processing architectures, algorithms: Some	Nominal	3.72
FLEX	• Need for software conformance with preestablished requirements: Basic (many aspects were not clearly specified) • Need for software conformance with external interface specifications: Considerable (integration with some external systems needed) • Combination of inflexibilities above with premium on early completion: High	High	2.03
RESL	• Percent of development schedule devoted to establishing architecture, given general product objectives: 25 • Percent of required top software architects available to project: 80 • Tool support available for resolving risk items, developing and verifying architectural specs: Some • Level of uncertainty in key architecture drivers: mission, user interface, COTS, hardware, technology, performance: Considerable • Number and criticality of risk items: Some	High	2.83

Scale Factor	*Motivation*	*Rating level*	*Weight*
TEAM	• Consistency of stakeholder objectives and cultures: Strong • Ability, willingness of stakeholders to accommodate other stakeholders' objectives: Considerable • Experience of stakeholders in operating as a team: Considerable • Stakeholder teambuilding to achieve shared vision and commitments: Considerable	Very High	1.10
PMAT	Answering to the KPAs (Key Process Area) we get a value of KPA thus the corresponding EPML (Equivalent Processing Maturity Level) is 1.5	Nominal	4.68

We obtain a total value for the exponent E of

$$E = B + 0.01 \cdot \sum_{j=i}^{5} SF_j = 1.0536$$

Notice that the value of the exponent is smaller with respect to the one obtained setting all SFs to nominal, this means that we are benefiting of the quality of the environment in which the software is developed.

3.2.3 Software Cost Drivers

In the following table we describe for each cost driver the motivations behind the rating level choice, we refer to the descriptors stated in [6]. Whenever the information for selecting the most appropriate rating level is not available we adopt nominal value.

Cost Driver	*Motivation*	*Rating level*	*Effort multiplier*
RELY	Moderate, easily recoverable losses.	Nominal	1.00
DATA	10 ≤Testing DB bytes/Pgm SLOC<100	Nominal	1.00
CPLEX	• *Control Operations*: Highly nested structured programming operators with many compound predicates. Queue and stack control. Homogeneous, distributed processing. Single processor soft real-time control. • *Computational Operations:* Use of standard math and statistical routines. Basic matrix/vector operations. • *Device dependent Operations*: Operations at physical I/O level (physical storage address translations; seeks, reads, etc.). Optimized I/O overlap. • *Data Management Operations*: Multi-file input and single file output. Simple structural changes, simple edits. Complex COTS-DB queries, updates. • *User Interface Management Operations*: Simple use of widget set.	High	1.17
RUSE	Across program[7]	High	1.07
DOCU	Right-sized to life-cycle needs	Nominal	1.00
ACAP	Analysts in the 75th percentile	High	0.85
PCAP	Programmes in the 75th percentile	High	0.88
PCON	Personell turnover 3% / year	Very High	0.81
AEXP	2 months	Very Low	1.22
PEXP	2 months	Very Low	1.19
LTEX	1 year	Nominal	1.00
TIME	85% use of available execution time	Very High	1.29
STOR	50% use of available storage	Nominal	1.00
PVOL	Major change every 12 mo.; Minor change every 1 mo.	Low	0.87
TOOL	basic lifecycle tools, moderately integrated	Nominal	1.00
SITE	• *Collocation Descriptors*: Same city or metro. area • *Communications Descriptors*: Wideband electronic communication.	High	0.93
SCED	Shedule compression 130% of nominal	High	1.00

We obtain a total value for the EAF of

$$EAF = \prod_{i=i}^{n} EM_i = 1.149$$

Note that the value of EAF is grater than the one obtained assuming all CDs as nominal, this mean that CDs turn out to affect negatively the overall effort.

3.2.4 Effort estimation and schedule estimation

Considering the values computed above, $E = 1.0536$ and $EAF = 1.149$, and knowing that $Size = KSLOC = 6.716$ we get the following results.

$$Effort = PM = 2.94 \cdot (6.716)^{1.0536} \cdot 1.149 = 25.1 \text{ person} - \text{month}$$

$$Duration = TDEV = 3.67 \cdot (25.1)^{0.30872} = 13.8 \text{ months}$$

$$Cost = 25.1 \cdot 2500\$ = 62832\ \$$$

$$N = \frac{25.1}{13.8} = 1.82 \text{ person}$$

Note that this second result is just slightly different with respect to the one obtained with all nominal SFs and CDs, in fact the effort here is grater of 1.2 person-month and also the duration is increased of 3.3 months; however the required number of people is still around 2.

We report some tables and graphs taken from the web site.

Acquisition Phase Distribution

Phase	*Effort (Person-months)*	*Schedule (Months)*	*Average Staff*	*Cost (Dollars)*
Inception	1.5	1.7	0.9	3770
Elaboration	6.0	52.	1.2	15080
Construction	19.1	8.6	2.2	47753
Transition	3.0	1.7	1.7	7540

Software Effort Distribution for RUP/MBASE (Person-Months)

Phase/Activity	Inception	Elaboration	Construction	Transition
Management	0.2	0.7	1.9	0.4
Environment/CM	0.2	0.5	1.0	0.2
Requirements	0.6	1.1	1.5	0.1
Design	0.3	2.2	3.1	0.1
Implementation	0.1	0.8	6.5	0.6
Assessment	0.1	0.6	4.6	0.7
Deployment	0.0	0.2	0.6	0.6

Staffing profile

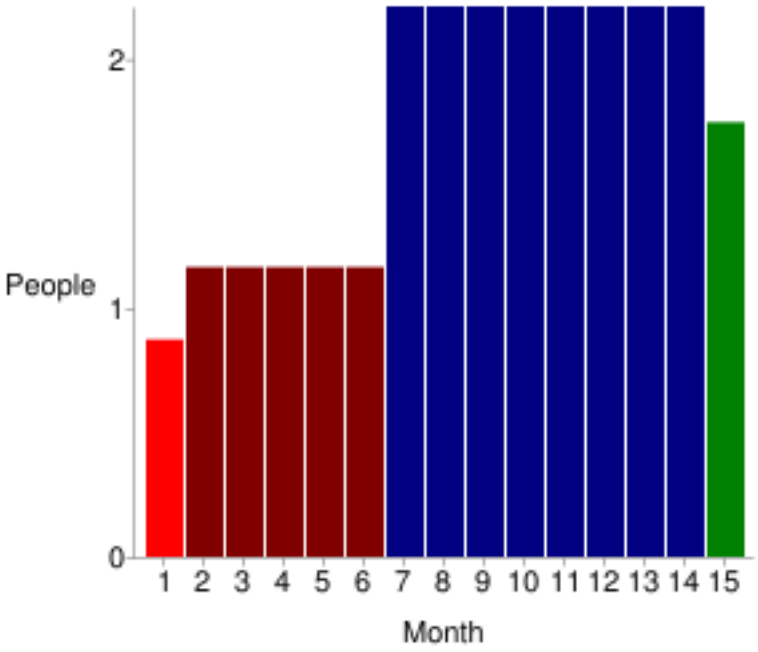


Figure 3: COCOMO II histogram

4 Tasks and schedule

Project planning is typically a in iterative process that starts at project initiation and continues evolving. In order to set up a suitable *project schedule*, and in particular decide the time and resource allocation, we need to identify:

- *tasks*: activities that must be completed to achieve the project goal and the corresponding dependencies;
- *milestones*: points in time in which you can assess the progress of the project;
- *deliverables*: work products delivered to the stakeholders.

This phase has to be driven by the *software development process* chosen and by the main features of the specific application we are going to design. In this section we provide a description of the tasks, their dependencies and a graphical representation of a schedule proposal.

4.1 Tasks

The following schedule tries to be a compromise between the need of producing a realistic program of the tasks and be compatible with the tasks we really performed during the semester. For what concerns the tasks related to RASD, DD and ITDP the dates are the real ones, for PP we assumed to having performed it as the first activity (as it is in a realistic context). Since we didn't perform the implementation of the software we couldn't exploit real data; to be realistic we allocate a duration of about three months. Also for code inspection and testing we choose reasonable duration. We can interpret this program as a schedule for an hypothetical annual project assigned to us at the beginning of the academic year. We didn't use the duration value provided by COCOMO II analysis since, also according to the applications made in the examples by the previous years, it tends to overestimate project duration and not to be coherent with the "breakneck speed" we are asked to keep at Politecnico.

The following table shows for each task the id, name, star and end date, the duration expressed in working days and the possible precedences. Task are grouped into macro tasks.

Task id	*Name*	*Start*	*End*	*Duration*	*Precendences*
T1	**Project planning**	01/10/2015	14/10/2015	10	-
T11	Project scheduling	01/10/2015	09/10/2015	7	-
T12	Resource allocation	12/10/2015	14/10/2015	3	T11
T13	Risk planning	05/10/2015	14/10/2015	8	-
M1	PP	15/10/2015	15/10/2015	-	
T2	**Requirement analysis and specification**	**15/10/2015**	**02/11/2015**	**13**	M1
T21	Requirement engineering	15/10/2015	21/10/2015	5	
T22	Use case design	22/10/2015	02/11/2015	8	T22
T23	High level data design	26/10/2015	28/10/2015	3	
T24	Alloy modeling	29/10/2015	02/11/2015	3	T23
M2	RASD	03/11/2015	03/11/2015	-	
T3	**Acceptance test plan design**	**03/11/2015**	**11/11/2015**	**7**	M2
T4	**Design**	**09/11/2015**	**04/12/2015**	**20**	M2
T41	Architectural design	09/11/2015	04/12/2015	20	
T42	Algorithm design	16/11/2015	04/12/2015	15	
T43	User interface design	30/11/2015	04/12/2015	5	
M3	DD	07/12/2015	07/12/2015	-	
T5	**Unit test plan design**	**07/12/2015**	**15/12/2015**	**7**	M2
T6	**Integration test plan design**	**07/01/2016**	**21/01/2016**	**11**	M2
M4	ITDP	22/01/2016	22/01/2016	-	
T7	**Implementation**	**07/12/2015**	**05/03/2016**	**65**	M2
T71	Components implementation	07/12/2015	05/03/2016	65	
T72	Subsystem implementation	08/02/2016	04/03/2016	20	
T8	**Code Inspection**	**07/03/2016**	**23/03/2016**	**13**	
T81	Manual inspection	07/03/2016	17/03/2016	9	T71
T82	Automated code inspection	18/03/2016	23/03/2016	4	T81
M5	CI	24/03/2016	24/03/2016	-	
T9	**Testing**	**24/03/2016**	**28/04/2016**	**26**	
T91	Unit testing	24/03/2016	08/04/2016	12	T5, T71
T92	Integration testing	11/04/2016	18/04/2016	6	T91, T6, T7
T93	System and performance testing	19/04/2016	20/04/2016	2	T92
T94	Acceptance testing	21/04/2016	28/04/2016	6	T93, T3
T10	**Deployment**	**29/04/2016**	**04/05/2016**	**4**	T9

The following is the same table in which just macro tasks and the delivarables are represented.

Task id	*Name*	*Start*	*End*	*Duration*
T1	**Project planning**	01/10/2015	14/10/2015	10
M1	PP	15/10/2015	15/10/2015	-
T2	**Requirement analysis and specification**	**15/10/2015**	**02/11/2015**	**13**
M2	RASD	03/11/2015	03/11/2015	-
T3	**Acceptance test plan design**	**03/11/2015**	**11/11/2015**	**7**
T4	**Design**	**09/11/2015**	**04/12/2015**	**20**
M3	DD	07/12/2015	07/12/2015	-
T5	**Unit test plan design**	**07/12/2015**	**15/12/2015**	**7**
T6	**Integration test plan design**	**07/01/2016**	**21/01/2016**	**11**
M4	ITDP	22/01/2016	22/01/2016	-
T7	**Implementation**	**07/12/2015**	**05/03/2016**	**65**
T8	**Code Inspection**	**07/03/2016**	**23/03/2016**	**13**
M5	CI	24/03/2016	24/03/2016	-
T9	**Testing**	**24/03/2016**	**28/04/2016**	**26**
T10	**Deployment**	**29/04/2016**	**04/05/2016**	**4**

4.2 Milestones

Here are the expected milestone of the project.

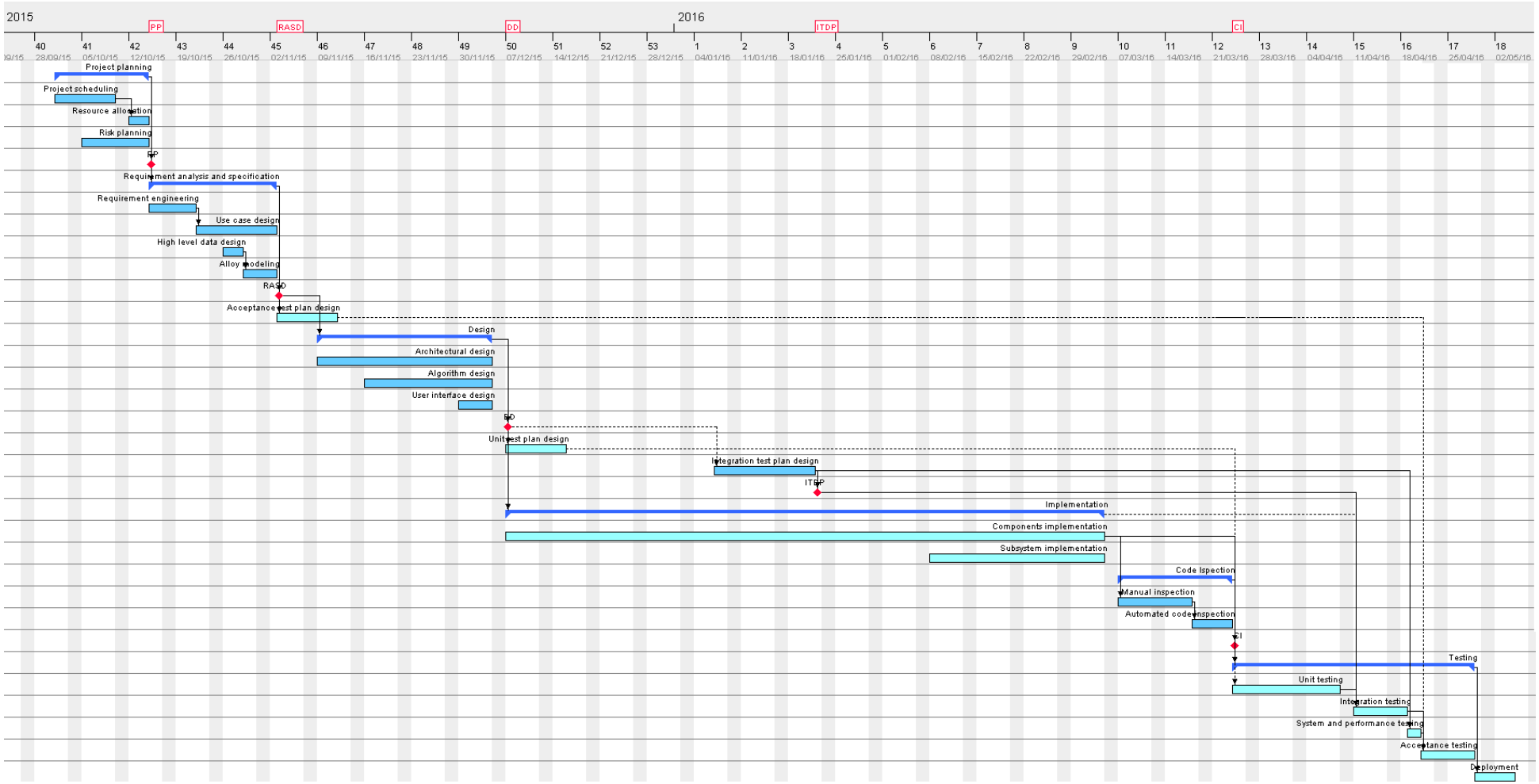
1. after RASD
2. after DD
3. after CI
4. after implementation
5. after testing
6. after deployment

4.3 Deliverables

Here are the expected deliverables of the project; the ones we really produced in the semester are highlighted in red in the previous table (actually we didn't perform code inspection on the myTaxiService but we included it as a deliverable anyway).

1. RASD (Requirement Analysis and Specification Document)
2. DD (Design Document)
3. CI (Code Inspection Report)
4. Code
5. UTPD (Unit Test Design Plan)
6. ITPD (Integration Test Design Plan)
7. User manual

4.4 Gantt chart



5 Resources

In this section we present the *resource allocation*. For each member of the team we will associate the tasks, as defined in the previous section, to be executed with reference to the available time of each team member. Since all team members contributed to the project playing different roles in the context of the software development cycle we will specify that role time by time.

5.1 Resource allocation

For the task we actually performed the hours specified are the real ones, for the others we estimated a reasonable allocation of time also according to our availability being university students. Moreover, according to our previous experiences, we qualified the implementation and the testing phases as the most time consuming.

The following table shows for each task the role played by the team member and the hours allocated; the attribute "Units", as commonly used in resource planning is the percentage ratio between the used hours and the allocated hours computed as the duration in days times 8 hours per day. As emerges from the table those values are by far lower than 50% because of our university activity.

Task id	*Name*	*Duration*	***Riccardo Mologni***			***Alberto Maria Metelli***		
			Role	*Hours*	*Units*	*Role*	*Hours*	*Units*
T1	**Project planning**	10	Project Manager	10	12,5%	Project Manager	10	12,5%
T11	Project scheduling	7	Project Manager	5	8,9%	Project Manager	3	5,4%
T12	Resource allocation	3	Project Manager	3	12,5%	Project Manager	3	12,5%
T13	Risk planning	8	Project Manager	2	3,1%	Project Manager	4	6,3%
T2	**Requirement analysis and specification**	13	Analyst	33	31,7%	Analyst	33	31,7%
T21	Requirement engineering	5	Requirements engineer	12	30,0%	Requirements engineer	12	30,0%
T22	Use case design	8	Analyst	15	23,4%	Analyst	11	17,2%
T23	High level data design	3	Analyst	3	12,5%	Analyst	3	12,5%
T24	Alloy modeling	3	Analyst	3	12,5%	Analyst	7	29,2%
T3	**Acceptance test plan design**	7	Analyst	8	14,3%	Analyst	8	14,3%
T4	**Design**	20	Software Architect	30	18,8%	Software Architect	35	21,9%
T41	Architectural design	20	Software Architect	15	9,4%	Software Architect	15	9,4%
T42	Algorithm design	15	Software Architect	7	5,8%	Software Architect	15	12,5%
T43	User interface design	5	Software Architect	8	20,0%	Software Architect	5	12,5%
T5	**Unit test plan design**	7	Analyst	13	23,2%	Analyst	13	23,2%
T6	**Integration test plan design**	11	Analyst	12	13,6%	Analyst	12	13,6%
T7	**Implementation**	65	Developer	260	50,0%	Developer	260	50,0%
T71	Components implementation	65	Developer	200	38,5%	Developer	200	38,5%
T72	Subsystem implementation	20	Developer	60	37,5%	Developer	60	37,5%
T8	**Code Ispection**	13	Ispector	15	14,4%	Ispector	18	17,3%
T81	Manual inspection	9	Ispector	13	18,1%	Ispector	13	18,1%
T82	Automated code inspection	4	Ispector	2	6,3%	Ispector	5	15,6%
T9	**Testing**	26	Tester	86	41,3%	Tester	86	41,3%
T91	Unit testing	12	Tester	45	46,9%	Tester	45	46,9%
T92	Integration testing	6	Tester	25	52,1%	Tester	25	52,1%
T93	System and performance testing	2	Tester	6	37,5%	Tester	6	37,5%
T94	Acceptance testing	6	Tester	10	20,8%	Tester	10	20,8%
T10	**Deployment**	4	Installer	15	46,9%	Installer	15	46,9%

The following is the same table in which just macro tasks are represented.

Task id	*Name*	*Duration*	***Riccardo Mologni***			***Alberto Maria Metelli***		
			Role	*Hours*	*Units*	*Role*	*Hours*	*Units*
T1	Project planning	10	Project Manager	10	12,5%	Project Manager	10	12,5%
T2	Requirement analysis and specification	13	Analyst	33	31,7%	Analyst	33	31,7%
T3	Acceptance test plan design	7	Analyst	8	14,3%	Analyst	8	14,3%
T4	Design	20	Software Architect	30	18,8%	Software Architect	35	21,9%
T5	Unit test plan design	7	Analyst	13	23,2%	Analyst	13	23,2%
T6	Integration test plan design	11	Analyst	12	13,6%	Analyst	12	13,6%
T7	Implementation	65	Developer	260	50,0%	Developer	260	50,0%
T8	Code Ispection	13	Ispector	15	14,4%	Ispector	18	17,3%
T9	Testing	26	Tester	86	41,3%	Tester	86	41,3%
T10	Deployment	4	Installer	15	46,9%	Installer	15	46,9%

5.2 Cumulative data

In this subsection we report some cumulative data derived from the previous table. We want to make a clear distinction between real deta referred to the task we actually performed and estimated data.

5.2.1 Estimated data

- *Total duration*: 176 days (1408 h)
- *Total working hours*
  - Riccardo Mologni: 482 h
  - Alberto Maria Metelli: 490 h
- *Working hours per day*
  - Riccardo Mologni: 2,72 h/day
  - Alberto Maria Metelli: 2,78 h/day
- *Units*
  - Riccardo Mologni: 34,2%
  - Alberto Maria Metelli: 34,8%

5.2.2 Real data

- *Total duration*: 67 days (536 h)
- *Total working hours*
  - Riccardo Mologni: 100 h
  - Alberto Maria Metelli: 108 h
- *Working hours per day*
  - Riccardo Mologni: 0,57 h/day
  - Alberto Maria Metelli: 0,61 h/day
- *Units*
  - Riccardo Mologni: 18,7%
  - Alberto Maria Metelli: 20,1%

5.3 Resource allocation chart

6 Risks and recovery actions

A *risk* is a potential condition that can lead to a the loss of some value, either resource (economic or not) or time. *Risk Mitigation, Monitoring, and Management* (*RMMM*) is a key step of any project plan, it is intended to help to pre-determine any possible major risks that may occur during development of this software. In this section we will identify the main risks concerning the development of myTaxiService and we will propose some strategy to tackle them.

6.1 Introduction

In this section we provide an analysis of the main risks harming the myTaxiService project. For each risk identified we will provide the the corresponding category *project risks*, *technical risks* and *business risks,* an estimation of the probability and the impact on the project [8].

6.1.1 Project risks

Project risks (also known as development risks) are risks that might threaten the project plan, if one of them becomes real it will harm the project schedule, making it slip and increase the overall cost. In myTaxiService we identified the following project risks.

Name	*Description*	*Probability*	*Impact*
Requirement problem - 1	Requirement engineers misunderstood the the requirements.	Possible	Catastrophic
Requirement problem - 2	Customers changes the requirement in the late phases of the software development.	Possible	Catastrophic
Requirement problem - 3	The customer is not available when needed	Likely	Marginal
Personnel problem - 1	Project manager is absent at critical times in the project.	Unlikely	Critical
Personnel problem - 2	Some team members are absent at the critical times in the project.	Unlikely	Marginal
Personnel problem - 3	Development team is not qulified to program a complex application with JEE.	Likely	Critical
Personnel problem - 4	Miscommunication in the project team.	Possible	Critical
Personnel problem - 5	Lack of documentation.	Rare	Critical

6.1.2 Business risks

Business risks can threaten the viability of the software to be produced; if one of them becomes real it can compromise the economic success of the project. In myTaxiService system we identified the following business risks.

[8] Probability and impact are expressed according to the standard qualitative scale. For probability: Certain, Likely, Possible, Unlikely, Rare. For impact/effect: Negligible, Marginal, Critical, Catastrophic.

Name	*Description*	*Probability*	*Impact*
Market risk	There is no demand for product, passengers prefer to use the traditional channels to call for a taxi.	Possible	Catastrophic
Budget risk	Due to organizational financial problems the budget is reduced.	Unlikely	Catastrophic

6.1.3 Technical risks

Technical risks can threaten the quality and the timeliness of the software to be developed; if a technical risk becomes real it can make the implementation more difficult or even impossible. In myTaxiService system we identified the following technical risks.

Name	*Description*	*Probability*	*Impact*
Design problem - 1	The architectural style and patter chosen in the architectural design phase are not suitable for the kind of application to be developed.	Unlikely	Catastrophic
Design problem - 2	The algorithm for taxi management proposed in the design phase are not correct.	Rare	Serious
Design problem - 3	Design does not reflect requirements	Unlikely	Catastrophic
Implementation problem - 1	Poor comments in the code.	Possible	Marginal
Implementation problems - 2	Code does not follow quality guidelines.	Unlikely	Critical
Implementation problem - 3	The code does not reflect the designed architecture.	Unlikely	Critical
Performance problems - 1	The DBMS cannot meet the performance requirements.	Unlikely	Catastrophic

6.2 Risk strategy

A *risk strategy* defines a set of rules to be applied in order to manage and tackle risk. Those strategies are typically divided into:

- *Reactive risk strategies*: there is no explicit risk management, nothing is done about risk until something goes wrong. This is the typical approach that is adopted by the majority of software teams and managers. This strategy is also known as *crisis management* since no preventive measure is designed.

- *Proactive risk strategies*: risk identification, analysis and ranking are performed in advance in order to develop a *contingency* plan to manage risk having high probability and high impact. This approach, even if more costly, is aimed to avoid risk in order to be able to deal with only unforeseen risks.

Since in the previous section we have identified the risks, we are actually following the proactive risk strategy. We think that identifying the risks, set up proper measures in order to tackle them in an appropriate way would make smaller the economic effort in the future stages.

Figure 4: Proactive risk strategy cycle

In the following tables we report for each risk identified in the previous section a possible *prevention plan* or a *correction plan*.

6.2.1 Project risks

Name	*Prevention*	*Correction*
Requirement problem - 1	• Adopt a formalized method for requirement engineering (like Jackson-Zave approach). • Perform validation of requirements right before design phase by meeting the stakeholders. • After requirement engineering phase provide customers with a small prototype.	Recycle on the requirement engineering phase fixing the RASD.
Requirement problem - 2	• It is explained to the customer, that after he has accepted a version of the RASD, it cannot be changed by the customer's wish only. • Trace the requirements in order to quantify the effort of changing.	Recycle on the requirement engineering phase fixing the RASD.
Requirement problem - 3	• Try to stay in touch with the customer in order to increase his interest on the product. • Meetings with the customer can be planned well in advance.	When the customer is not available, meetings may have to be rescheduled.
Personnel problem - 1	Nominate a vice project manager.	Vice project manager takes over the role of the project manager.
Personnel problem - 2	• Team members should warn the project manager timely before a planned period of absence. • Ensure that knowledge is shared between team members	An other team member takes care of the work.

Name	*Prevention*	*Correction*
Personnel problem - 3	Provide an introductory course to JEE	
Personnel problem - 4	• After a meeting, one group member creates an interview report. • Team members should not hesitate to ask and re-ask questions if things are unclear.	When it becomes clear that miscommunication is causing problems, the team members involved and the customer are gathered in a meeting to clear things up.
Personnel problem - 5	Impose that before starting a new phase of the software development the corresponding document is ready.	Ask the employee to write documentation.

6.2.2 Business risks

Name	*Prevention*	*Correction*
Market risk	Make sure there is the interest around the product.	Advertise the product.
Budget risk	Make sure at the beginning of the project that there is the availability of budget.	Communicate with the founders showing how the project makes a contribution to the goals of the business and argument about the fact that cuts to the budget would not be cost effective.

6.2.3 Technical risks

Name	*Prevention*	*Correction*
Design problem - 1	• Perform inspection of the DD. • Ask some advisor on his opinion about the feasibility and the correctness of certain design decisions.	• When errors in the design are noticed consulted some advisor to help correct the design errors as soon as possible. • Also all the work, that depends on the faulty design, should be halted until the error is corrected.
Design problem - 2	Use simulation techniques to test the correctness of the algorithm	Revise the algorithm in order to fix the flaws.
Design problem - 3	• Perform inspection of the DD. • Show at least some part of the DD to the customer in order to see his/her opinion.	Recycle on the design phase fixing the DD.
Implementation problem - 1	Provide the developer with a clear specification of what to comment.	Ask the developer to add the missing comments.
Implementation problems - 2	• Provide the developer with a document in which code conventions are stated. • Perform code inspection.	Ask the developer to fix the issues.
Implementation problem - 3	• Perform unit testing after each module is ready • Continuesly check the work of the developers	Ask the developer to fix the involved component.
Performance problems - 1	Estimate the load and acquire a suitable DBMS.	Investigate the possibility of acquiring a highier performance DBMS.

A Appendix

Used tools

1. LyX visual editor for LaTeX (`http://www.lyx.org/`) to write this document.
2. GanttProject for the Gantt diagram and the resource diagram `http://www.ganttproject.biz/`.

Hours of works

Time spent by each group member:

- Alberto Maria Metelli: 10 h
- Riccardo Mologni: 10 h

Revision history

Version	*Date*	*Revision description*	*Revision notes*
0.1	30-1-2016	Initial draft	-
1.0	2-2-2016	Final draft	-
2.0	22-2-2016	Final release	Fixed introduction and some terminology.

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

Software Engineering 2 - Project

CI
Code Inspection

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 term CI (*Code Inspection*) refers to peer review of any work product, involving code, by trained individuals who look for mistakes (also known as defects or issues) using a well defined process. A mistake is an incorrect feature of the code that can result in the introduction of faults into a system which in turn can become errors and failures. Therefore the main purpose of code inspection is to identify defects and, if necessary, improve the quality of the code. Code inspection is a specific case of *static analysis* as a part of the V&V (*Verification* and *Validation*) process.

Code revision can be classified into *walkthoughts* and *inspection*, those are general approaches to revision of products that can be exploited in any stage of the software lifecycle. Even if they share the same goal the revision process is quite different:

- *walkthroughts* are typically informal reviews that involve experts of domain which are in charge of verifying the correctness of the product with respect to their viewpoint. The producer, in case of code walkthrought the developer, presents the code and the attached documentation, if any, and the reviewers discuss the correctness of the product.
- *inspections* are based on formal evaluation techniques proposed by Fagan in which code is exterminated by a group of professional inspectors to check its correctness, typically on the bases of a set of quality standards defined as *checklist*. Contrary to what happens in walkthroughts, here people taking part to the inspection have well-defined roles and the process is composed of several steps: the moderator chooses participants (readers, testers and inspectors) and schedules the meeting that takes place analysing code line-by-line with the support of the explenation of the author.

This document is intended to be a track for the inspection process of an extract of the Glassfish 4.1 application server code we performed. This document is mainly addressed to developers in order to fix possible mistakes highlighted during the inspection process and for possible further inspection sessions.

1.2 References

[1] Software Engineering 2 course slides.

[2] Glassfish reference `http://glassfish.pompel.me/`.

[3] Oracle documentation `https://docs.oracle.com/cd/E18930_01/html/821-2416/ggjue.html#abllb`.

1.3 Code inspection checklist

In this subsection we present the checklist followed to perform code inspection of an extract of the Glassfish 4.1 application server code.

1.3.1 Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.

3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite.

4. Interface names should be capitalized like classes. 5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().

5. Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter.

6. All the remaining words in the variable name have their first letter capitalized. Examples: _windowHeight, timeSeriesData.

7. Constants are declared using all uppercase with words separated by an underscore. Examples: MIN_WIDTH; MAX_HEIGHT.

1.3.2 Indentation

8. Three or four spaces are used for indentation and done so consistently.

9. No tabs are used to indent.

1.3.3 Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).

11. All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces. Example, avoid this:

```
if ( condition )
        doThis();
```

Instad do this:

```
if ( condition )
{
        doThis();
}
```

1.3.4 File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

13. Where practical, line length does not exceed 80 characters.

14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

1.3.5 Wrapping Lines

15. Line break occurs after a comma or an operator.

16. Higher-level breaks are used.

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

1.3.6 Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

1.3.7 Java Source Files

20. Each Java source file contains a single public class or interface.

21. The public class is the first class or interface in the file.

22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

1.3.8 Package and Import Statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

1.3.9 Class and Interface Declarations

25. The class or interface declarations shall be in the following order:

    1. A. class/interface documentation comment
    2. class or interface statement
    3. class/interface implementation comment, if necessary
    4. class (static) variables
        (a) first public class variables
        (b) next protected class variables
        (c) next package level (no access modifier)
        (d) last private class variables
    5. instance variables
        (a) first public instance variables
        (b) next protected instance variables
        (c) next package level (no access modifier)
        (d) last private instance variables
    6. constructors
    7. methods

26. Methods are grouped by functionality rather than by scope or accessibility.

27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

1.3.10 Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected)

29. heck that variables are declared in the proper scope

30. Check that constructors are called when a new object is desired

31. Check that all object references are initialized before use

32. Variables are initialized where they are declared, unless dependent upon a computation

33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces "{" and "}"). The exception is a variable can be declared in a 'for' loop.

1.3.11 Method Calls

34. Check that parameters are presented in the correct order

35. Check that the correct method is being called, or should it be a different method with a similar name

36. Check that method returned values are used properly

1.3.12 Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index)

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds

39. Check that constructors are called when a new array item is desired

1.3.13 Object Comparison

40. Check that all objects (including Strings) are compared with "equals" and not with "=="

1.3.14 Output Format

41. Check that displayed output is free of spelling and grammatical errors.

42. Check that error messages are comprehensive and provide guidance as to how to correct the problem

43. Check that the output is formatted correctly in terms of line stepping and spacing

1.3.15 Computation, Comparisons and Assignments

44. Check that the implementation avoids "brutish programming: (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`)

45. Check order of computation/evaluation, operator precedence and parenthesizing

46. Check the liberal use of parenthesis is used to avoid operator precedence problems.

47. Check that all denominators of a division are prevented from being zero

48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding

49. Check that the comparison and Boolean operators are correct

50. Check throw-catch expressions, and check that the error condition is actually legitimate

51. Check that the code is free of any implicit type conversions

1.3.16 Exceptions

52. Check that the relevant exceptions are caught

53. Check that the appropriate action are taken for each catch block

1.3.17 Flow of Control

54. In a switch statement, check that all cases are addressed by break or return

55. Check that all switch statements have a default branch

56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions

1.3.18 Files

57. Check that all files are properly declared and opened

58. Check that all files are closed properly, even in the case of an error

59. Check that EOF conditions are detected and handled correctly

60. Check that all file exceptions are caught and dealt with accordingly

1.4 Our code inspection process

Fagan inspection, as already mentioned, being a formal analysis technique, has a specific process and roles associated to the participants; considered the academic nature of this document we think that such an approach should be inappropriate, therefore beside the manual inspection either based on the given checklist or exploiting other considerations about the code, we leveraged on automatic tools in order to discover further defects. The following list, shows the techniques and tools adopted.

- *Manual inspection*: revision of the code line by line performed by the group members together in order to discover defects according to:

  - the *checklist* proposed in the assignment,
  - other considerations based on the experience of the group members.

- *Automatic code review*: revision of the code performed by static code analyzers, in particular the following has been used[1]:
  - *SonarQube*: an open platform to manage code quality (duplications, complexity, potential bugs, coding rules, comments, architecture and design) `http://www.sonarqube.org/`,
  - *PMD*: a Java source code analyzer aimed to finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation `https://pmd.github.io/`,
  - *FindBugs*: a program which uses static analysis to look for bugs in Java code `http://findbugs.sourceforge.net/`.

1.5 Document Structure

This document is composed of five sections and an appendix.

- The first section, this one, provides and overall description of the review processes focusing on the specific assignment consisting in the inspection of an extract of the Glassfish 4.1 application server code. It also presents the checklist used in the following sections.
- The second section is devoted to the description of the extract of code. Classes and method names will be stated with their location in the source code.
- The third section provides the illustration of the functional role of those classes and methods both in informal language with respect to the JEE architecture and in semiformal way by means of UML class diagram to highlight dependencies between classes. We will focus on the reverse engineering approach we adopted.
- The fourth section is devoted to the inspection. For each category of defects and for each defect description, issues identified will be stated with reference to the line of the code involved, the motivation and a possible solution when appropriate.
- The fifth section describes other possible problems identified during the inspection that do not conform to the points presented in the checklist.
- 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.

[1] They are all very similar but differ in the importance given to different aspects of code quality.

2 Assigned classes

2.1 Classes

2.1.1 ComponentEnvManagerImpl

- Signature: `public class ComponentEnvManagerImpl implements ComponentEnvManager`
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2 Methods

2.2.1 getJndiNameEnvironment

- Signature: `public JndiNameEnvironment getJndiNameEnvironment(String componentId)`
- Start line: 160
- End line: 168
- SLOC: 9
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2.2 bindToComponentNamespace

- Signature: `public String bindToComponentNamespace(JndiNameEnvironment env)throws NamingException`
- Start line: 188
- End line: 272
- SLOC: 85
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2.3 addAllDescriptorBindings

- Signature: `private void addAllDescriptorBindings addAllDescriptorBindings(JndiNameEnvironment env , ScopeType scope , Collection < JNDIBinding > jndiBindings)`
- Start Line: 312
- End line: 366
- SLOC: 55
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2.4 unbindFromComponentNamespace

- Signature: `public void unbindFromComponentNamespace(JndiNameEnvironment env)throws NamingException`
- Start line: 373
- End line: 418
- SLOC: 46
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

3 Functional roles of classes

This section it devoted to the description of the overall functionalists of the class focusing in particular on the role of the methods. Understanding the role of a module or some methods of that module is a non trivial *reverse engineering* task, even more complex when the reference and the documentation is entirely lacking. Therefore we tried, as best as we could, to exploit the code, the few comments present and a variety of online resources to extract at least the general characteristics. The description we provide below is a very high level explanation intended to show the conceptual functionalists of the code assigned without going into details that actually we were not be able to understand because of the complexity and the bad documentation of this part of the project.

The first subsection gives a general description of what JNDI and JNDI environment are in the context of Glassfish, this is necessary to understand the role of the methods; if the reader already knows how they work, subsection 3.1 can be skipped.

3.1 About JNDI (from Oracle doc)

Mainly taken from [3].

By making calls to the JNDI API, applications locate resources and other program objects. A resource is a program object that provides connections to systems, such as database servers and messaging systems. Each resource object is identified by a unique, people-friendly name, called the JNDI name. A resource object and its JNDI name are bound together by the naming and directory service, which is included with the GlassFish Server. When a new name-object binding is entered into the JNDI, a new resource is created.

3.1.1 Java EE Naming Environment

JNDI names are bound to their objects by the naming and directory service that is provided by a Java EE server. Because Java EE components access this service through the JNDI API, the object usually uses its JNDI name.

Java EE application clients, enterprise beans, and web components must have access to a JNDI naming environment.

The *application component's naming environment* is the mechanism that allows customization of the application component's business logic during deployment or assembly. This environment allows you to customize the application component without needing to access or change the source code off the component. A Java EE container implements the provides the environment to the application component instance as a JNDI naming context.

3.1.2 How the Naming Environment and the Container Work Together

The application component's environment is used as follows:

- The application component's business methods access the environment using the JNDI interfaces. In the deployment descriptor, the application component provider declares all the environment entries that the application component expects to be provided in its environment at runtime.
- The container provides an implementation of the JNDI naming context that stores the application component environment. The container also provides the tools that allow the deployer to create and manage the environment of each application component.

- A deployer uses the tools provided by the container to initialize the environment entries that are declared in the application component's deployment descriptor. The deployer sets and modifies the values of the environment entries.
- The container makes the JNDI context available to the application component instances at runtime. These instances use the JNDI interfaces to obtain the values of the environment entries.

Each application component defines its own set of environment entries. All instances of an application component within the same container share the same environment entries. Application component instances are not allowed to modify the environment at runtime.

3.2 Diagrams

A class diagram showing the main methods and dependencies of the assigned class is now provided.

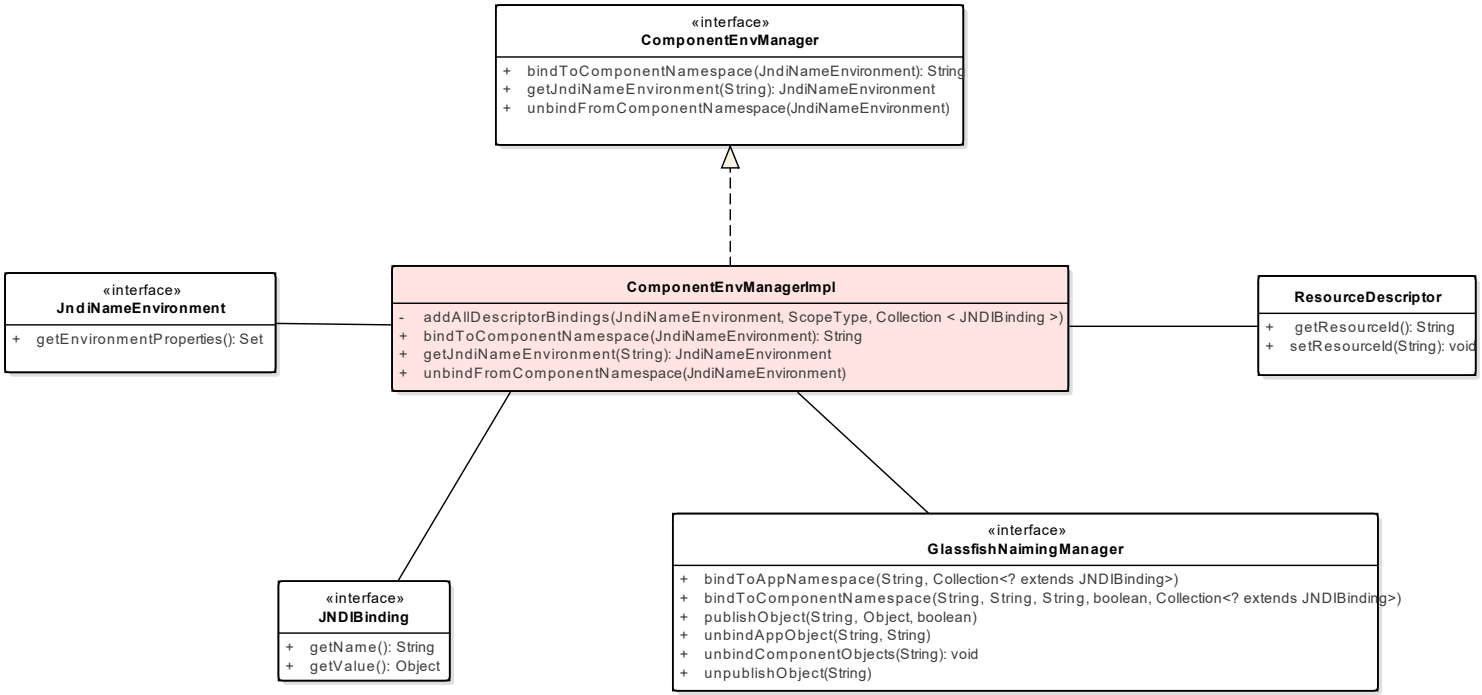


Figure 1: UML Class Diagram of the main methods and dependencies

3.3 Classes

3.3.1 ComponentEnvManagerImpl

The class ComponentEnvManagerImpl implements the interface *ComponentEnvManager* that defines its public interface and is in charge of all the main operations concerning the management of the application component's naming environment. In particular it provides methods to bind/unbind a JNDI environment of a component or an application to the JNDI service interacting with the naming manager (these methods will be discussed further later), keeping a local track of those bindings by means of the register/unregister methods. It also provides interface methods to get the JNDI environment associated to one component, the current JNDI environment, the current Application environment and the string name of a JNDI environment. Eventually it allows to modify properties of already bound JNDI environments.

3.4 Methods

3.4.1 getJndiNameEnvironment

The *getJndiNameEnvironment* method returns the JNDI environment associated to the component whose string name is passed as a parameter, if the name of the component is not bound in any JNDI environment the method returns null.

3.4.2 bindToComponentNamespace

The *bindToComponentNamespace* method is in charge of publishing in the JNDI server the name of the JNDI environment passed as parameter. It collects all JNDI bindings associated to the JNDI environment for both component, module and application scope; then according to the type of the JNDI environment (application or application client descriptor) uses the methods provided by the naming manager to bind it and rebind all the JNDI binding collected before within the context of the JNDI environment. Eventually it registers locally the binding. It propagates the possible NamingException generated by the naming manager.

3.4.3 addAllDescriptorBindings

The *addAllDescriptorBindings* method is in charge of converting the resource descriptors associated to the JNDI environment within a specific scope (application, component or module) to JNDI bindings. It receives as parameters the JNDI environment, the scope and the collection of JNDI bindings and augments the latter with the newly generated JNDI bindings.

3.4.4 unbindFromComponentNamespace

The *unbindFromComponentNamespace* method plays the opposite role with respect to *bindToComponentNamespace*. It receives as parameter a JNDI environment and it undeploys all resources descriptors associated; then uses the naming manager to unpublish all the JNDI bindings associated to the JNDI environment. Eventually it unregisters locally the binding. It propagates the possible NamingException generated by the naming manager.

4 Issues

4.1 Issue list

In the following we present the list of all issues found during the code inspection divided into the categories defined in the checklist. The field #Issue is used to number progressively the issues, few numbers may be missing because of successive revisions.

4.1.1 Naiming conventions

#Issue	**1**
Class/Method	ComponentEnvManagerImpl/getJndiNameEnvironment
#Line	161
Code fragment	`RefCountJndiNameEnvironment rj`
Issue category	Naiming conventions
Issue ref	1
Motivation	Method variable name rj non meaningful.
Comment	Variable name should refer to the object referenced.

#Issue	**2**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	236
Code fragment	`JndiNameEnvironment next`
Issue category	Naiming conventions
Issue ref	1
Motivation	Method variable name next non corresponding to meaning.
Comment	Variable name should refer to the object referenced.

#Issue	**3**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	241, 253
Code fragment	`JNDIBinding next`
Issue category	Naiming conventions
Issue ref	1
Motivation	Method variable name next non corresponding to meaning.
Comment	Variable name should refer to the object referenced.

#Issue	**4**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	383, 399
Code fragment	`JNDIBinding next`
Issue category	Naiming conventions
Issue ref	1
Motivation	Method variable name next non corresponding to meaning.
Comment	Variable name should refer to the object referenced.

4.1.2 Indentation

No issues found.

4.1.3 Braces

#Issue	5
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	265
Code fragment	`if (_logger.isLoggable(Level.FINEST))`
Issue category	Braces
Issue ref	11
Motivation	Avoid using if statements without curly braces.
Comment	

4.1.4 File Organization

#Issue	6
Class/Method	ComponentEnvManagerImpl
#Line	71, 85
Code fragment	`blank line`
Issue category	File organization
Issue ref	12
Motivation	Blank line between groups of imports.
Comment	Acceptable because separates different types of imports.

#Issue	7
Class/Method	ComponentEnvManagerImpl
#Line	186, 273, 371, 767, 780, 781, 800, 801, 802, 803, 804, 805
Code fragment	`double blank line`
Issue category	File organization
Issue ref	12
Motivation	Double blank line between methods.
Comment	Just one line is preferrable.

#Issue	8
Class/Method	ComponentEnvManagerImpl
#Line	1056
Code fragment	`blank line`
Issue category	File organization
Issue ref	12
Motivation	Blank line between closing curly brackets.
Comment	No blank line between curly brackets is preferred.

#Issue	9
Class/Method	ComponentEnvManagerImpl
#Line	1058
Code fragment	`blank line`
Issue category	File organization
Issue ref	12
Motivation	Blank line at the end of the file.
Comment	No blank line at the end of the file is preferred.

#Issue	**10**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	262
Code fragment	`double blank line`
Issue category	File organization
Issue ref	12
Motivation	Double blank line within a method.
Comment	Just one line to seprarate different sections within a method is preferred.

#Issue	**11**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	319
Code fragment	`no blank line`
Issue category	File organization
Issue ref	12
Motivation	Missing blank line between declaration and code.
Comment	One blank line to separate declarations and code is preferred.

#Issue	**12**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	417
Code fragment	`blank line`
Issue category	File organization
Issue ref	12
Motivation	Blank line between closing curly brackets.
Comment	No blank line between curly brackets is preferred.

#Issue	**13**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	405
Code fragment	`blank line`
Issue category	File organization
Issue ref	12
Motivation	Blank line before closing curly bracket
Comment	No blank line between curly brackets is preferred.

#Issue	**14**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	213, 216, 219, 242, 256, 258
Code fragment	
Issue category	File organization
Issue ref	13
Motivation	Line exceeds 80 charachters.
Comment	In those case line wrap should be possible.[2]

#Issue	**15**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	390, 402
Code fragment	
Issue category	File organization
Issue ref	13
Motivation	Line exceeds 80 charachters
Comment	In those case line wrap should be possible.

#Issue	**16**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	290
Code fragment	
Issue category	File organization
Issue ref	13
Motivation	Line exceeds 80 charachters
Comment	In those case line wrap should be possible.

#Issue	**17**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	312, 362
Code fragment	
Issue category	File organization
Issue ref	14
Motivation	Line exceeds 120 charachters
Comment	In those cases line wrap should be possible.

4.1.5 Wrapping Lines

No issues found.

4.1.6 Comments

#Issue	**18**
Class/Method	ComponentEnvManagerImpl
#Line	89
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Class behaviour explanation missing.
Comment	Every class should be provided with a comment explaining its main functionalities.

#Issue	**19**
Class/Method	ComponentEnvManagerImpl/getJndiNameEnvironment
#Line	160
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Method behaviour explanation missing, the corresponding comment in the interface declaration is inconsistent.
Comment	Every public method should be provided with a comment explaining its main functionalities, preferably in the interface declaration.

#Issue	**21**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	188
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Method behaviour explanation missing
Comment	Every public method should be provided with a comment explaining its main functionalities, preferably in the interface declaration.

#Issue	**22**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	200, 204, 225, 251
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Block behaviour explanation missing
Comment	Every significant block should be provided with a comment explaining its main functionalities.

#Issue	**23**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	312
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Method behaviour explanation missing
Comment	Every method should be provided with a comment explaining its main functionalities.

#Issue	**24**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	319, 336
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Block behaviour explanation missing
Comment	Every significant block should be provided with a comment explaining its main functionalities.

#Issue	**25**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	373
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Method behaviour explanation missing
Comment	Every public method should be provided with a comment explaining its main functionalities, preferably in the interface declaration.

#Issue	**26**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	395, 409
Code fragment	
Issue category	Comments
Issue ref	18
Motivation	Block behaviour explanation missing
Comment	Every significant block should be provided with a comment explaining its main functionalities.

4.1.7 Java Source Files

#Issue	**27**
Class/Method	ComponentEnvManagerImpl
#Line	88
Code fragment	
Issue category	Java Source Files
Issue ref	23
Motivation	Incomplete javadoc of the class.
Comment	Every class should be documented explaining its role in the whole project.

#Issue	**28**
Class/Method	ComponentEnvManagerImpl/getJndiNameEnvironment
#Line	160
Code fragment	
Issue category	Java Source Files
Issue ref	23
Motivation	Incomplete javadoc for the method.
Comment	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

#Issue	**29**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	188
Code fragment	
Issue category	Java Source Files
Issue ref	23
Motivation	Incomplete javadoc for the method.
Comment	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

#Issue	**30**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	380
Code fragment	
Issue category	Java Source Files
Issue ref	23
Motivation	Incomplete javadoc for the method.
Comment	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

#Issue	**31**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	312
Code fragment	
Issue category	Java Source Files
Issue ref	23
Motivation	Javadoc missing for the method.
Comment	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

4.1.8 Package and Import Statements

No issues found.

4.1.9 Class and interface declaration

#Issue	**34**
Class/Method	ComponentEnvManagerImpl
#Line	100
Code fragment	`private ServiceLocator locator`
Issue category	Class and interface declaration
Issue ref	25, d, d
Motivation	Private instance variable before package level variable.
Comment	Private instance variables should appear as last.

#Issue	**35**
Class/Method	ComponentEnvManagerImpl
#Line	103
Code fragment	`private Logger _logger`
Issue category	Class and interface declaration
Issue ref	25, d, d
Motivation	Private instance variable before package level variable.
Comment	Private instance variables should appear as last.

#Issue	**36**
Class/Method	ComponentEnvManagerImpl
#Line	188
Code fragment	
Issue category	Class and interface declaration
Issue ref	26
Motivation	Methods are not presented in the order in which they are declared in the interface.
Comment	Methods implementation should appear in the same order in which they are defined in the corresponding interface.

#Issue	**37**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	188
Code fragment	
Issue category	Class and interface declaration
Issue ref	27
Motivation	Method is too long, the Cyclomatic Complexity of this method is 14.
Comment	Method should not be too much long, maximum Cyclomatic Complexity is 10.

#Issue	**38**
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	221
Code fragment	`compEnvId = getComponentEnvId(env)`
Issue category	Class and interface declaration
Issue ref	27
Motivation	Duplicated code at line 191.
Comment	Since variable compEnvId is not modified in between, this declaration is useless.

4.1.10 Initialization and declarations

#Issue	39
Class/Method	ComponentEnvManagerImpl
#Line	106
Code fragment	`GlassfishNamingManager namingManager`
Issue category	Initialization and Declarations
Issue ref	28
Motivation	Friendly instance variable not used by same package classes, should be private.
Comment	Keeping the lowest visibility as possible is preferred for information hiding.

#Issue	40
Class/Method	ComponentEnvManagerImpl
#Line	109
Code fragment	`ComponentNamingUtil componentNamingUtil`
Issue category	Initialization and Declarations
Issue ref	28
Motivation	Friendly instance variable not used by same package classes, should be private.
Comment	Keeping the lowest visibility as possible is preferred for information hiding.

#Issue	41
Class/Method	ComponentEnvManagerImpl
#Line	112
Code fragment	`transient private CallFlowAgent callFlowAgent`
Issue category	Initialization and Declarations
Issue ref	29
Motivation	Instance variable used only at line 820, could be declared as static field of the inner class FactoryForEntityManagerWrapper.
Comment	Keeping the smallest scope as possible is preferred. for information hiding.

#Issue	42
Class/Method	ComponentEnvManagerImpl
#Line	115
Code fragment	`transient private TransactionManager txManager`
Issue category	Initialization and Declarations
Issue ref	29
Motivation	Instance variable used only at line 820, could be declared as static field of the inner class FactoryForEntityManagerWrapper
Comment	Keeping the smallest scope as possible is preferred. for information hiding.

#Issue	43
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	250
Code fragment	`Application app = DOLUtils.getApplicationFromEnv(env)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks.
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	**44**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	348
Code fragment	`String resourceId = getResourceId(env, descriptor)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks.
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	**45**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	351
Code fragment	`CommonResourceProxy proxy = locator.getService(CommonResourceProxy.class)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks.
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	**46**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	354
Code fragment	`String logicalJndiName = descriptorToLogicalJndiName(descriptor)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks.
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	**47**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	355
Code fragment	`CompEnvBinding envBinding = new CompEnvBinding(logicalJndiName, proxy)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	**48**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	362
Code fragment	`CompEnvBinding jmscfEnvBinding = new CompEnvBinding(ConnectorsUtil.getPMJndiName(logicalJndiName), jmscfProxy)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	49
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	380
Code fragment	`Collection<JNDIBinding> globalBindings = new ArrayList<JNDIBinding>()`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	50
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	387
Code fragment	`Application app = DOLUtils.getApplicationFromEnv(env)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks
Comment	Variable declaration at the beginning of a block is preferred for readability.

#Issue	51
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	390
Code fragment	`Set<ApplicationClientDescriptor> appClientDescs = app.getBundleDescriptors(ApplicationClientDescriptor.class)`
Issue category	Initialization and Declarations
Issue ref	33
Motivation	Declarations do not appear at the beginning of blocks
Comment	Variable declaration at the beginning of a block is preferred for readability.

4.1.11 Method calls

No issues found.

4.1.12 Arrays

No issues found.

4.1.13 Object comparision

No issues found.

4.1.14 Output format

No issues found.

4.1.15 Computation, Comparitions and Assigments

#Issue	**52**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	342
Code fragment	`if(descriptor.getResourceType().equals(DSD)){if (((DataSourceDefinitionDescriptor)descriptor).isDeployed())`
Issue category	Computation, Comparitions and Assigments
Issue ref	45
Motivation	Enclosed if should be merged.
Comment	If not needed netsed if should be merged in one if with the congiunction of the conditions.

#Issue	**53**
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	395
Code fragment	`if( !(env instanceof ApplicationClientDescriptor )&& (app.getBundleDescriptors(ApplicationClientDescriptor.class).size()> 0))`
Issue category	Computation, Comparitions and Assigments
Issue ref	46
Motivation	Useless parenthesis around second operand.
Comment	Non necessary parenthesis are acceptable only to highlight precedence.

4.1.16 Exceptions

No issues found.

4.1.17 Flow of control

No issues found.

4.1.18 Files

No issues found.

4.2 Cumulative data

In this subsection we present some aggregate data derived from the process of code inspection that could be useful to understand the distribution of issues in category and in methods. We will use both tables and diagrams.

4.2.1 Issues per category and class/method - table

Class/Method - Issue category	SLOC	Naiming conventions	Indentation	Braces	File organization	Wrapping lines	Comments	Java Source Files	Package and import statements	Class and interface declaration	Initialization and declarations	Method calls	Arrays	Object comparision	Output format	Computation, Comparitions and Assigments	Exceptions	Flow of control	Files	*Total for class/method*	*Total per class/method per KSLOC*
ComponentEnvManagerImpl	121	0	0	0	15	0	1	0	0	2	2	0	0	0	0	0	0	0	0	20	165,3
ComponentEnvManagerImpl/ getJndiNameEnvironment	9	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	2	222,2
ComponentEnvManagerImpl/ addAllDescriptorBindings	55	0	0	0	4	0	3	1	0	0	1	0	0	0	0	1	0	0	0	10	181,8
ComponentEnvManagerImpl/ unbindFromComponentNames-pace	46	2	0	0	4	0	3	0	0	0	3	0	0	0	0	1	0	0	0	13	282,6
ComponentEnvManagerImpl/ bindToComponentNamespace	85	3	0	1	7	0	5	0	0	2	1	0	0	0	0	0	0	0	0	19	223,5
Total per category	316	6	0	1	30	0	13	1	0	4	7	0	0	0	0	2	0	0	0	64	215,1
Total per category (%)		9,38	0	1,6	46,9	0	20,3	1,6	0	6,3	10,9	0	0	0	0	3,1	0	0	0	100	

4.2.2 Issues per category - hystogram

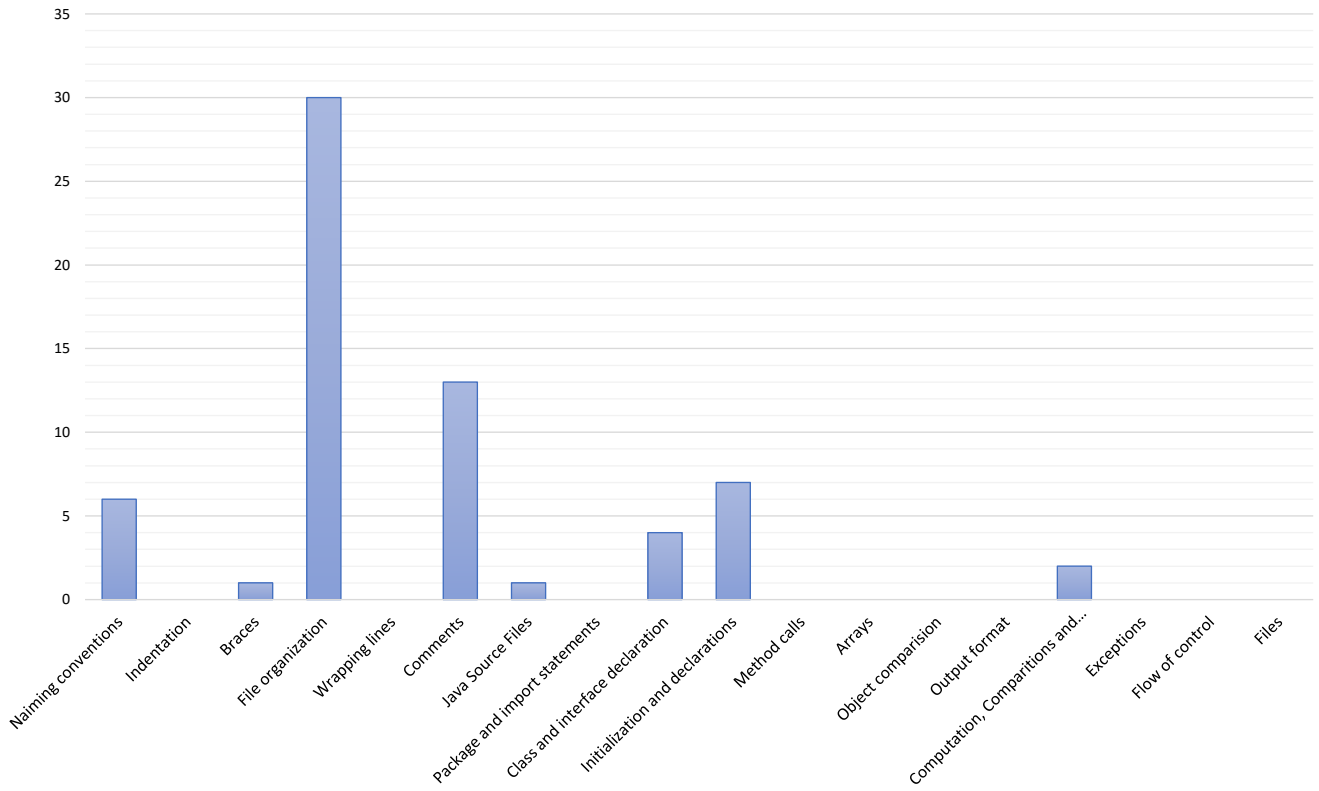


Figure 2: Hystogram showing the distribution of issues in categories.

4.2.3 Issues per class/method per KSLOC - hystogram

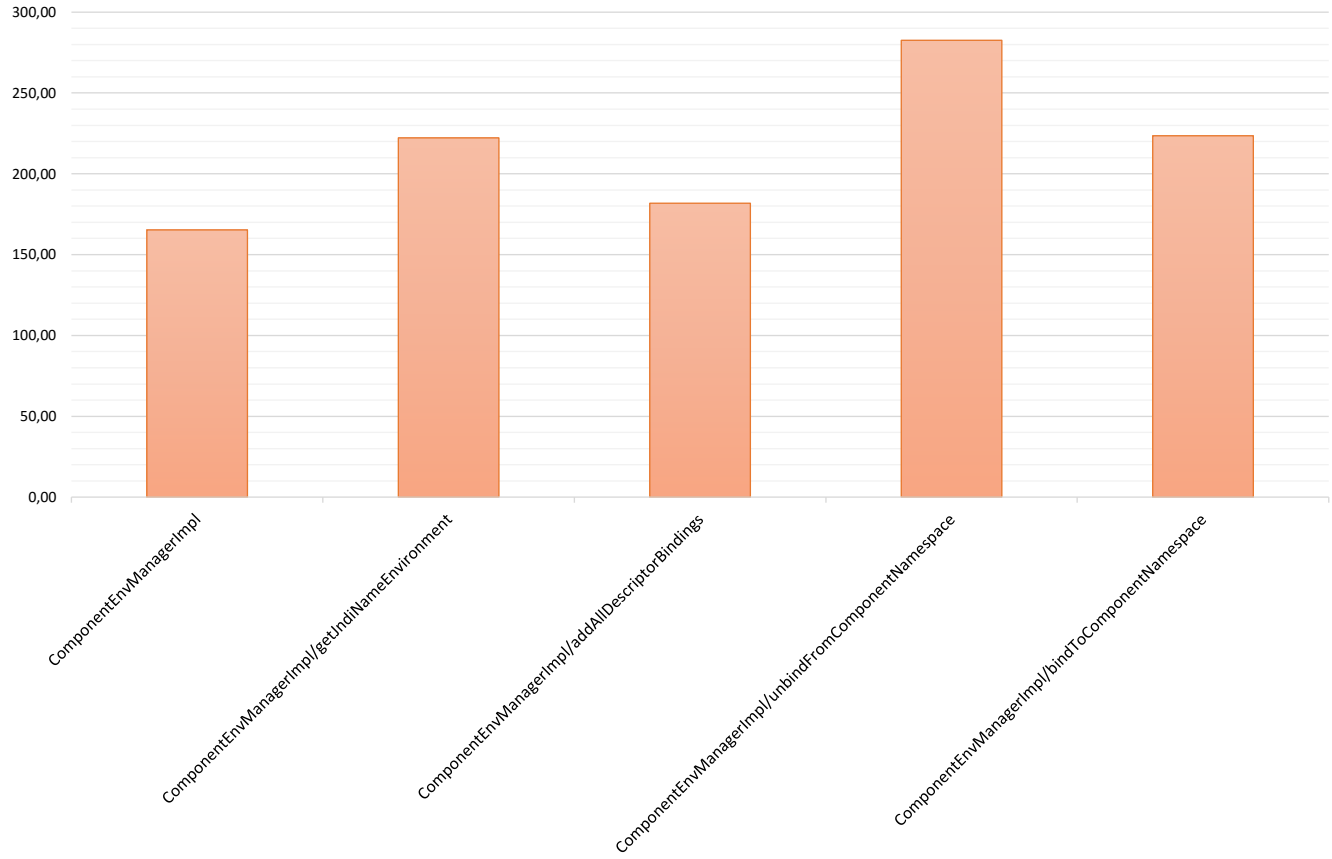


Figure 3: Hystogram showing the distribution of issues in method/class.

4.2.4 Issues per class/method per KSLOC - pie chart

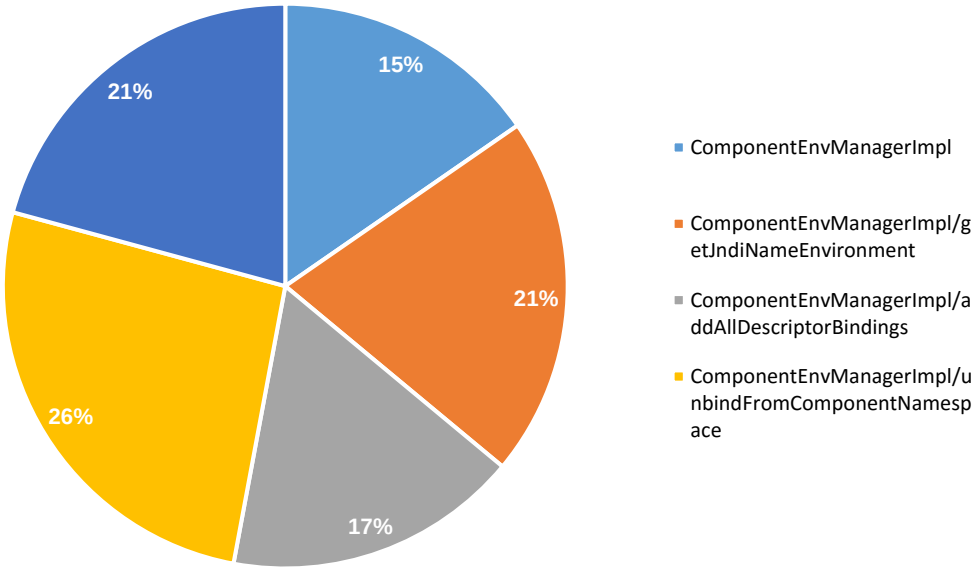


Figure 4: Pie chart showing the distribution of issues in method/class.

5 Other problems

In this section we expose some issues, not contained in the checklist, higlighted by other automatic tools for code revision or identified during the manual revision that, in our opinion, can lead to bugs.

5.1 Automatic code review

We don't present all issues reported by the tools, since many of them has already been identified during the manual inspection driven by the checklist and some of them we think are not appropriate (for instance, PMD reports all variables local to methods to be declared final if not modifies, we think this is exaggerated while this policy should be adopted for instance variable of classes.

#Issue	**1**
Class/Method	ComponentEnvManagerImpl
#Line	61
Code fragment	`import org.glassfish.hk2.api.ActiveDescriptor`
Issuer	FindBugs
Message	Unused import.
Comment	

#Issue	**2**
Class/Method	ComponentEnvManagerImpl
#Line	63
Code fragment	`import org.glassfish.hk2.utilities.BuilderHelper`
Issuer	FindBugs
Message	Unused import.
Comment	

#Issue	**3**
Class/Method	ComponentEnvManagerImpl
#Line	103
Code fragment	`private Logger _logger`
Issuer	FindBugs, PMD
Message	The logger declaration field _logger should be static and final.
Comment	

#Issue	**4**
Class/Method	ComponentEnvManagerImpl
#Line	128
Code fragment	`private ConcurrentMap<String, RefCountJndiNameEnvironment> compId2Env = new ConcurrentHashMap<String, RefCountJndiNameEnvironment>()`
Issuer	FindBugs
Message	Field compId2Env should be final.
Comment	Instance variable non modified after instansiation should be declared final.

#Issue	5
Class/Method	ComponentEnvManagerImpl
#Line	129
Code fragment	`private ConcurrentMap<String, RefCountJndiNameEnvironment> compId2Env = new ConcurrentHashMap<String, RefCountJndiNameEnvironment>()`
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad).
Comment	

#Issue	6
Class/Method	ComponentEnvManagerImpl/getJndiNameEnvironment
#Line	159
Code fragment	`public JndiNameEnvironment getJndiNameEnvironment(String componentId)`
Issuer	FindBugs
Message	Add @Override annotation.
Comment	All methods inherited by superclasses or interfaces shoud be marked with @Override annotation.

#Issue	7
Class/Method	ComponentEnvManagerImpl/getJndiNameEnvironment
#Line	163
Code fragment	`_logger.finest("ComponentEnvManagerImpl: "+ "getCurrentJndiNameEnvironment "+ inv.componentId + "is "+ desc.getClass())`
Issuer	FindBugs
Message	Inefficient use of string concatenation in logger.
Comment	A unique string should be used instad of concatenation.

#Issue	8
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	188
Code fragment	`public String bindToComponentNamespace(JndiNameEnvironment env)`
Issuer	FindBugs
Message	Add @Override annotation
Comment	All methods inherited by superclasses or interfaces shoud be marked with @Override annotation.

#Issue	9
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	193
Code fragment	`Collection<JNDIBinding> bindings = new ArrayList<JNDIBinding>()`
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad).
Comment	

#Issue	10
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	228
Code fragment	`Collection<JNDIBinding> globalBindings = new ArrayList<JNDIBinding>()`
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad).
Comment	

#Issue	11
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	266
Code fragment	`_logger.finest("ComponentEnvManagerImpl: "+ "register "+ compEnvId + "is "+ env.getClass())`
Issuer	FindBugs
Message	Inefficient use of string concatenation in logger
Comment	A unique string should be used instad of concatenation.

#Issue	12
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	314
Code fragment	`Set<ResourceDescriptor> allDescriptors = new HashSet<ResourceDescriptor>()`
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad)
Comment	

#Issue	13
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	343
Code fragment	`((DataSourceDefinitionDescriptor)descriptor)`
Issuer	FindBugs
Message	Unchecked/unconfirmed cast of return value from method.
Comment	Casting an instance of superclass to an instance of subclass, not necessarely wrong but maybe deserved an explicative comment.

#Issue	14
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	373
Code fragment	`public void unbindFromComponentNamespace(JndiNameEnvironment env)`
Issuer	FindBugs
Message	Add @Override annotation
Comment	All methods inherited by superclasses or interfaces shoud be marked with @Override annotation.

#Issue	15
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	380
Code fragment	`Collection<JNDIBinding> globalBindings = new ArrayList<JNDIBinding>()`
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad)
Comment	

#Issue	16
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	397
Code fragment	`Collection<JNDIBinding> appBindings = new ArrayList<JNDIBinding>()`
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad)
Comment	

#Issue	17
Class/Method	ComponentEnvManagerImpl
#Line	112
Code fragment	`transient private CallFlowAgent callFlowAgent`
Issuer	SonarQube
Message	Reorder the modifiers to comply with the Java Language Specification.
Comment	private should precede transient.

#Issue	18
Class/Method	ComponentEnvManagerImpl
#Line	115
Code fragment	`transient private TransactionManager txManager`
Issuer	SonarQube
Message	Reorder the modifiers to comply with the Java Language Specification.
Comment	private should precede transient.

#Issue	19
Class/Method	ComponentEnvManagerImpl
#Line	120
Code fragment	`// TODO: container-common shouldn't depend on EJB stuff, right?`
Issuer	SonarQube
Message	Complete the task associated to this TODO comment.
Comment	

#Issue	**20**
Class/Method	ComponentEnvManagerImpl
#Line	89
Code fragment	
Issuer	PMD
Message	The class 'ComponentEnvManagerImpl' has a Cydomatic Complexity of 6 (Highest = 16). Rule: CydomaticComplexity Rule set Code Size
Comment	

#Issue	**21**
Class/Method	ComponentEnvManagerImpl
#Line	90
Code fragment	
Issuer	PMD
Message	This class has too many methods, consider refactoring it.
Comment	

#Issue	**22**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	319
Code fragment	`if(!(env instanceof ApplicationClientDescriptor)){ [...] } else { [...] }`
Issuer	PMD
Message	Avoid if (x != y) .. else .. ConfusingTernary
Comment	It is preferrable to use if(x==y) .. else ..

#Issue	**23**
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	325
Code fragment	`if(!(env instanceof ApplicationClientDescriptor)){ [...] } else { [...] }`
Issuer	PMD
Message	Avoid if (x != y) .. else .. ConfusingTernary
Comment	It is preferrable to use if(x==y) .. else ..

5.2 Other issues from manual inspection

1. The usage of continue statements should be avoided since it modifies the flow of execution in a brutal way, making maintainance harder since the code following the continue statement seems to be non conditioned.

```
for (ResourceDescriptor descriptor : allDescriptors) {

        if (!dependencyAppliesToScope(descriptor, scope)) {
                continue;
        }

        if(descriptor.getResourceType().equals(DSD)) {
                if
                    (((DataSourceDefinitionDescriptor)descriptor).isDeployed())
                    {
```

```
                continue;
                }
        }

        [...]
}
```

The code above should be rephrased as follows

```
for (ResourceDescriptor descriptor : allDescriptors) {

        if (dependencyAppliesToScope(descriptor, scope) &&
                !(descriptor.getResourceType().equals(DSD) &&
                ((DataSourceDefinitionDescriptor)descriptor).isDeployed())
                   ) {

                [...]
        }
}
```

2. This inner class, even if it does not belong to the fragment of code we were assigned, is good example of "conscientious" usage of public variables. The class RefCountJndiNameEnvironment is a private inner class of ComponentEnvManagerImpl which is used exactly as a struct in C language. Its instance variable are public, but since the class is a private inner class its visibility is restricted to the outer class, therefore having those public variable does not breack encapsulation. We must report, however, that since the variable env is never modified after creation it should be declared final.

```
private static class RefCountJndiNameEnvironment {
        public RefCountJndiNameEnvironment(JndiNameEnvironment
           env) {
                this.env = env;
                this.refcnt = new AtomicInteger(1);
        }
        public JndiNameEnvironment env;
        public AtomicInteger refcnt;
}
```

3. There is no coherent useage of spacing at the beginning and at the end of the if conditions. All styles are valid but it's preferrable to use always the same.

```
if (componentId != null && _logger.isLoggable(Level.FINEST))
if( env instanceof Application)
if( env instanceof Application )
if (wsRefMgr != null )
```

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. NetBeans IDE 8.1 (`https://netbeans.org/`) for code inspection.
4. The automatic analysis tools listed in section 1.5.

Hours of works

Time spent by each group member:

- Alberto Maria Metelli: 18 h
- Riccardo Mologni: 15 h

Revision history

Version	*Date*	*Revision description*	*Revision notes*
0.1	1-1-2016	Initial draft	-
1.0	5-1-2016	Final draft	-
2.0	22-2-2016	Final release	Fixed introduction and some terminology.