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.

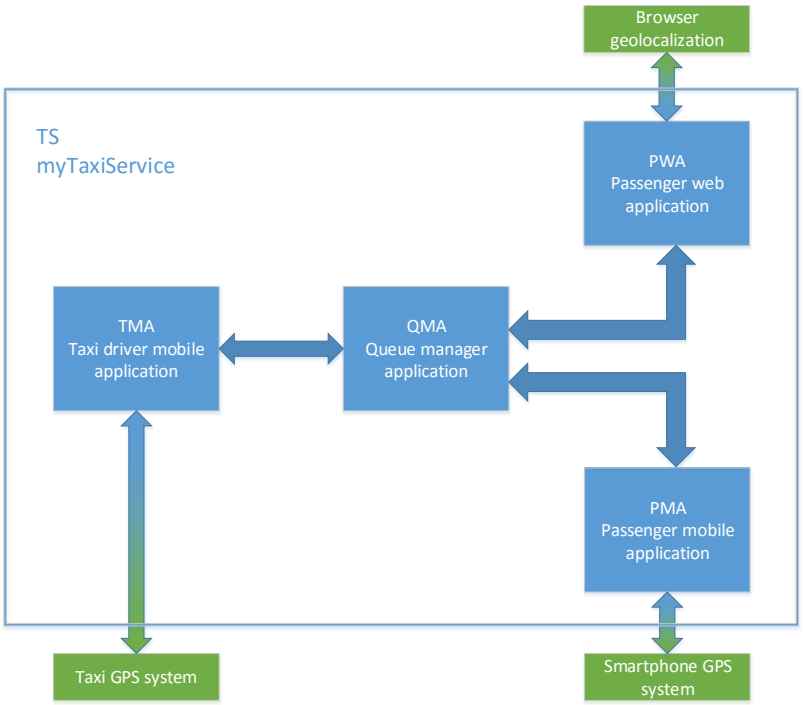

Figure 1: Block schema representing the conceptual interaction between subsystems.

- 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.

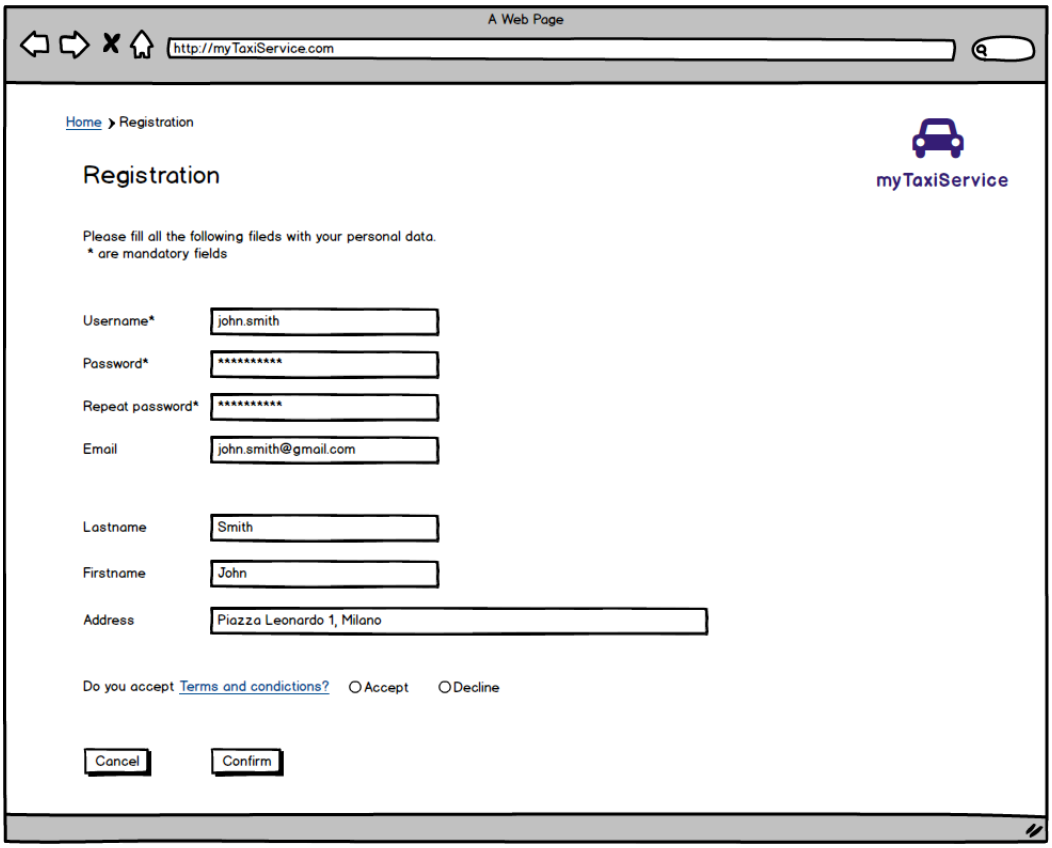

Figure 2: Registration form (web interface)

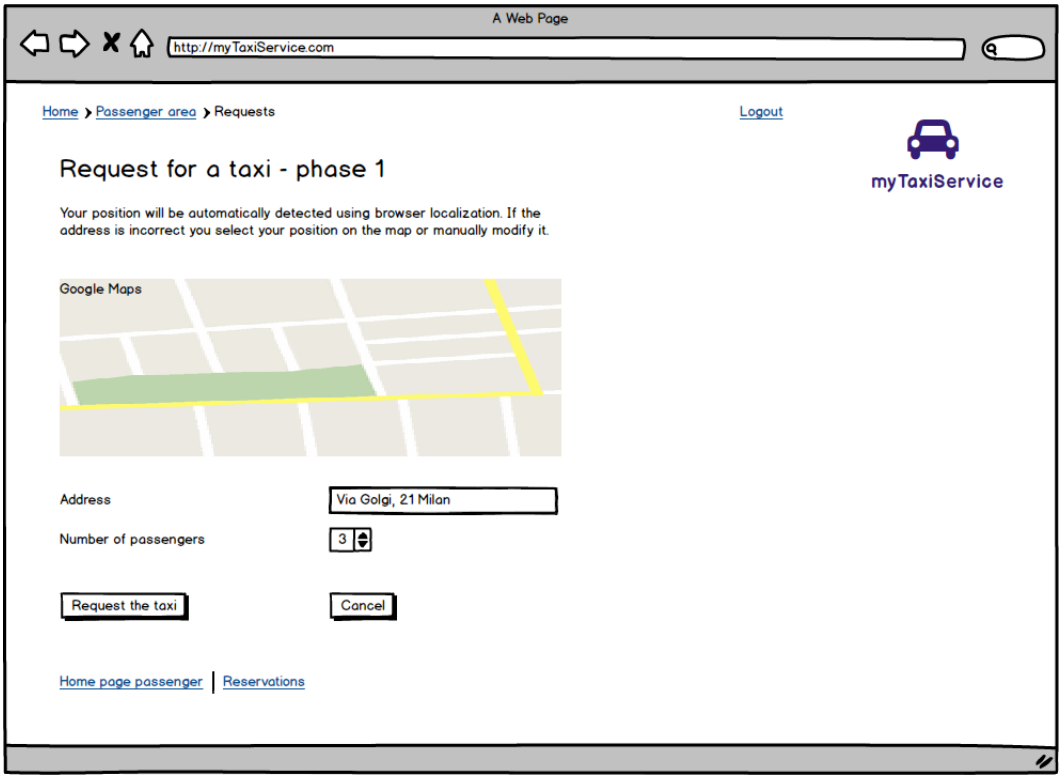

Figure 3: Request for registered passenger 1 (web interface)

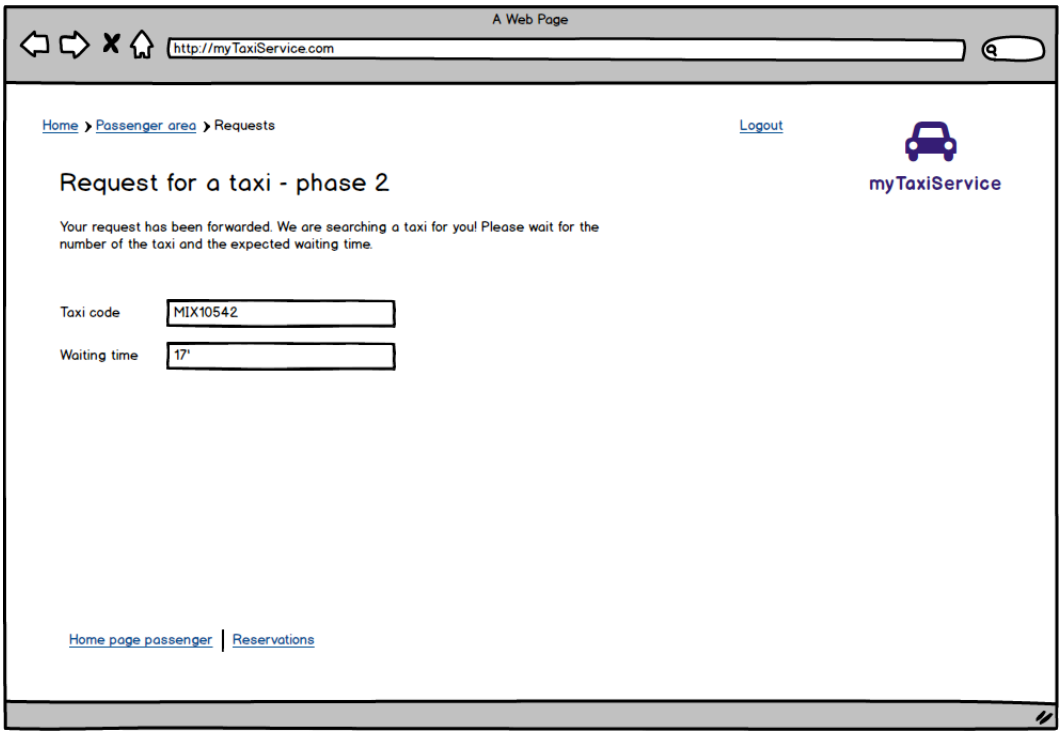

Figure 4: Request for registered passenger 2 (web interface)

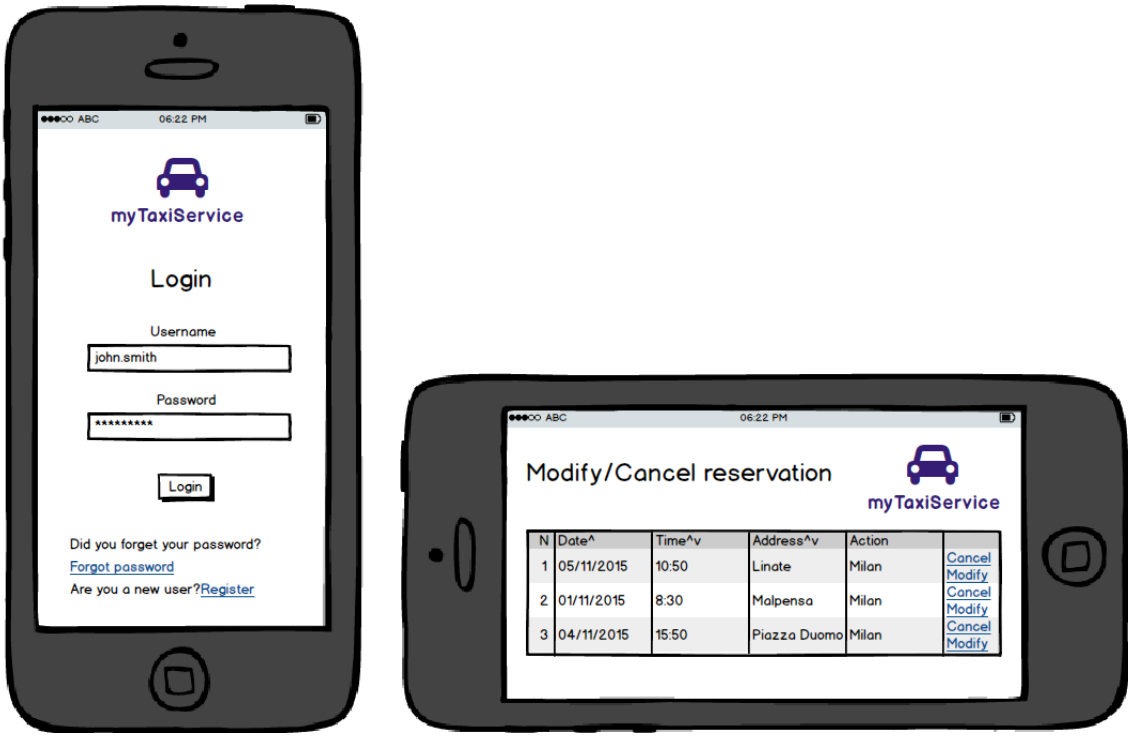

Figure 5: Login (mobile interface) - Modify/Cancel reservation (mobile interface)

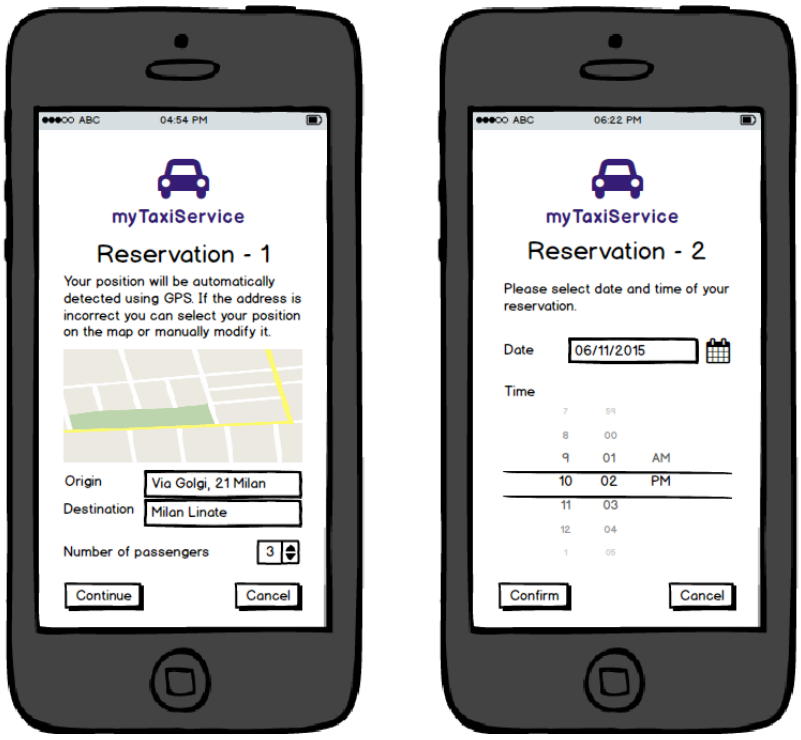

Figure 6: Reservation 1/2 (mobile interface)

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.

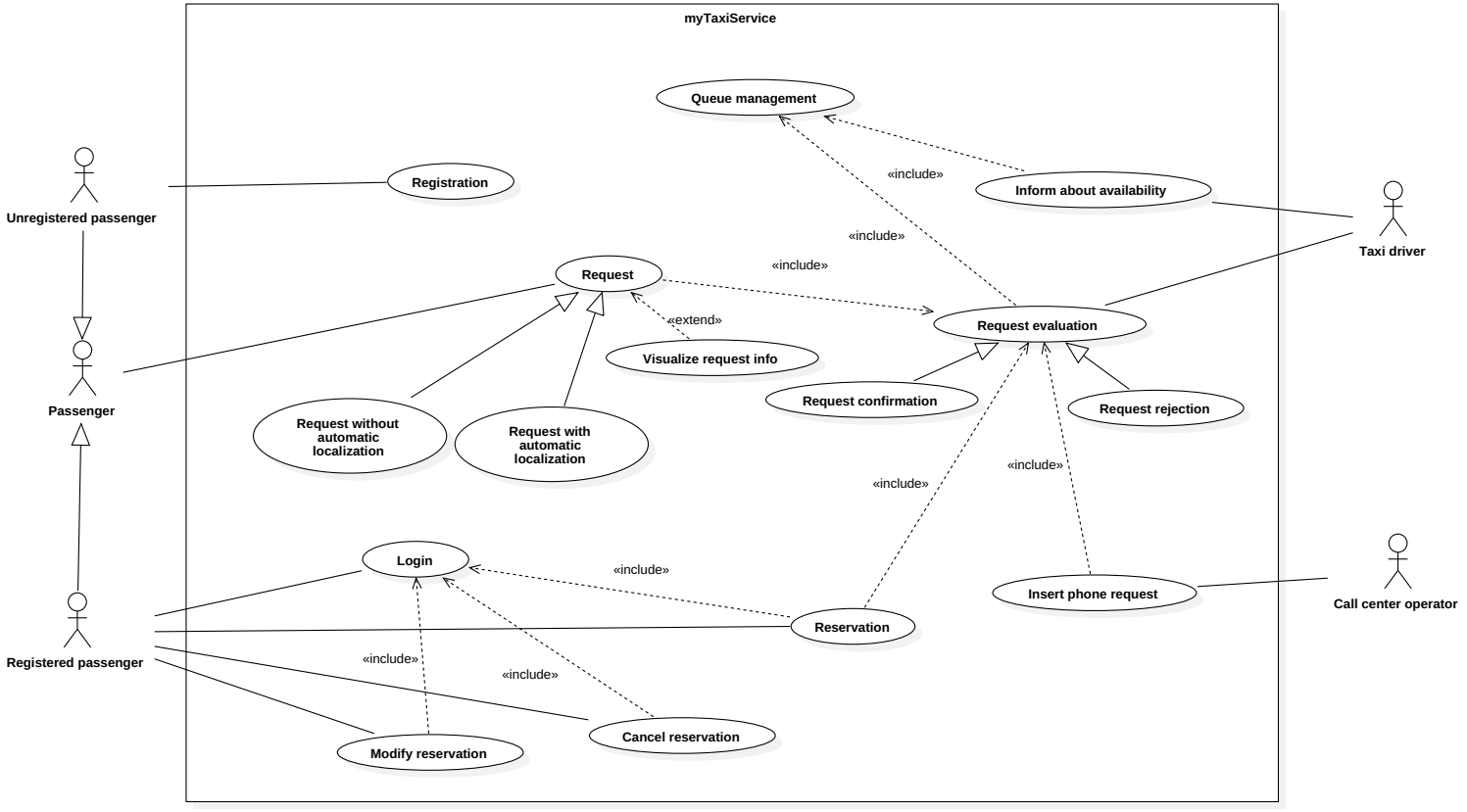

Figure 7: UML Use Case Diagram

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.

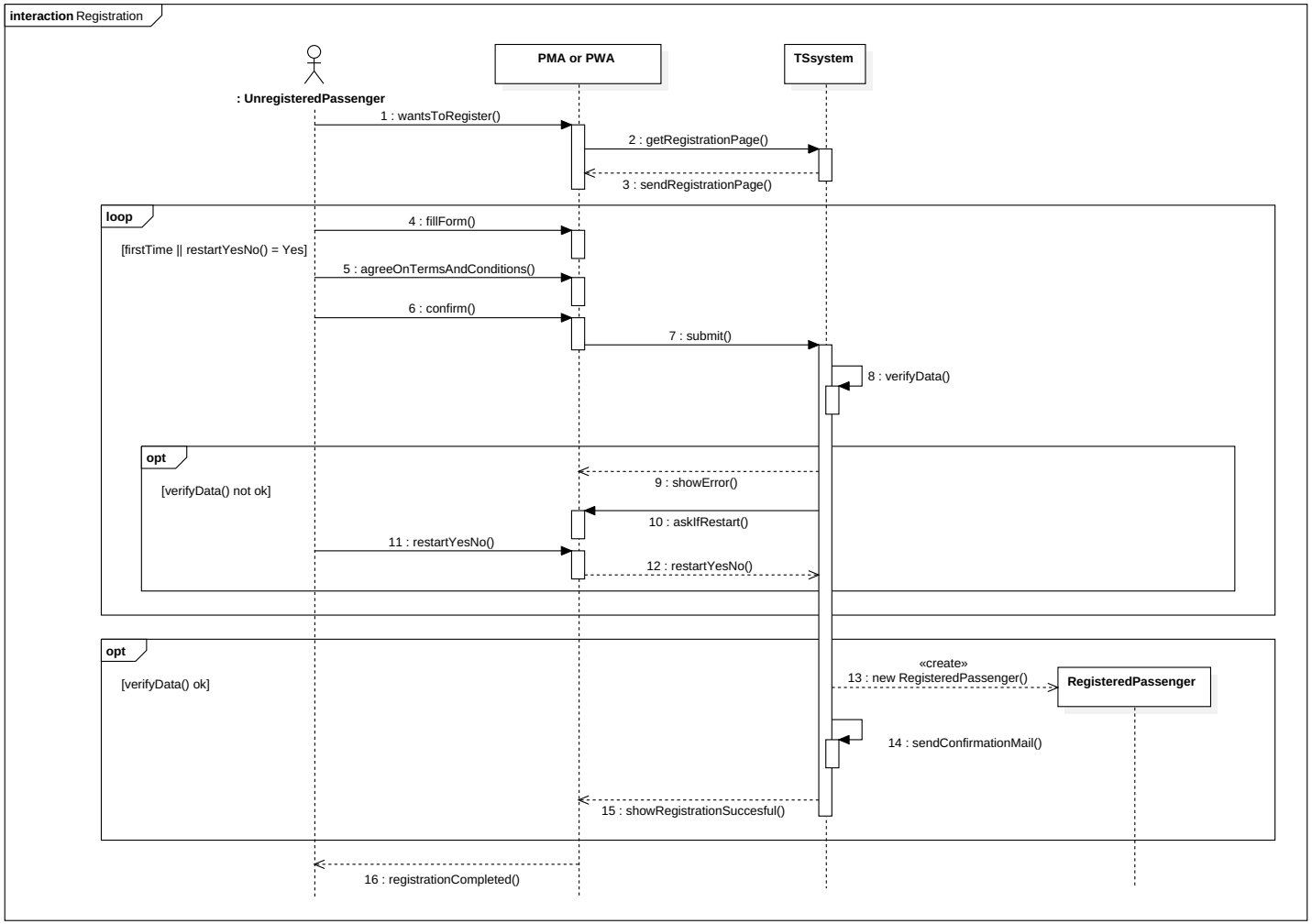

Figure 8: Registration - UML sequence diagram

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.

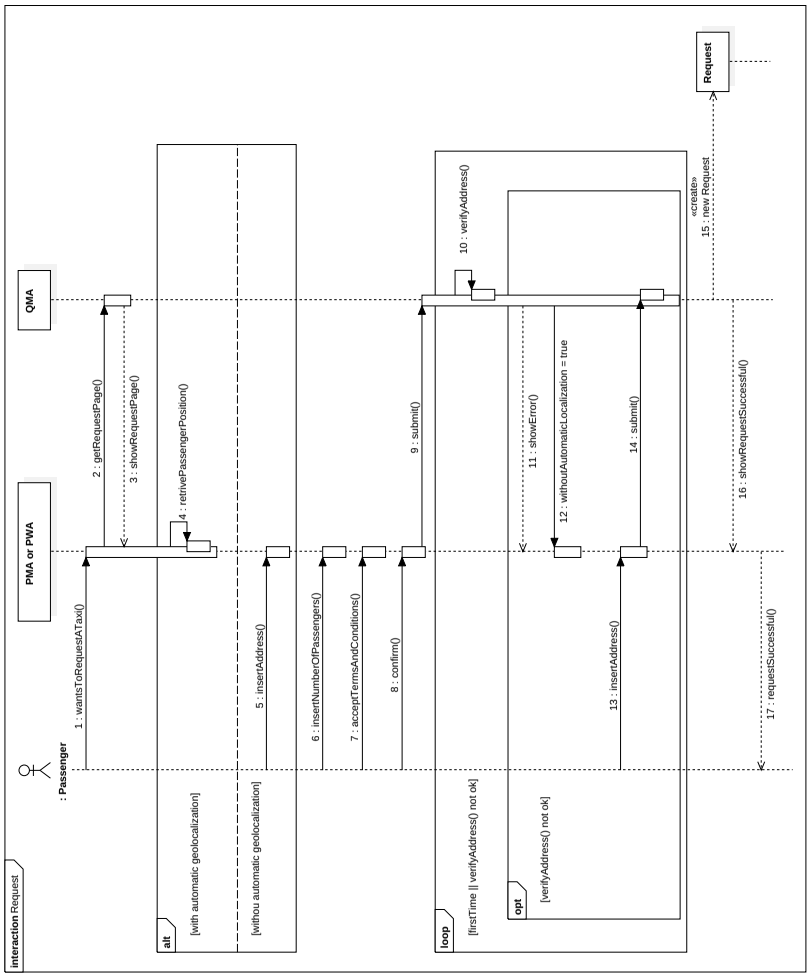

Figure 9: Request - UML sequence diagram

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.

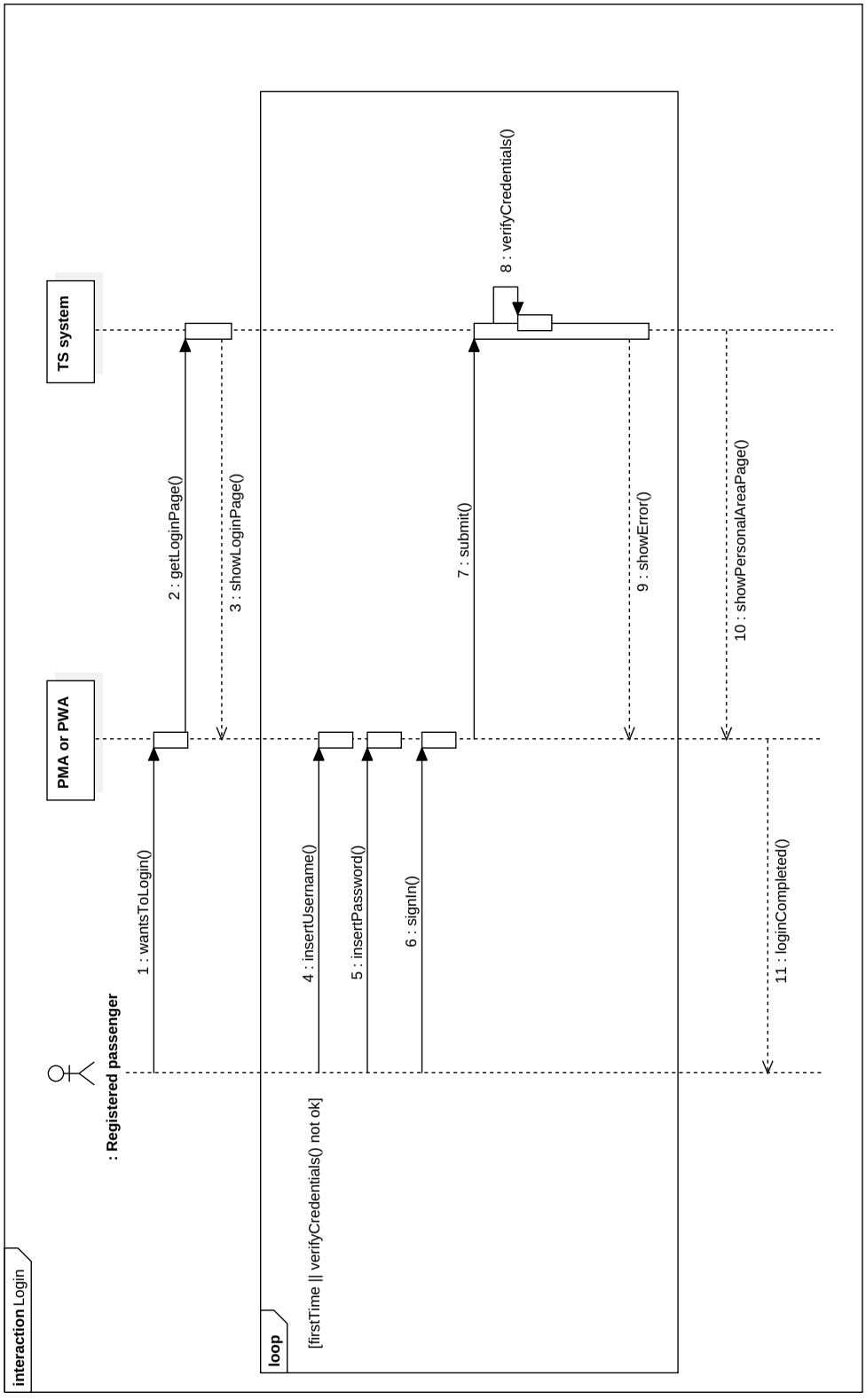

Figure 10: Login - UML sequence diagram

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.

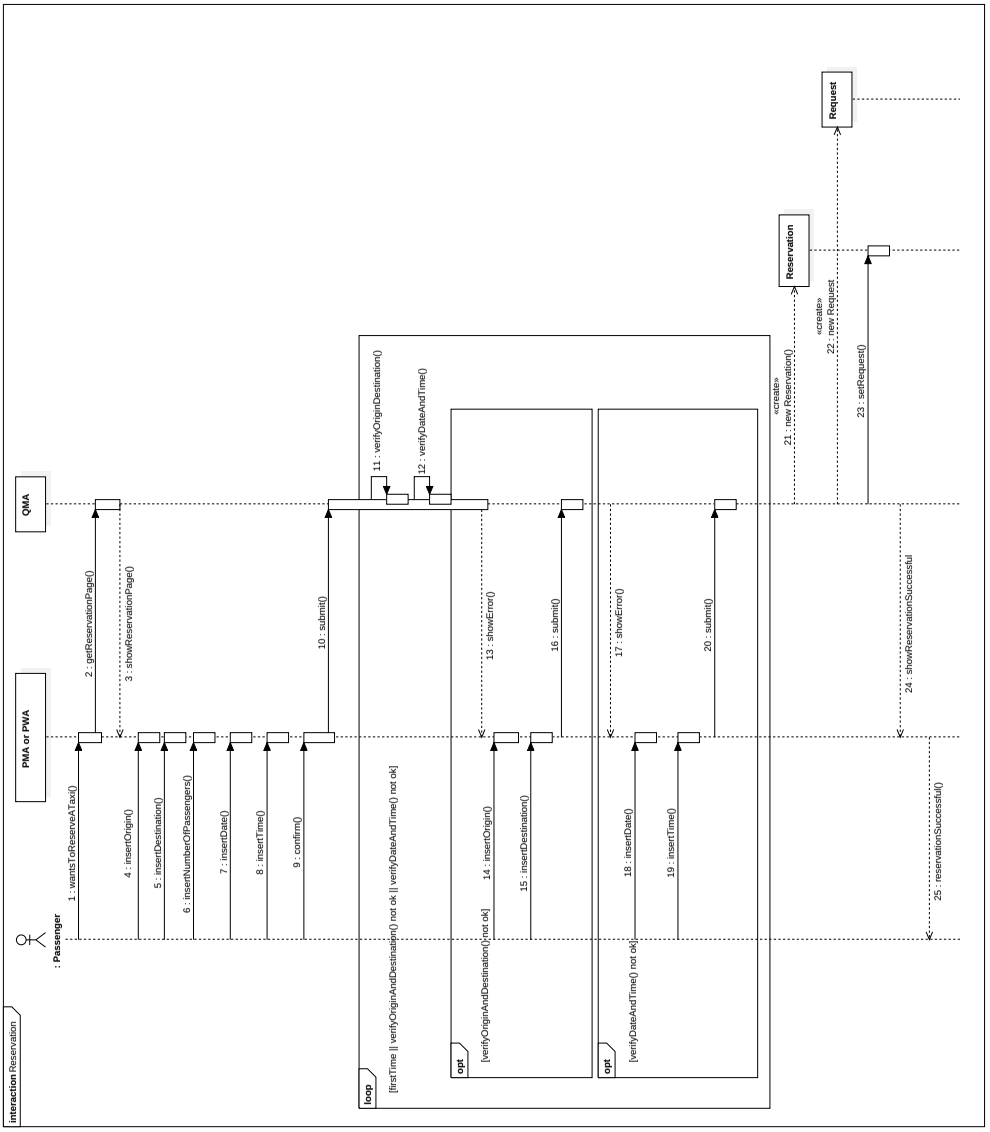

Figure 11: Reservation - UML sequence diagram

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.

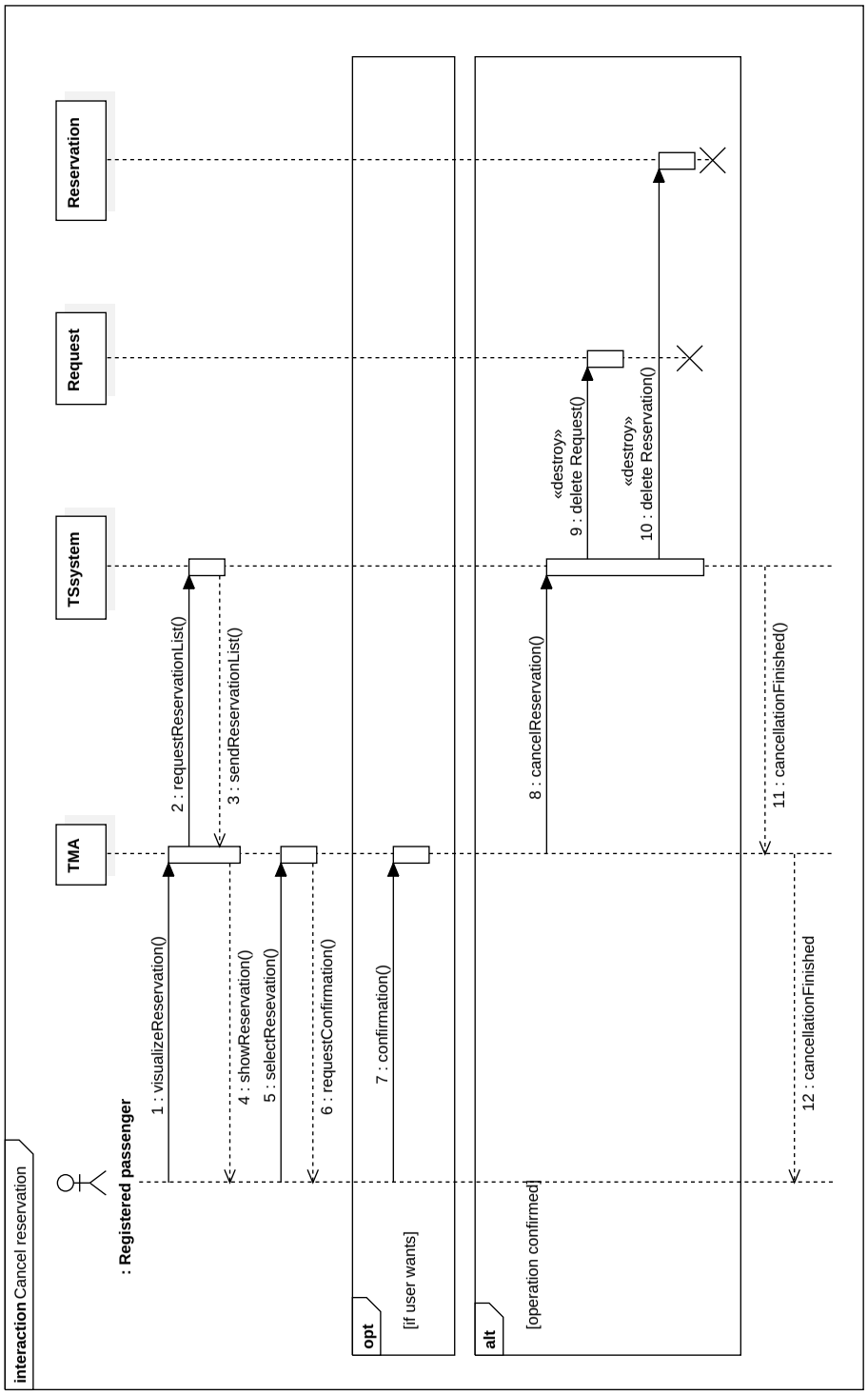

Figure 12: Cancel reservation - UML sequence diagram

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.

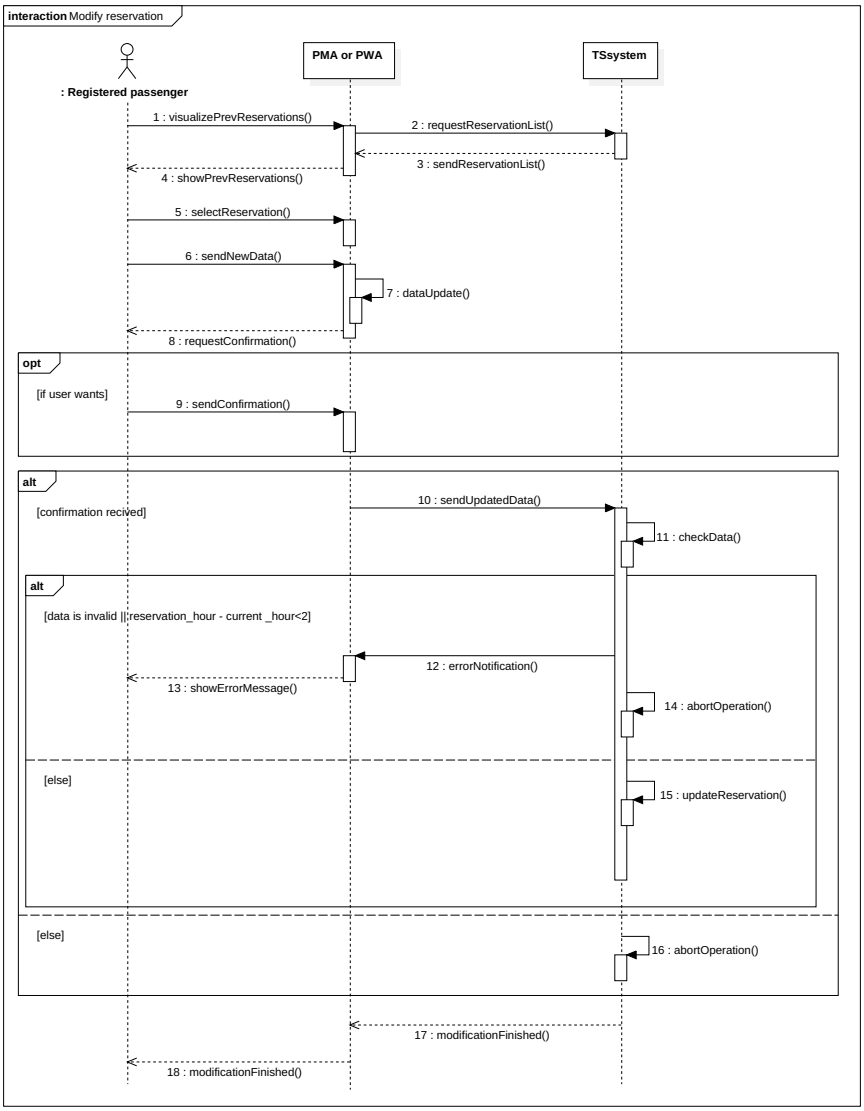

Figure 13: Modify reservation - UML sequence diagram

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.

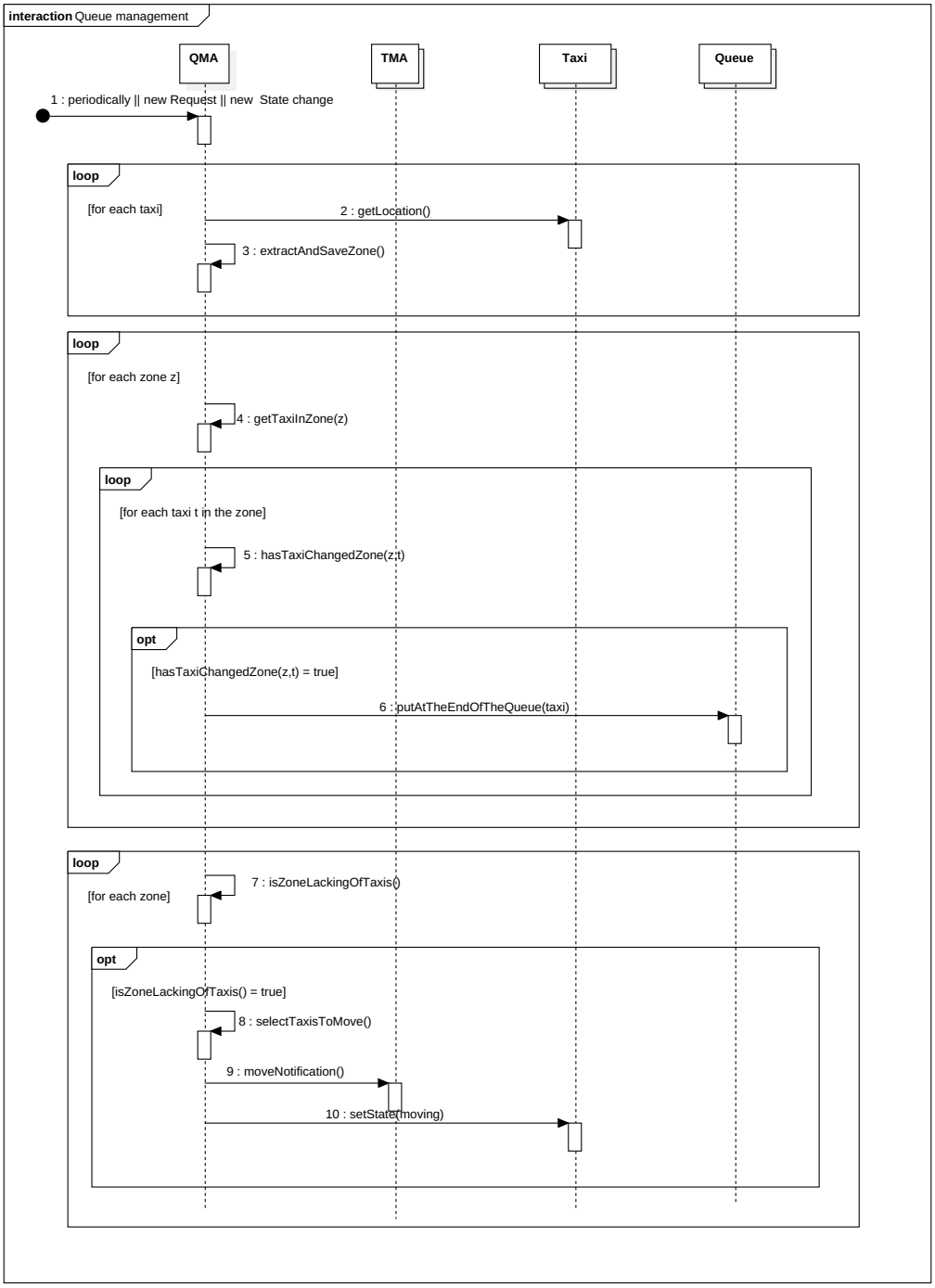

Figure 14: Queue Management - UML sequence diagram

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.

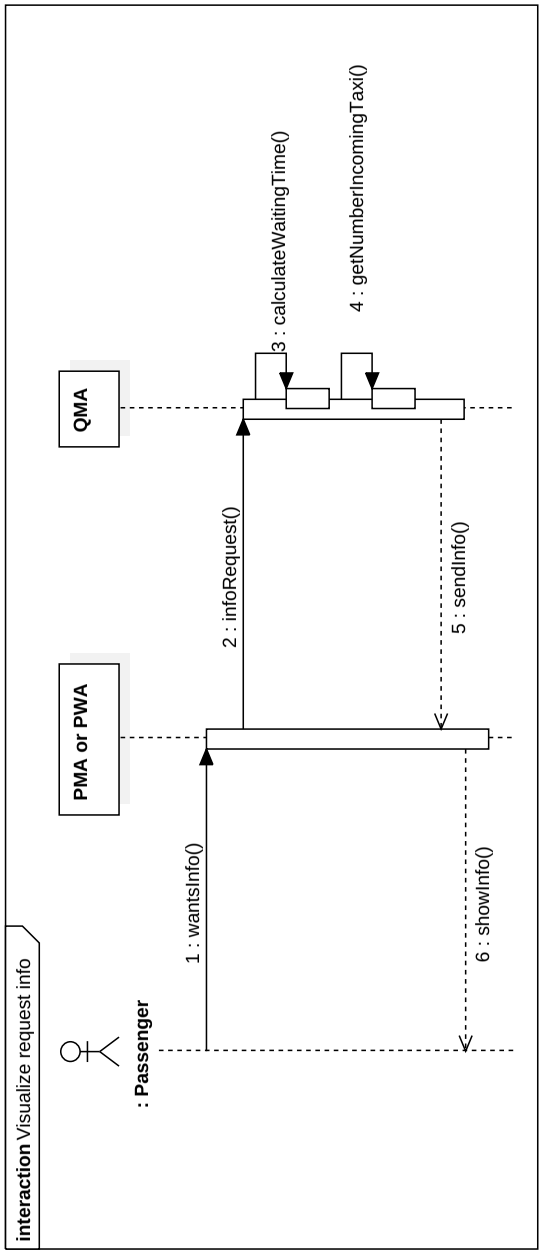

Figure 15: Visualize request info - UML sequence diagram

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.

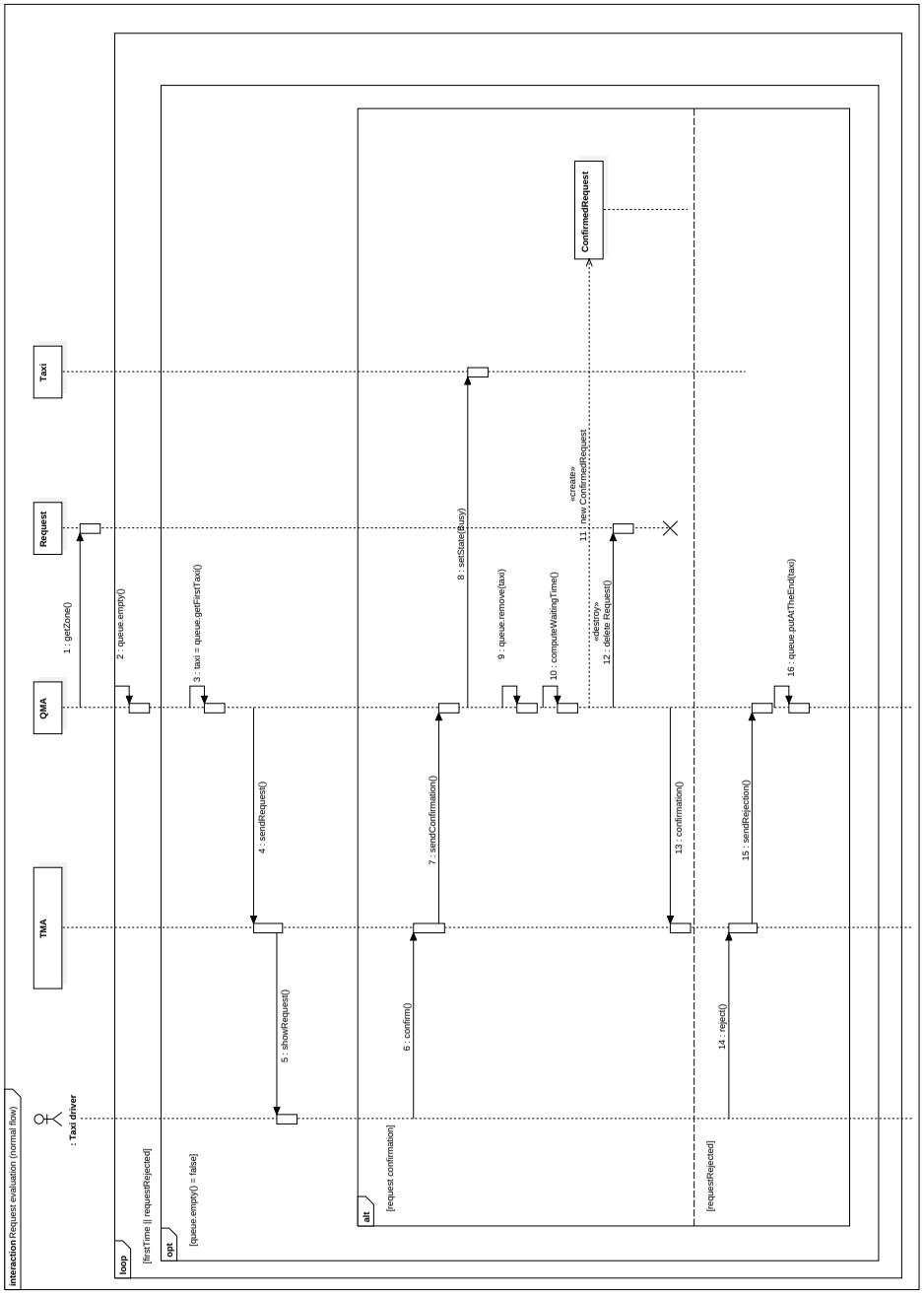

Figure 16: Request evaluation - UML sequence diagram

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.

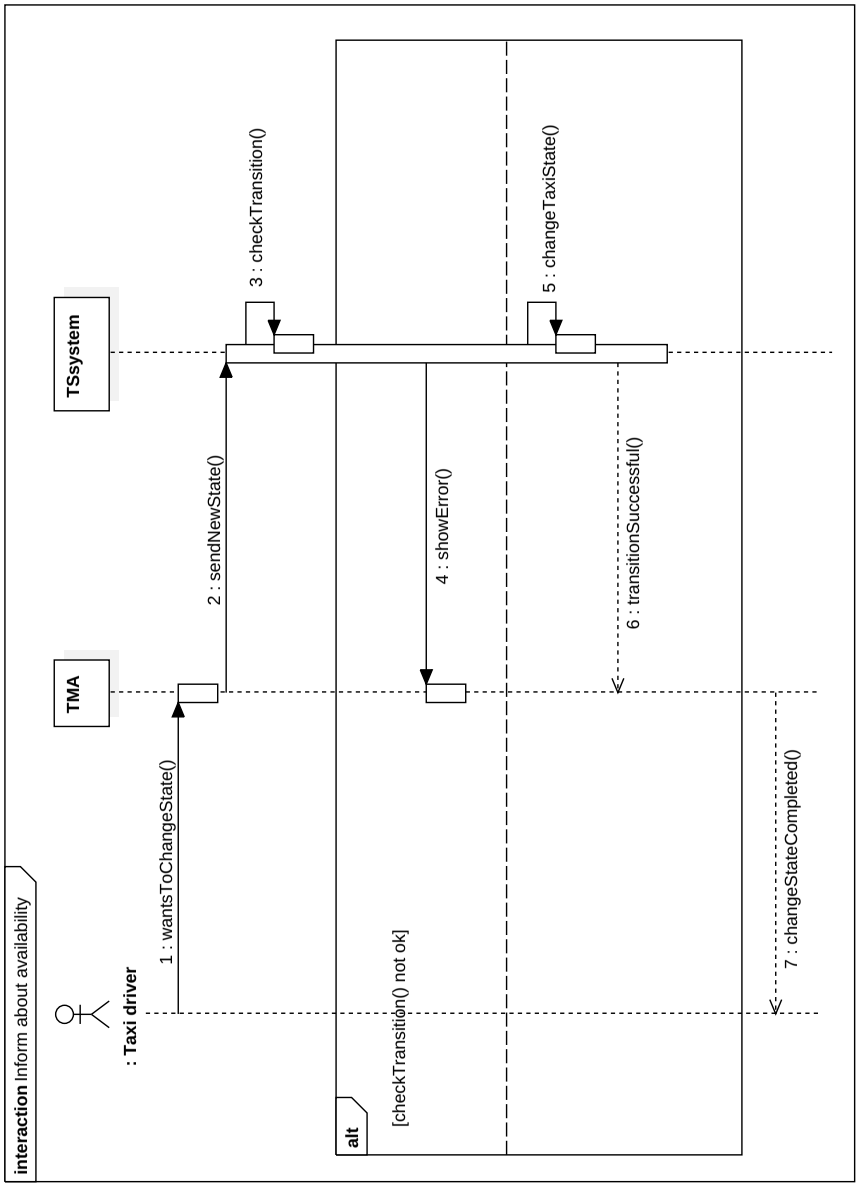

Figure 17: Inform about availability - UML sequence diagram

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.

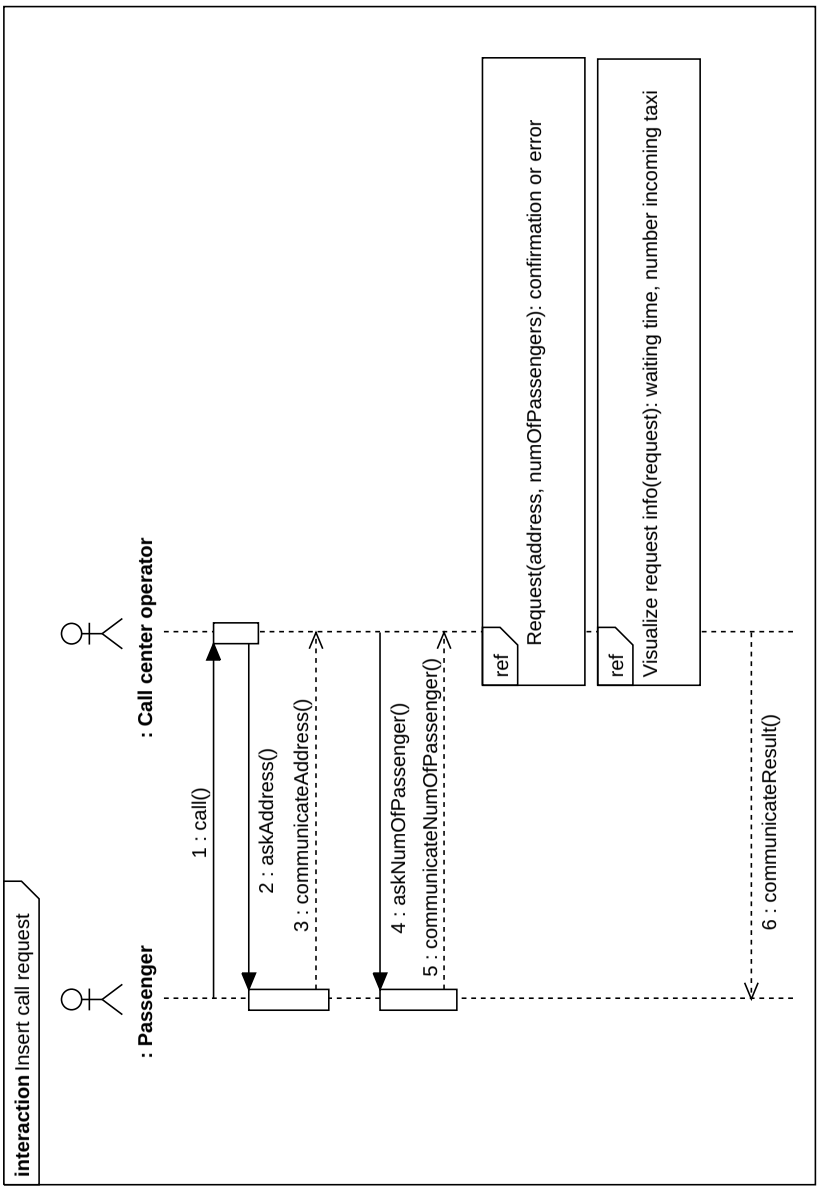

Figure 18: Insert call request - UML sequence diagram

3.5 Other UML diagrams

3.5.1 Class diagram

In this subsection a conceptual UML Class Diagram is shown.

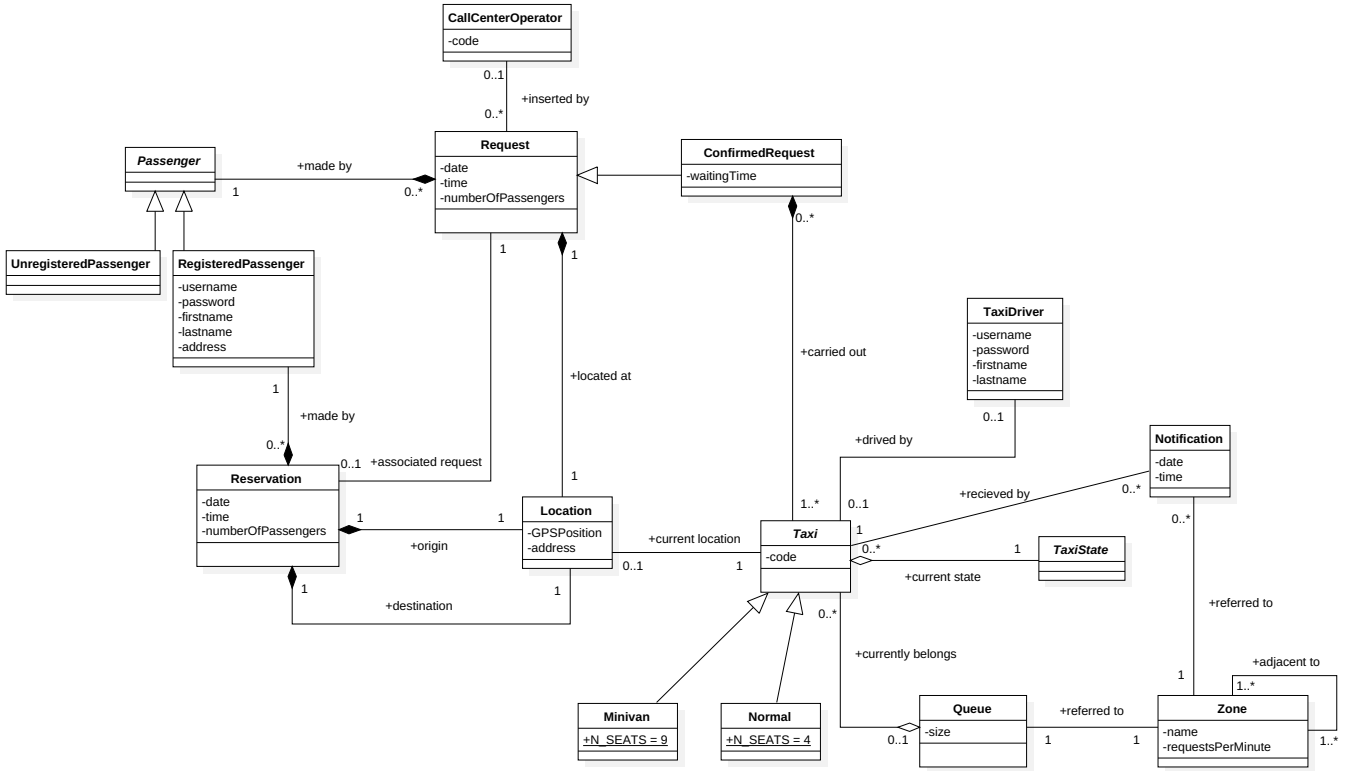

Figure 19: UML Class Diagram

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.

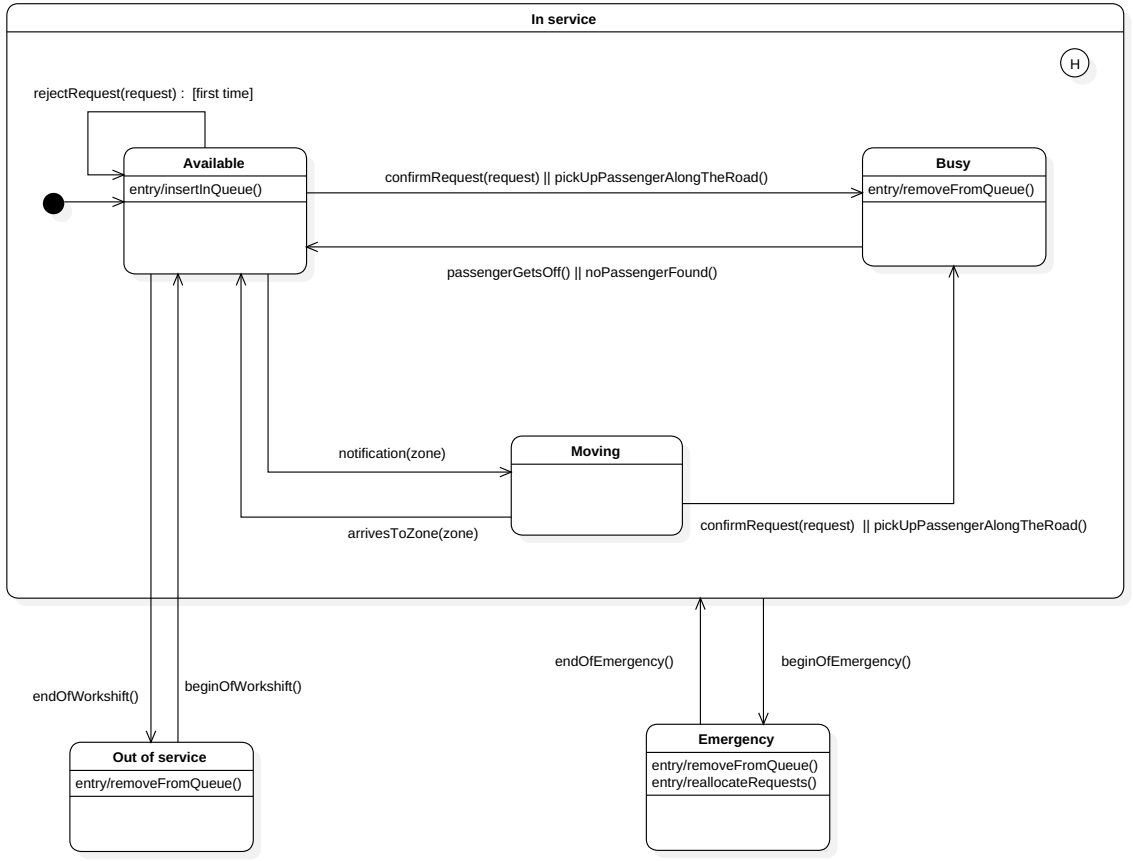

Figure 20: Taxi state - UML State Chart diagram

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.

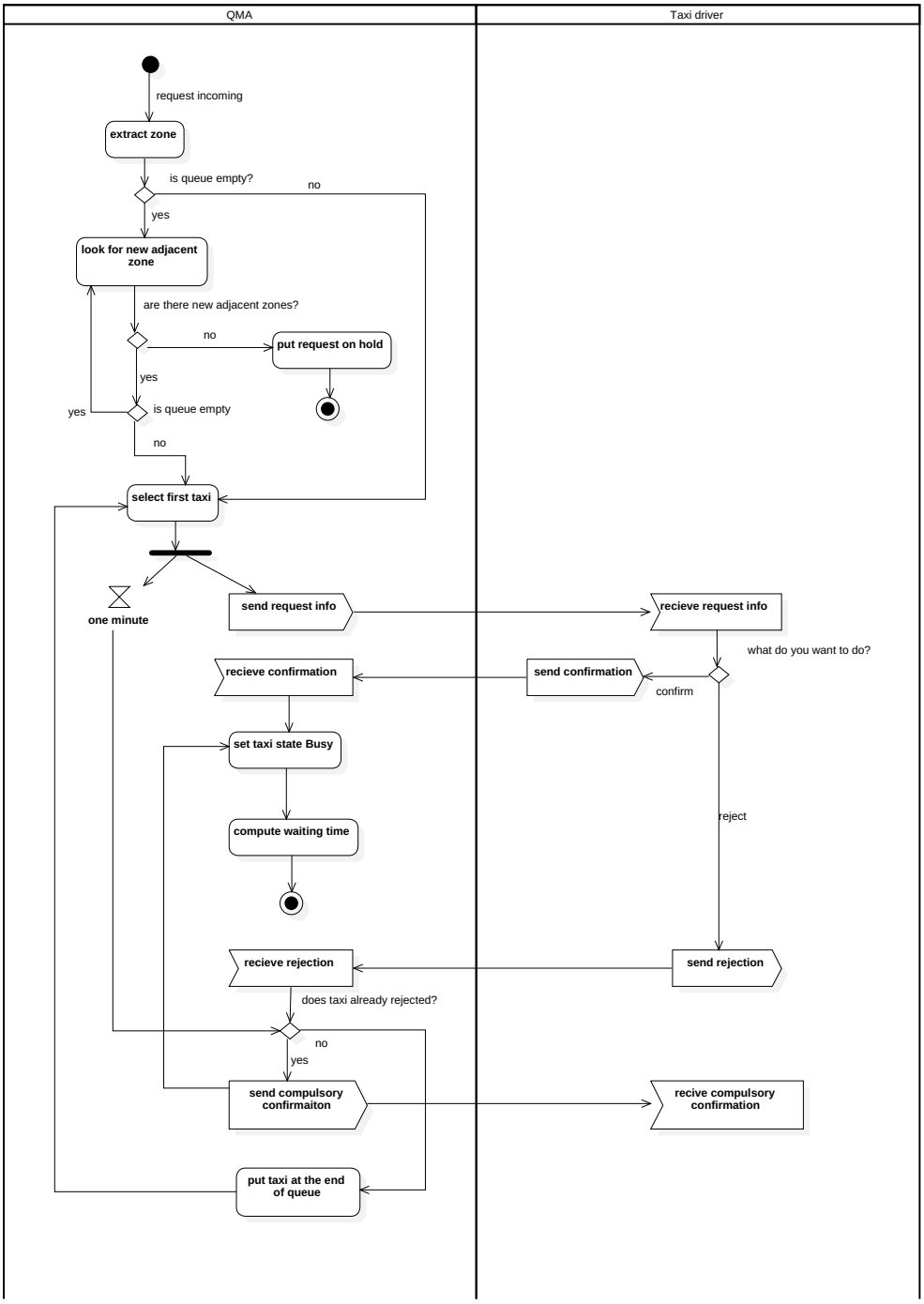

Figure 21: Request evaluation - UML Activity diagram

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.

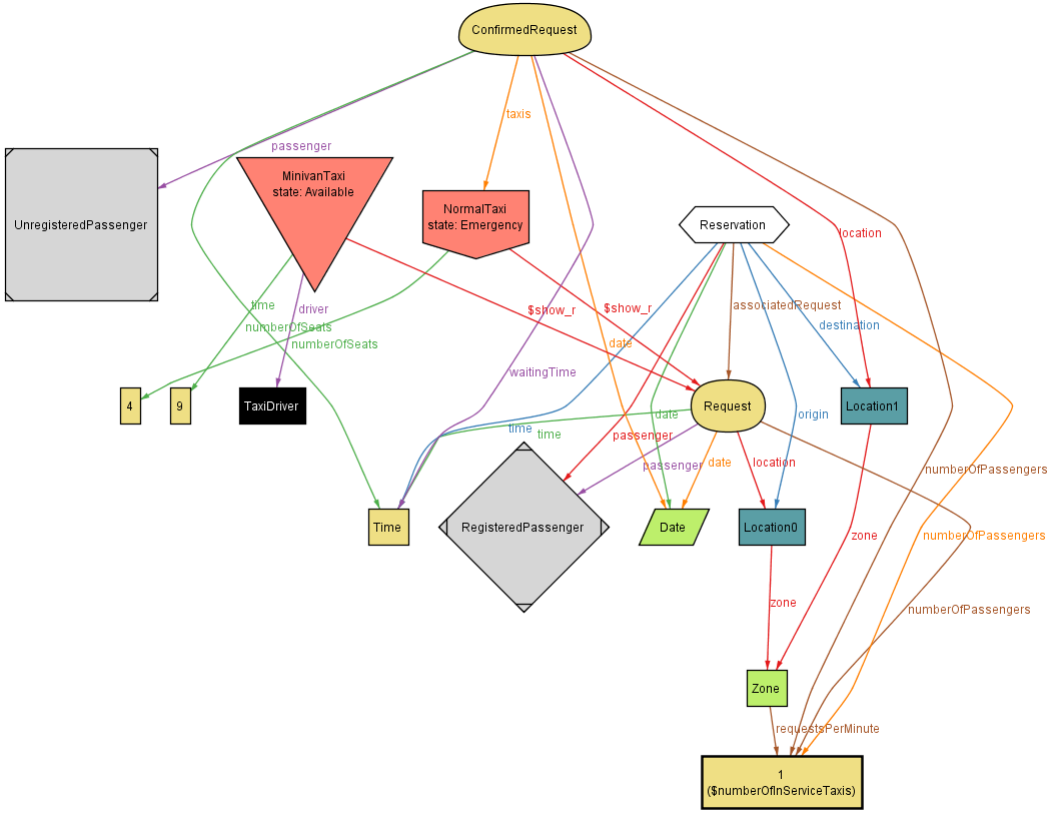

Figure 22: World generated by predicate **pred** `show`

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.

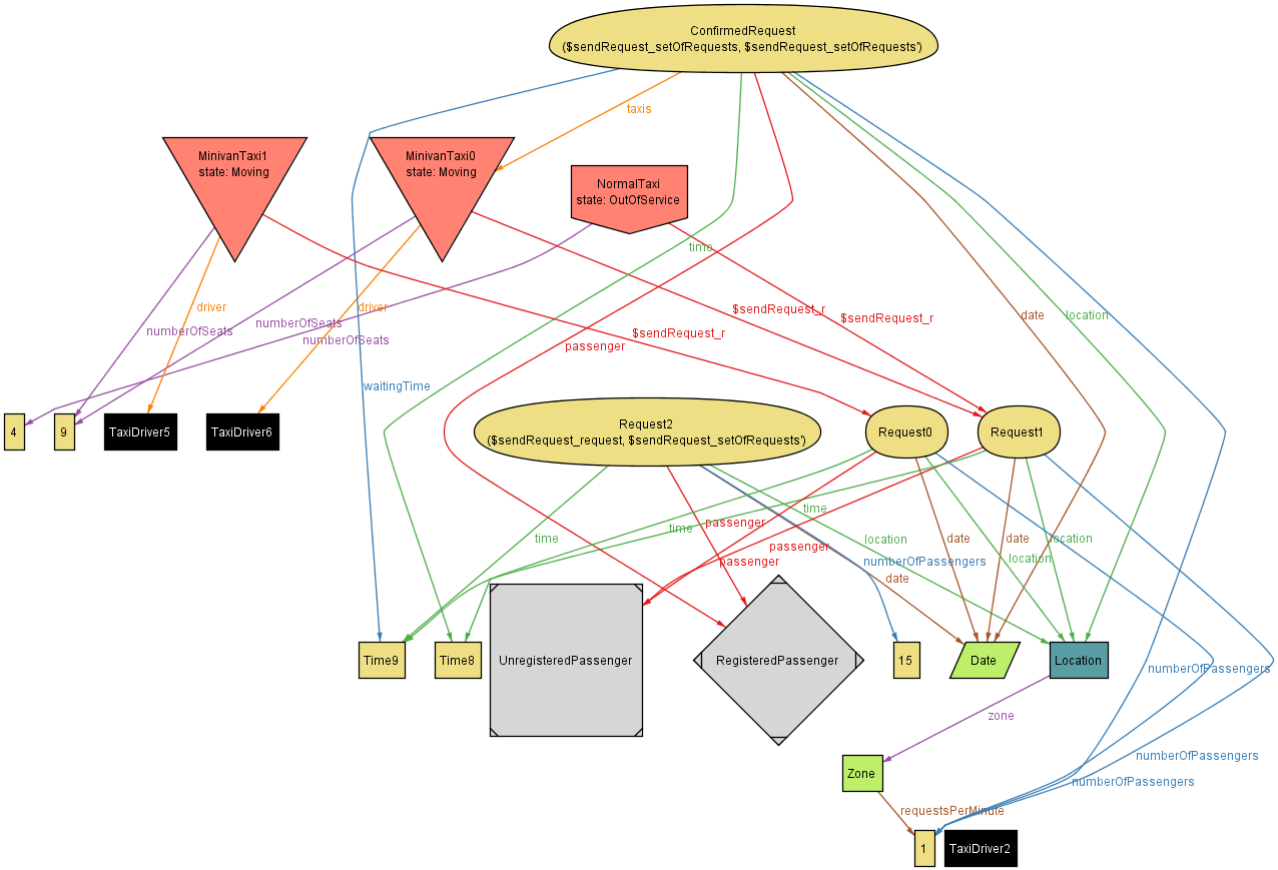

Figure 23: World generated by predicate **pred** `sendRequest`

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).

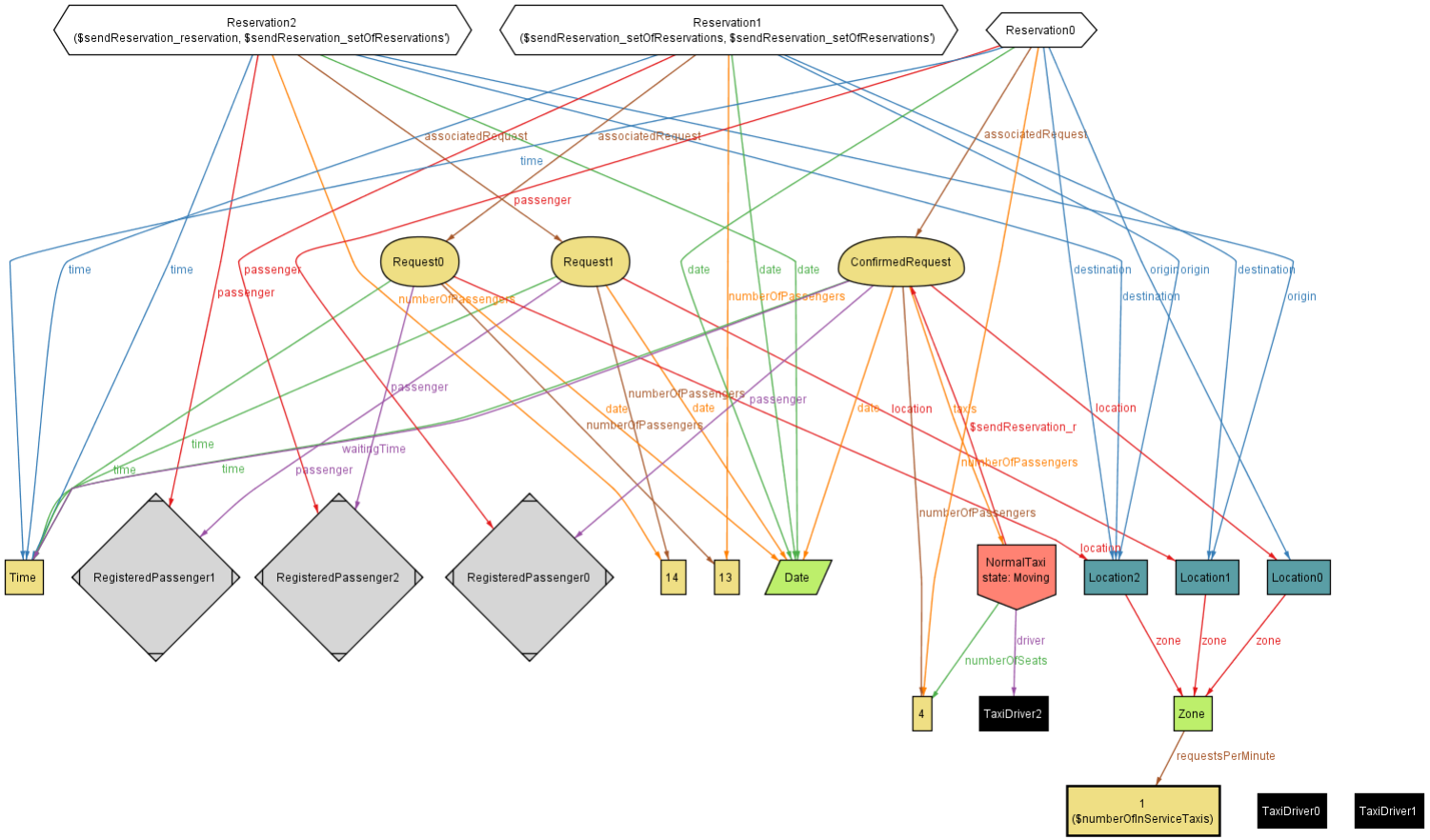

Figure 24: World generated by predicate **pred** `sendReservation`

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.

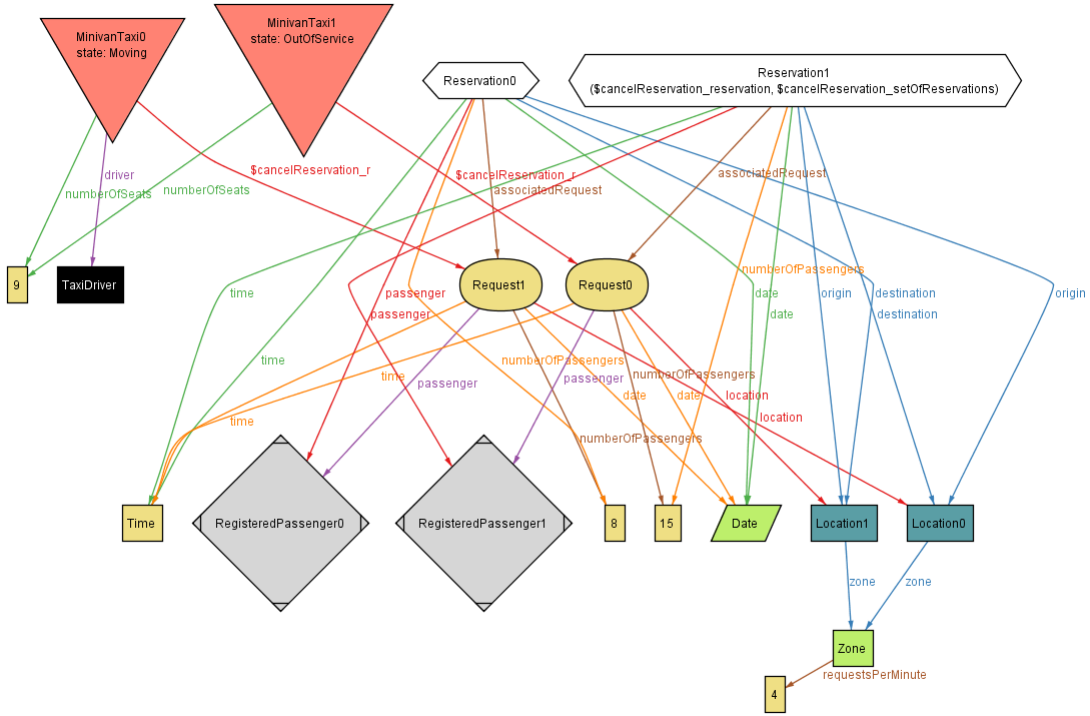

Figure 25: World generated by predicate **pred** `cancelReservation`

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.

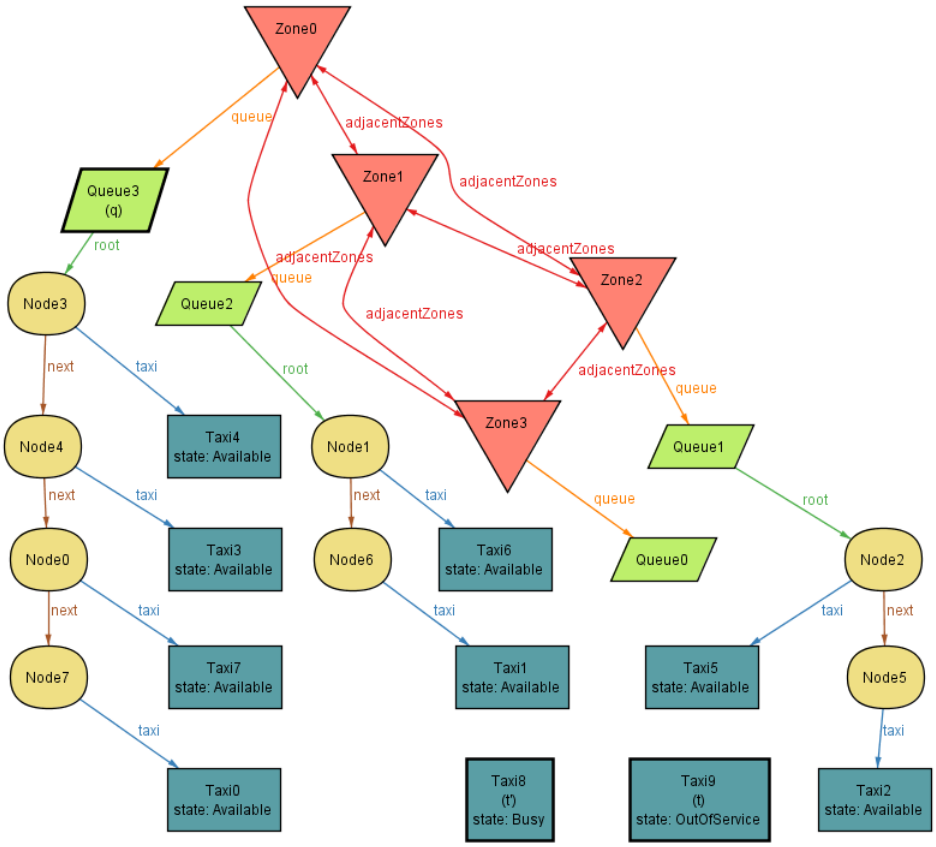

Figure 26: World generated by the predicate **pred** `showQueues`

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.

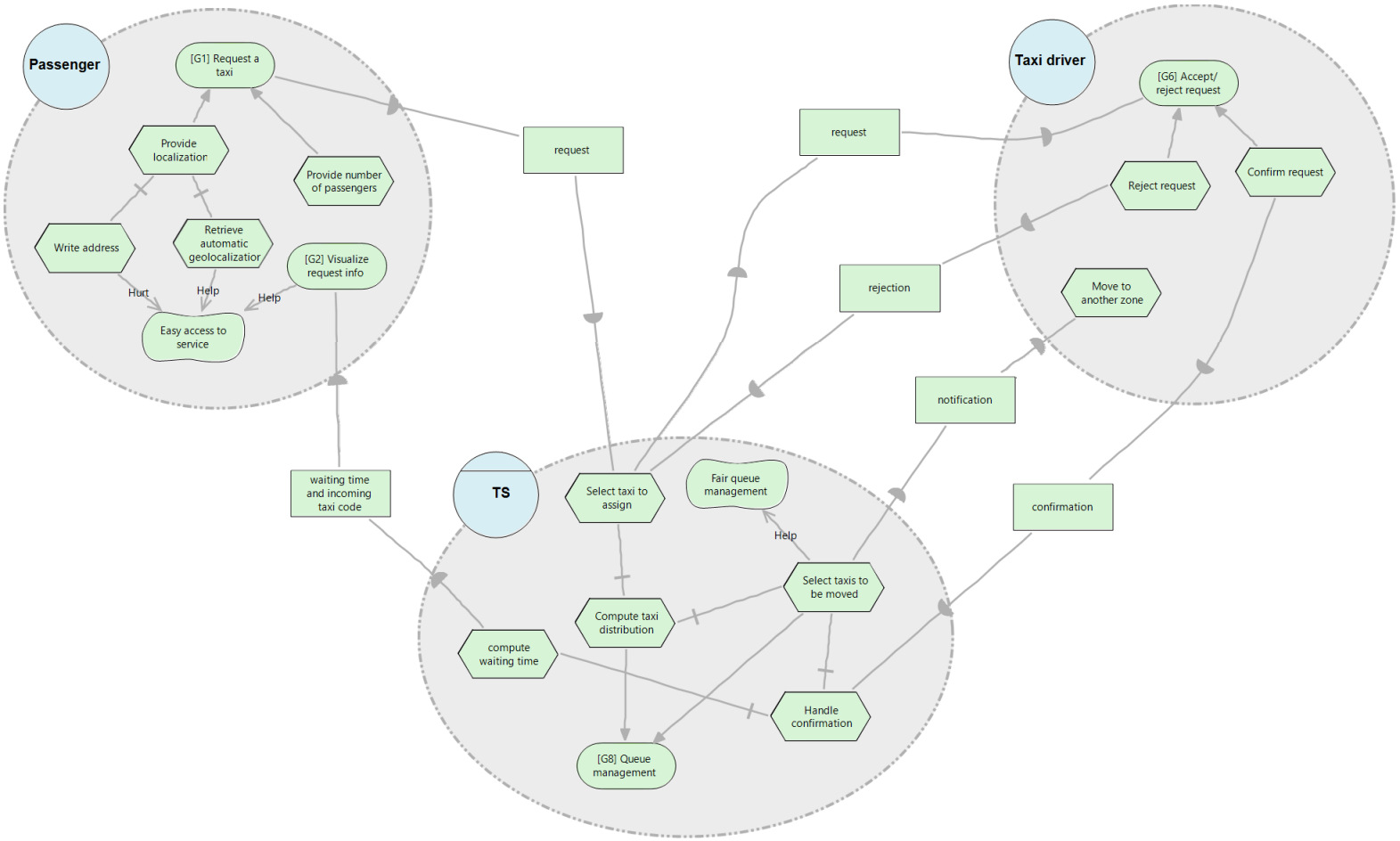

Figure 27: i* model

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.