Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

myTaxiService
Requirements Analysis and Specification Document

Supervisor:
Prof. Elisabetta Di Nitto

Students:
Alberto Gasparin
Vito Matarazzo

Year 2015-2016

Contents

1 Introduction

1.1 Purpose

This document collects all the requirement for myTaxiService project, stating them in a way that allows tracing from the original (ambiguous, unrefined) form to the analysed and refined form.
The intended audience for this document are the stakeholders and the technical community.

1.2 Actual system

We assume that the city that wants to adopt our software system is now lacking an information system to manage taxi service. Thus, the application will be created without integration with any legacy system.

1.3 Product scope

The product that will be developed is a software application called myTaxiService. The aim of the product is

1. to improve the organization of taxi service in a large city simplifying the communication between customers and taxi drivers, having the system acting as a dispatcher.
2. guarantee a fair management of taxi queues

The city will be divided into zones, each zone will be associated to a taxi queue that will constantly be updated by the system, adding free taxi and removing the unavailable ones. Taxi drivers will use a mobile application (previously installed in their smartphone/tablet) to inform the system about their availability and to confirm that they are going to take care of a certain call, while passengers would access the system via mobile or web application. Passengers will be able to request a taxi immediately or to reserve one for a specific date (and time). The development of a set of basic APIs to allow developers to communicate with myTaxiService is also part of the project, so that third-party apps can use these APIs to take advantage of or extend the functionality of the existing services.

1.4 Identifying Stakeholders

The system that will be developed is addressed to every cities that have at least one taxi company working in it, in particular the large ones that would take advantage from the optimization of taxi management. Therefore, our typical stakeholders can be governments of those cities, that want to organize all the taxis working in them, or single private companies, for their own taxi fleet. The typical user of this application may be people that frequently use taxis, who will speed up the process of requesting a taxi thanks to their registration to the service, but also occasional users who are used to mobile or web apps. Those users are also possible stakeholders due to the fact that they benefit of our product. The same can be said about taxi drivers.

1.5 Actors

Administrator: this type of user is the one allowed to register new taxi driver to the system and give them their unique username and password so that they can log into the system via mobile application. Administrators are also the ones that perform maintenance.

Visitor: this users are not yet registered to the system and so they cannot interact with it, apart for signing up.

Registered User: simply called **users**, they are already registered to the system. There are three types of users of this kind: Passenger, Taxi driver and Developer.

Passenger: they can access the system via web application or mobile application, and ask for services (Request a taxi in a given location, Reserve a taxi for a given data and time and more).

Taxi driver: they log into the system via a mobile application in order to communicate with the system.

Developer: they interact with the system only via web browser, no mobile app has been created. They can request an API key and consult the documentation about the system's APIs.

1.6 Goals

- [G1] Keep trace of users of the application
- [G2] Answer to passengers' requests within an acceptable time.
- [G3] Grant a correct management of the taxi queues.
- [G4] Keep trace of the status of each taxi driver.
- [G5] Allow developers to extend the application by adding new functionalities.

1.7 Definitions, acronyms, abbreviations

1.7.1 Definitions

API Key: is a code passed in by computer programs calling an API (application programming interface) to identify the calling program, its developer, or its user to the Web site.

Contract: a legally binding document agreed upon by the customer and supplier. This includes the technical and organizational requirements, cost, and schedule for a product. A contract may also contain informal but useful information such as the commitments or expectations of the parties involved.

Customer: the person, or persons, who pay for the product and usually (but not necessarily) decide the requirements. In the context of this recommended practice the customer and the supplier may be members of the same organization.

Functional requirement: a requirement that specifies a function that a system or a system component must be able to perform.

Queue: a list in which items are appended to the last position of the list and retrieved from the first position.

Requirement: (1) a condition or capability needed by user to solve a problem or achieve an object. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation or capability as in (1) or (2).

User: the person, or persons, who operate or interact directly with the product. The user(s) and the customer(s) are often not the same person(s).

1.7.2 Acronyms

API : Application Programming Interface.

DB : Database.

ETA : Estimated Time of Arrival.

GPS : Global Positioning System.

UML : Unified Modelling Language

1.7.3 Abbreviations

[**An**] **:** n-assumption.

[**Gn**] **:** n-goal.

[**FRn**] **:** n-functional requirement.

[**NFRn**] **:** n-non functional requirement.

1.8 Reference Document

Specification Document: myTaxiService Project AA 2015-2016.pdf.

IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications.

IEEE Std 1233, 1998 Edition, IEEE Guide for Developing System Requirements Specifications.

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.

1.9 Overview

The rest of the document is organized in :

- Section 2: Overall Description, gives general information about the software product with more focus about constraints and assumptions.
- Section 3: Specific Requirements, this part lists all functional and non functional requirements, typical scenarios and use cases. UML diagrams would be presented in this section to better understand the functionality of the system at different level of granularity.
- Section 4: Appendix, this part contains information about the attached .als file and the software used in the development of this document.

2 Overall Description

2.1 Product Perspective

This product is a standalone system and will only require the users an internet connection to work properly. This means that the users of the system do not need to invest in any other software to get the most out of the software system.

2.2 User characteristics

The web application and the mobile application for passengers will be designed in such a way that no technical expertise would be required to the passengers. They will just have to have basic computer skills (which include working with a web browser such as Internet Explorer, Mozilla Firefox or Google Chrome) and/or basic mobile skills. Taxi drivers would need some training because they have to be proficient in using their custom mobile app.

2.3 Constraints

2.3.1 Regulatory Policies

Passengers

1. All personal data send by the user (first-name, last-name, address, number, email, etc) are saved and treated in conformity with the government law in matter of treatment of personal data. In Italy the main disposition are contained in D.Lgs. n. 196/2003.
2. myTaxiService offers the possibility to the users to contact the system via mobile or web application. The system will save only those personal data used to provide an answer to the user's request. Personal data are not forwarded to third party.
3. myTaxiService uses cookies to record the preferences of users and optimize the design of its websites. They make navigation easier and increase the user-friendliness of a website. Cookies are small files that are stored on the user's end device. They can be used to determine whether there has been any contact between the system and the user's end device in the past. Personal data can only be saved in cookies if the user have given his/her consent. By using our website, the user consents to the use and storage of cookies on his/her end device.

Taxi drivers

1. Taxi drivers are subject to the government law in matter of taxi services.
2. Personal data and other details cannot be saved, used or transferred to third parties by taxi drivers.

2.3.2 Hardware Limitations

Every passenger should have a smartphone/tablet or a PC and a working internet connection. While taxi drivers should posses a smartphone/tablet since their custom application is only available on mobile devices.

2.3.3 Interfaces to other applications

Since the system has to compute the optimal path (in terms of travel time) between two locations(taxi driver's current position - pickup location, pickup location - destination) being aware of traffic, it will retrieve data from Google Maps using its APIs.

2.4 Assumption and Dependencies

[A1] When a passenger wants to request or reserve a taxi, he/she must provide a name, the pickup location, the destination, the number of people to carry, the payment method (cash or credit card) and an optional note. The number of people is needed in order to provide the correct taxi in every situations. In the note customers can write if they have some particular things to carry or can specify more precisely the place of meeting.

[A2] When making a reservation, passenger must indicate the information written above and also the date and time of meeting.

[A3] Reservation can be cancelled except if they occur within 15 minutes from pickup time. In this case the system will reject the action.

[A4] There are two categories of taxis: normal auto vehicles that can transport up to 3 people and minivans, up to 7 people.

[A5] All taxi drivers must be logged in the system in order to receive updates and requests in every moment.

[A6] All passengers must be logged in the system in order to interact with it.

[A7] When a taxi driver doesn't answer to a request within 1 minute, it is automatically considered as refused.

[A8] When the system receives a request from a zone in which there is no taxi available, the search is forwarded to the adjacent zones.

[A9] The ETA indicated by the system is a good approximation of the real time thanks to the routing algorithm used by the system.

[A10] Only taxi driver hired by the company can access the system. To provide that an administrator will register them.

[A11] Taxi drivers status can be:

- *free*, when he/she is ready to receive passenger's request.
- *unavailable*, when he/she is occupied by a passenger or when he/she is having a break.

[A12] Taxi Driver that accept a call will go in the desired address in the shortest possible time.

[A13] If a person that has requested a taxi is not present at the pickup location at the designated time the taxi driver changes his status from unavailable to free and the ride is cancelled.

[A14] At the end of the ride every person pays the right amount of money.

[A15] All API requests are made to: https://api.mytaxiservice.com/$< version >$.

3 Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

Here we list a series of mockups for the web application and also for the mobile application We start from the Passenger's side:

3.1.1.1 Sign Up Figure 1 and Figure 2 shows respectively the registration form in the web application and in the mobile application.

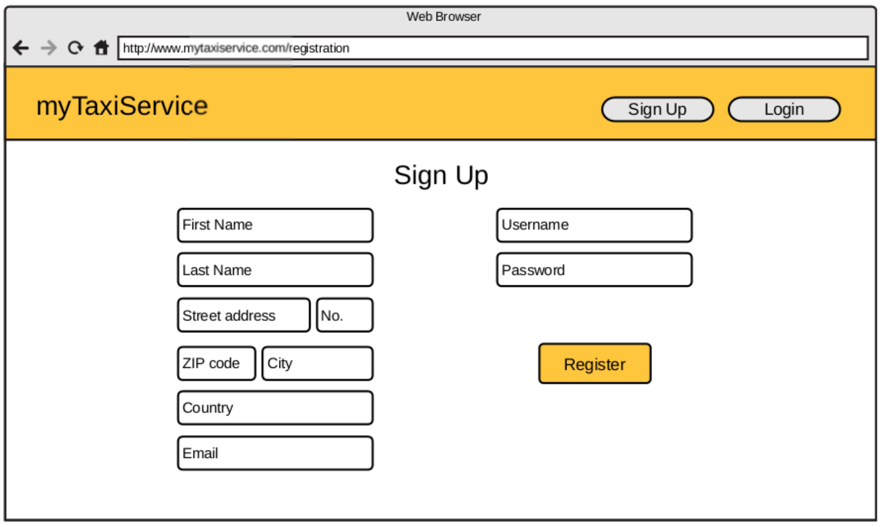


Figure 1

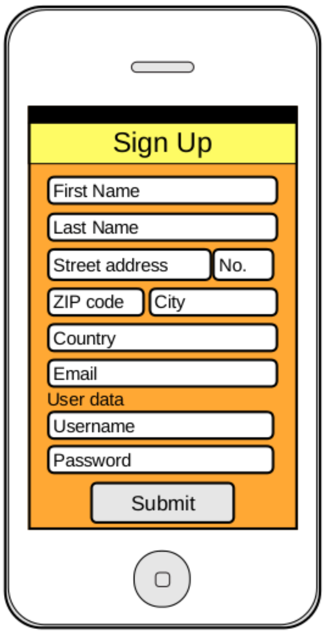


Figure 2

3.1.1.2 Login Figure 3 and Figure 4 shows respectively the login form in the web application and in the mobile application.

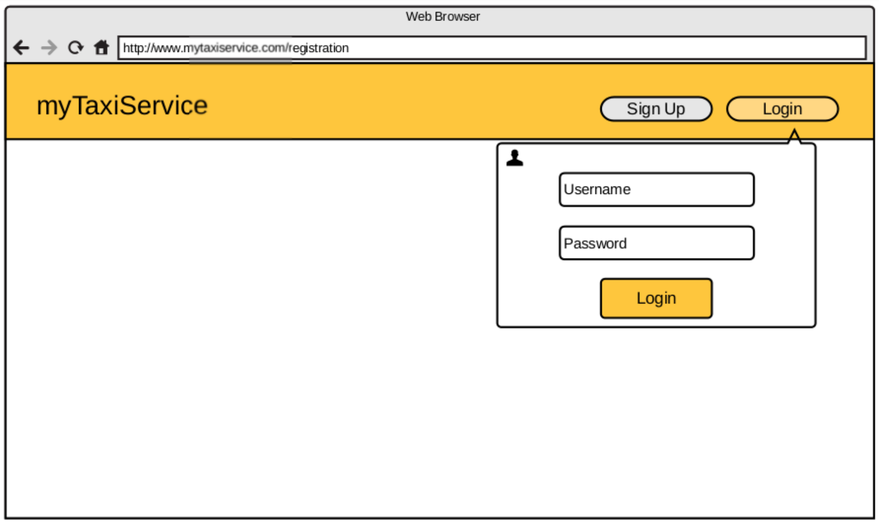


Figure 3

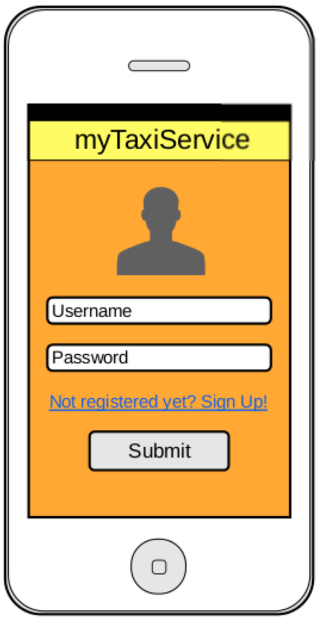
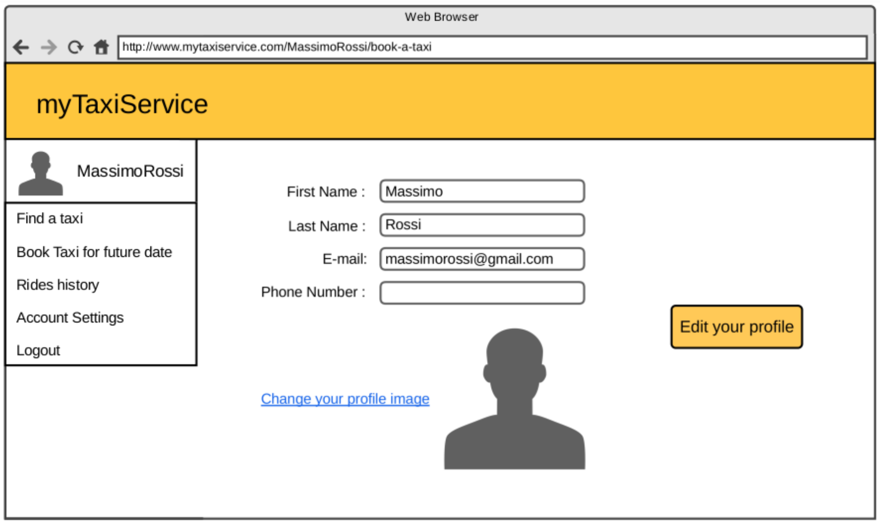


Figure 4

3.1.1.3 Passenger's homepage Figure 5 and Figure 6 shows respectively the passenger's homepage once he/she has logged in the web application or in the mobile application.


Web Browser
http://www.mytaxiservice.com/MassimoRossi/book-a-taxi
myTaxiService
MassimoRossi
Find a taxi
Book Taxi for future date
Rides history
Account Settings
Logout
Change your profile image
Edit your profile

First Name : Massimo
Last Name : Rossi
E-mail: massimorossi@gmail.com
Phone Number :

Figure 5

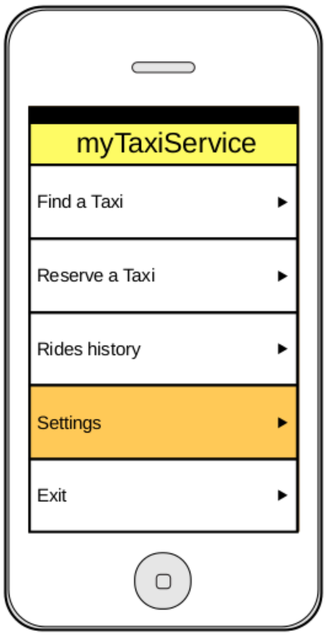


Figure 6

3.1.1.4 Taxi driver's menu Figure 7 shows the taxi driver's menu once he/she has selected the option in the mobile application.

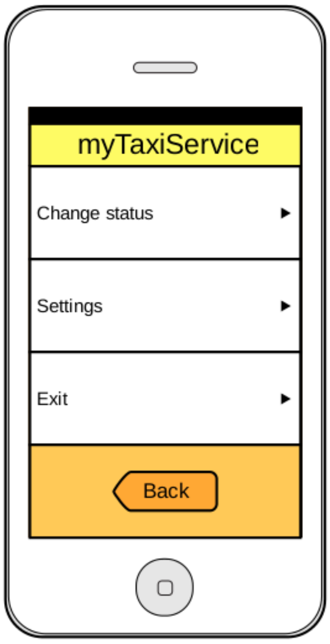


Figure 7

3.1.1.5 Find a taxi Figure 8 and Figure 9 shows respectively the request form in the web application and in the mobile application.

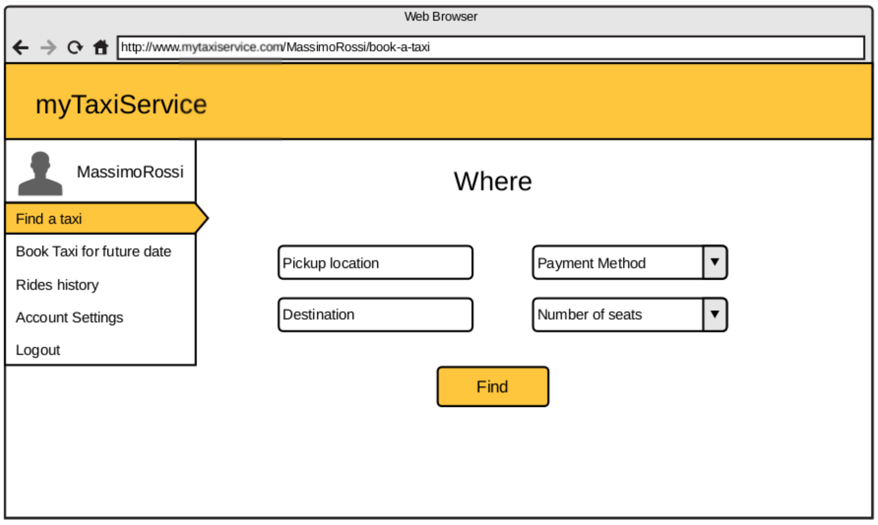


Figure 8

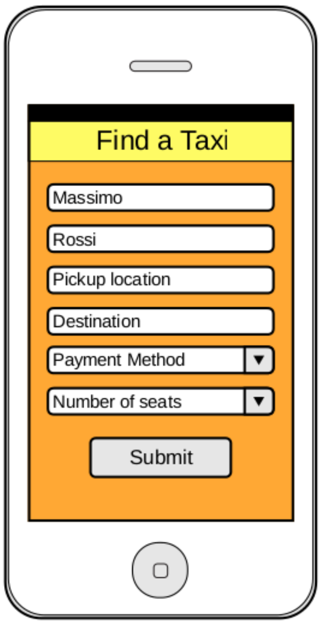


Figure 9

3.1.1.6 Reserve a taxi Figure 10 adn Figure 11 shows respectively the request form in the web application and in the mobile application.

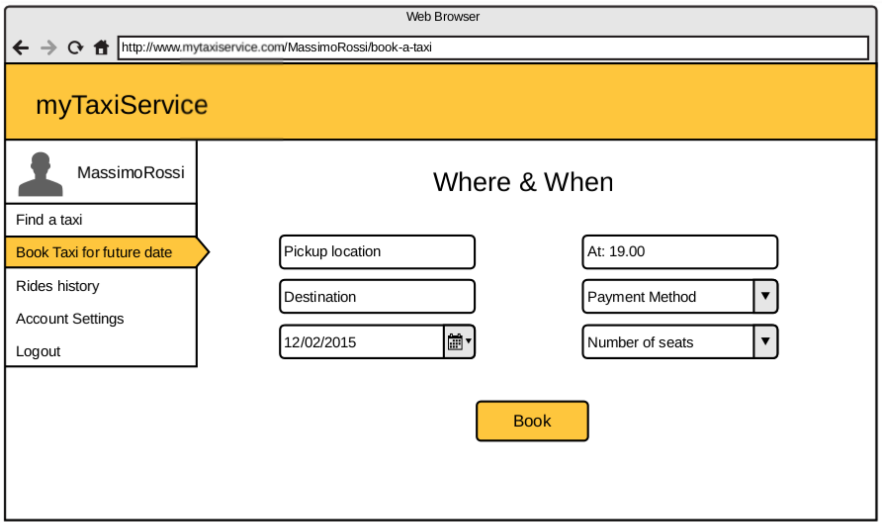


Figure 10

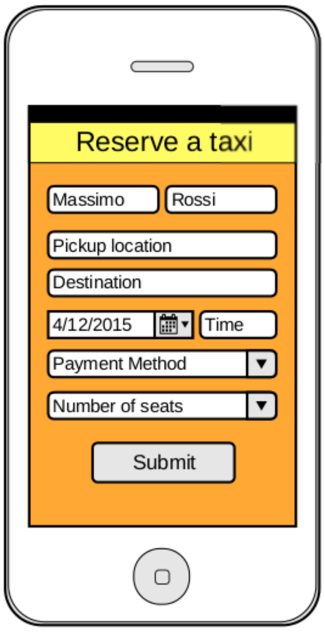
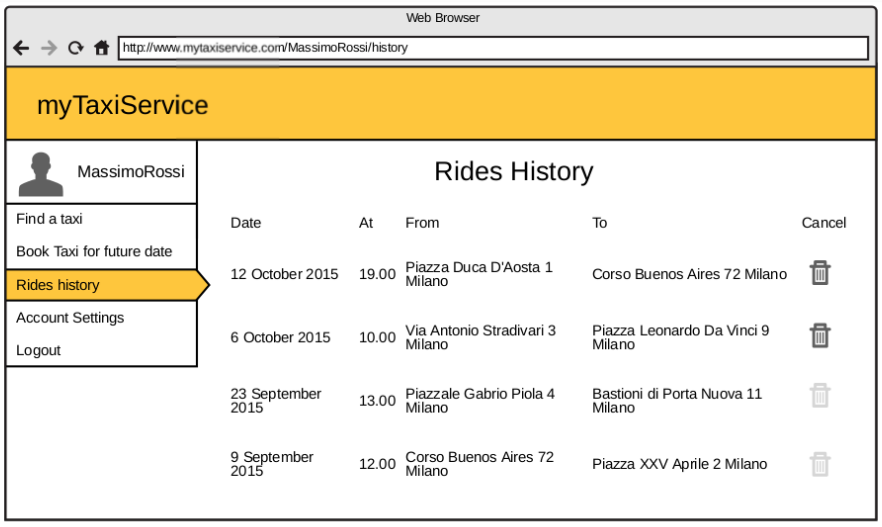


Figure 11

3.1.1.7 Rides history Figure 12 and Figure 13 shows respectively the details of previous reservation booked by the passenger via web application and mobile application.



Date	At	From	To	Cancel
12 October 2015	19.00	Piazza Duca D'Aosta 1 Milano	Corso Buenos Aires 72 Milano	
6 October 2015	10.00	Via Antonio Stradivari 3 Milano	Piazza Leonardo Da Vinci 9 Milano	
23 September 2015	13.00	Piazzale Gabrio Piola 4 Milano	Bastioni di Porta Nuova 11 Milano	
9 September 2015	12.00	Corso Buenos Aires 72 Milano	Piazza XXV Aprile 2 Milano	

Figure 12

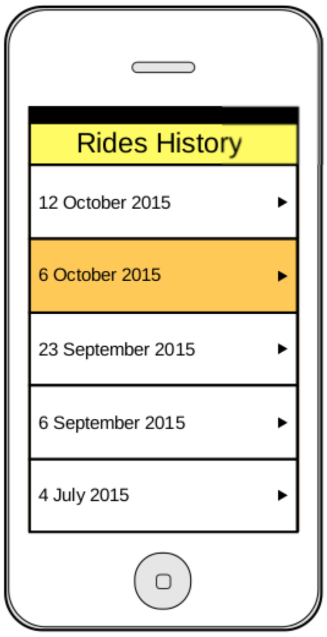
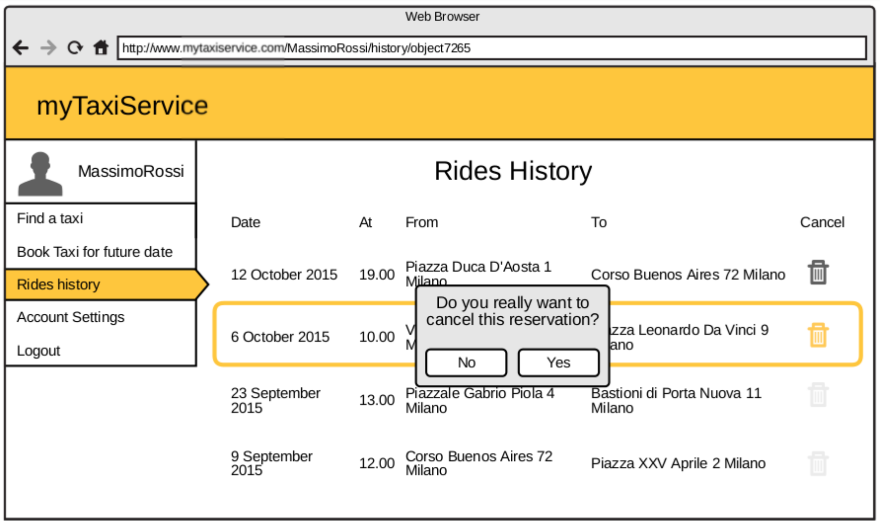


Figure 13

3.1.1.8 Cancel a reservation Figure 14 and Figure 15 shows respectively the screen activated when the user decide to delete a reservation via web application and mobile application. In the web application 🗑 means that it is possible to delete this order. 🗑 indicates the selected order that is going to be delated. 🗑 means that is not possible to delate this order (See requirements).



Web Browser

http://www.mytaxiservice.com/MassimoRossi/history/object7265

myTaxiService

MassimoRossi

- Find a taxi
- Book Taxi for future date
- Rides history
- Account Settings
- Logout

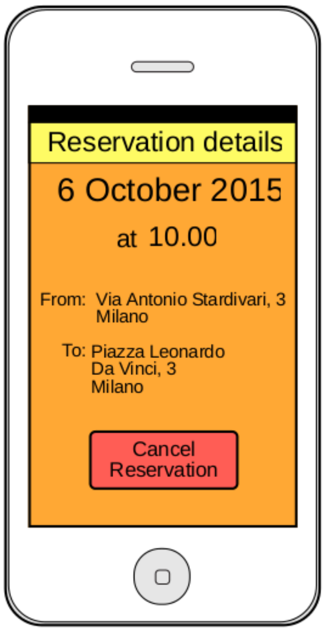
Rides History

Date	At	From	To	Cancel
12 October 2015	19.00	Piazza Duca D'Aosta 1 Milano	Corso Buenos Aires 72 Milano	🗑
6 October 2015	10.00	V… M…	…zza Leonardo Da Vinci 9 …ano	🗑
23 September 2015	13.00	Piazzale Gabrio Piola 4 Milano	Bastioni di Porta Nuova 11 Milano	🗑
9 September 2015	12.00	Corso Buenos Aires 72 Milano	Piazza XXV Aprile 2 Milano	🗑

Do you really want to cancel this reservation?

No Yes

Figure 14



Reservation details

6 October 2015

at 10.00

From: Via Antonio Stardivari, 3 Milano

To: Piazza Leonardo Da Vinci, 3 Milano

Cancel Reservation

Figure 15

3.1.1.9 Incoming request Figure 16 shows the mockup of an incoming request on the smartphone of the taxi driver.

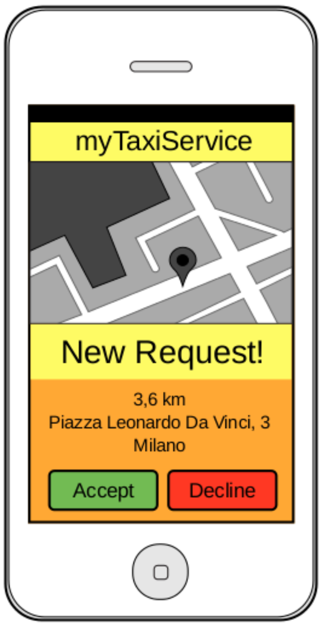


Figure 16

3.1.1.10 Change Status Figure 17 shows the mockup of the change status screen in the taxi driver's mobile application.

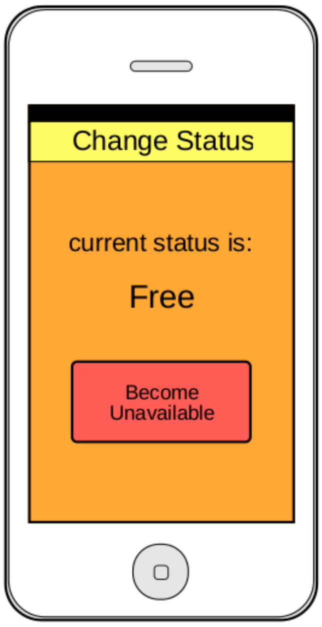
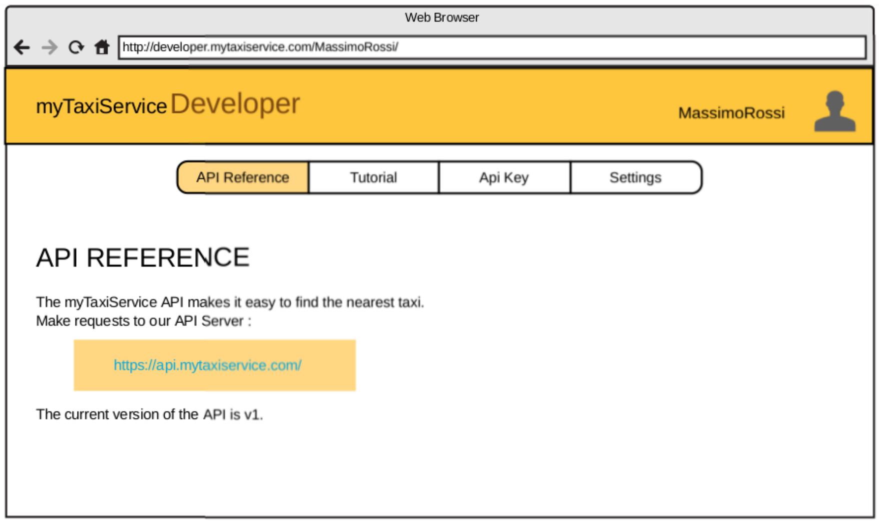


Figure 17

3.1.1.11 Developer's homepage Figure 18 shows the mockup of the homepage of the developer once he/she is logged in the system.



API REFERENCE

The myTaxiService API makes it easy to find the nearest taxi.
Make requests to our API Server :

https://api.mytaxiservice.com/

The current version of the API is v1.

Figure 18

3.2 Functional Requirements

3.2.1 [G1] Keep trace of users of the application

[FR1] The system should provide a "Sign Up" function so that every visitor can become a registered user.

[FR2] The system should provide a "Login" function that allows registered users to interact with the application.

[FR3] The system should not allow a visitor to input a username and/or an e-mail that is already in use.

[FR4] The system has to check that a user does not sign up twice.

[FR5] The system has to keep trace of all the orders (immediate requests and reservations) done by passengers and show them when required.

3.2.2 [G2] Answer to passengers' requests within an acceptable time

[FR6] The system should allow passengers to request a taxi for the current date and time.

[FR7] The system should allow passengers to reserve a taxi for a desired date and time.

[FR8] The system should allow passengers to accept or decline the proposed solution .

[FR9] The system should display to passengers the ETA and code of the selected taxi.

[FR10] The system should allow passengers to cancel a reservation within 15 minutes before the pickup time.

[FR11] The system should not allow reservation where the date is previous to the current one and/or the pickup time occurs within 2 hours from the current time.

3.2.3 [G3] Grant a correct management of taxi queues

[FR12] The system should divide the city into zones of $2km^2$ and keep a queue for each of them.

[FR13] The system should constantly receive taxi's position (via GPS) and status to keep trace of the zone they are into.

[FR14] The system should maintain the queue up to date, adding free taxi drivers and removing the unavailable ones.

[FR15] When a request from a passenger arrives, the system should forward it to the first taxi driver in the queue.

3.2.4 [G4] Provide an efficient communication between taxi drivers and the system

[FR16] The system should notify taxi drivers of all incoming request.

[FR17] The system should allow taxi drivers to change status from free to unavailable and vice versa.

[FR18] The system automatically changes the status of a taxi driver when he/she has accepted a request and the associated passenger has also accepted the proposal.

[FR19] If an unexpected event occurs to the taxi driver while going to the pickup place, the system should notify the passenger and contact another taxi driver.

3.2.5 [G5] Allow developers to extend the application by adding new functionalities

[FR20] The system should provide an API that allow users to make a call to the API Server which will return all taxi that are available in the zone containing the specified location.

[FR21] The system should provide an API that allow users to make a call to the API Server, which will return the ETA of the first available taxi.

[FR22] The system should provide an API key when the developer request it.

[FR23] The system should provide the API documentation to the developers.

3.3 Scenarios

3.3.0.1 Scenario 1 Giovanni has just arrived in Milan after a long travel from Naples, because he wants to see the concert of his favourite singer at San Siro Stadium. He needs to take a taxi to reach the stadium, as it is far from the Central Station in which he arrived. So he opens the myTaxiService application on his smartphone and activates the "Find a taxi" function. He writes his name, his current position (specifying in the notes that he is on the left side of the station), his destination, and indicates one passenger and credit card as payment method, because he isn't travelling with a lot of cash money. Then sends the request. The system forwards the request to the first taxi available in that zone, and receives a positive answer. After a few seconds, the application shows the first taxi available to Giovanni: taxi number 63, waiting time (ETA) 3 minutes. Giovanni, surprised by the fastness of the answer, accepts the taxi happily and starts waiting.

3.3.0.2 Scenario 2 Mario starts his work shift at 8.00 AM. He wake up, have breakfast and get into his car, as he always do in the morning. But that time the car refuses to start. Mario, desperate, decide to call a Taxi. He fills in the form request and wait for answer. The system get the request and check for an available taxi in his zone, unluckily there is none. The system then sends the request to the closest zone, and once a taxi driver has accepted the ride sends a notification to Mario reporting the code of the taxi and an ETA of 15 minutes. In the same moment, Mario's neighbour passes by and offers Mario a lift, so he happily accept and decline the taxi requested. The system immediately notify the taxi driver that the ride has been cancelled.

3.3.0.3 Scenario 3 Lorenzo, the driver of taxi 63, is waiting for a call near the Central Station. Suddenly the myTaxiService application on his smartphone sends him a notification: there's a request from the Central Station. He reads all the information and rapidly accepts the ride. The system indicates to the passenger the information about Lorenzo's taxi, and after receiving a positive response signals to Lorenzo that the passenger has accepted. So the taxi driver starts his car and drives to the station.

3.3.0.4 Scenario 4 Luca, the taxi driver of 54, is free but he's talking with a friend outside his taxi. He has forgetten his phone in the cab and, right in that moment a request arrive from the system. Luca doesn't hear the call so, after 1 minutes of waiting, the system puts Luca at the bottom of the current queue and forward the request to the next taxi driver.

3.3.0.5 Scenario 5
Pietro and Angela are going to the graduation party of their friend Marco tonight, which will take place in a disco in down town Milan. They are going by the metropolitan at 8 p.m., but knowing that the party is going to finish late, they decide to take a taxi for the return. As Pietro is a registered user of myTaxiService, he decides to reserve a taxi. He logs in the web application on his laptop and chooses the “Book a taxi for future date” function. He indicates the address of the party as origin place, his home’s address as destination place, 5 people as number of passengers (because 3 of his friends are going to stay at his home after the party) and selects 1 a.m. as time of reservation. The system checks that the time of reservation is at least two hours after the current time and saves the reservation in the database, indicating that a minivan will be needed for the ride. At this point sends a confirmation to Pietro.

3.3.0.6 Scenario 6
Angelo is travelling from Padova to Milan for a meeting. His arrival is expected at 9.55 AM. Knowing that trains are always a bit late, Angelo booked a taxi for 10.15 outside the Central Station. Unluckily the train that morning suffers a breakdown. The train conductor announces that it will take 90 minutes until the train is able to resume its journey. Angelo is forced to cancel his reservation, so he access the app via mobile and delete it. The system checks if the cancellation has been done in time (no more then 15’ to the ride), and confirms it. No taxi is then allocated for that reservation.

3.3.0.7 Scenario 7
Lorenzo has successfully ended his ride from Central Station to San Siro Stadium. Giovanni, the passenger, pays his ride with credit card and leaves. At this point Lorenzo communicate to the system that the passenger correctly paid the amount and he is now free. The system then puts Lorenzo at the bottom of the queue corresponding to the zone in which he is now.

3.3.0.8 Scenario 8
Massimo has just ended his work shift, so before going home he enters the myTaxiSevice app and select the ”Change status” function. The system then removes Massimo from the queue he’s in. The next day, Massimo signal the system that he’s back to work changing his status from Unavailable to Free, immediately the system traces Massimo’s position via GPS, finds that he is in zone2, change his status and allocate him at the bottom of the queue associated to zone2.

3.4 UML Diagrams

3.4.1 Use Cases

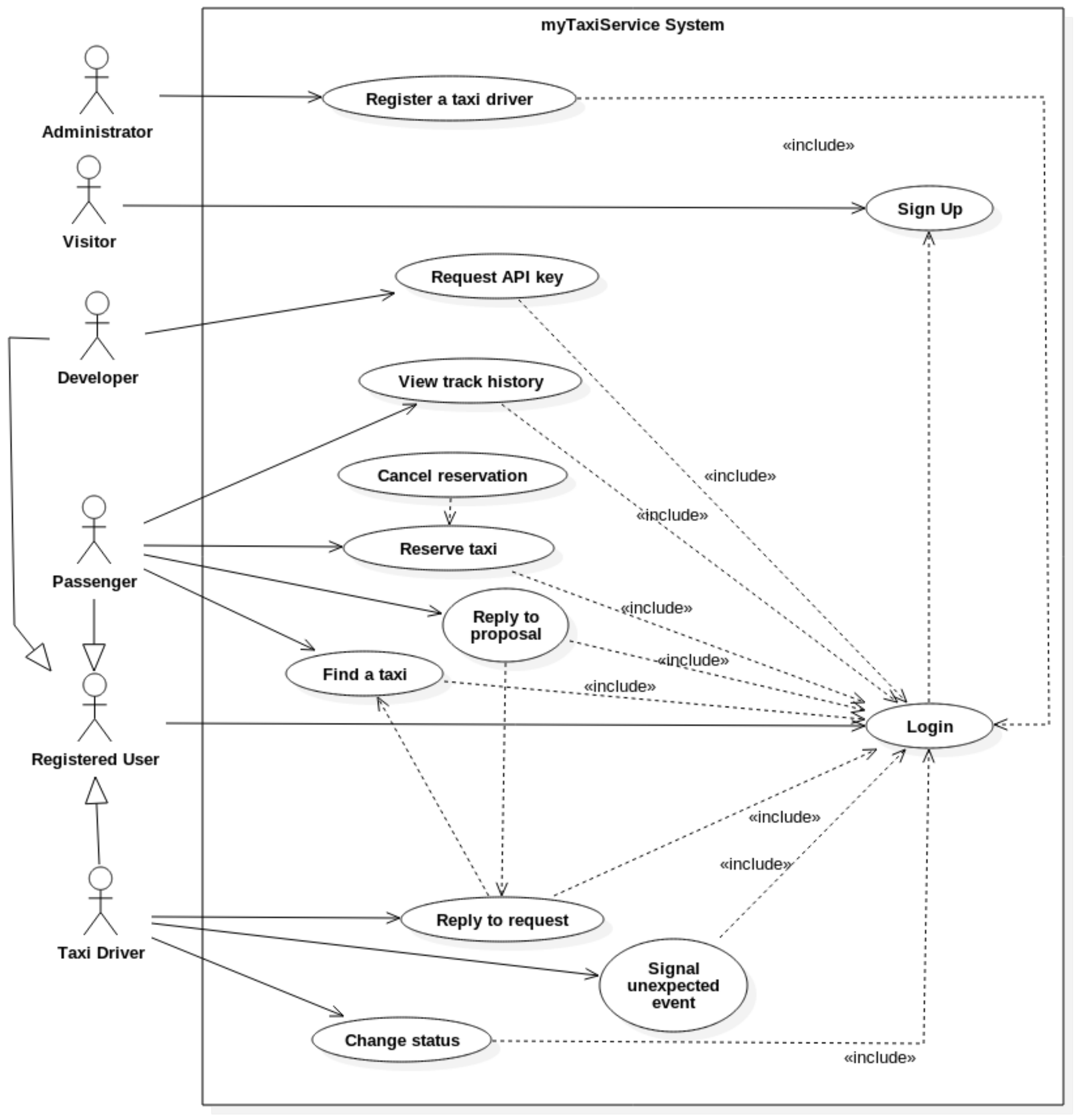
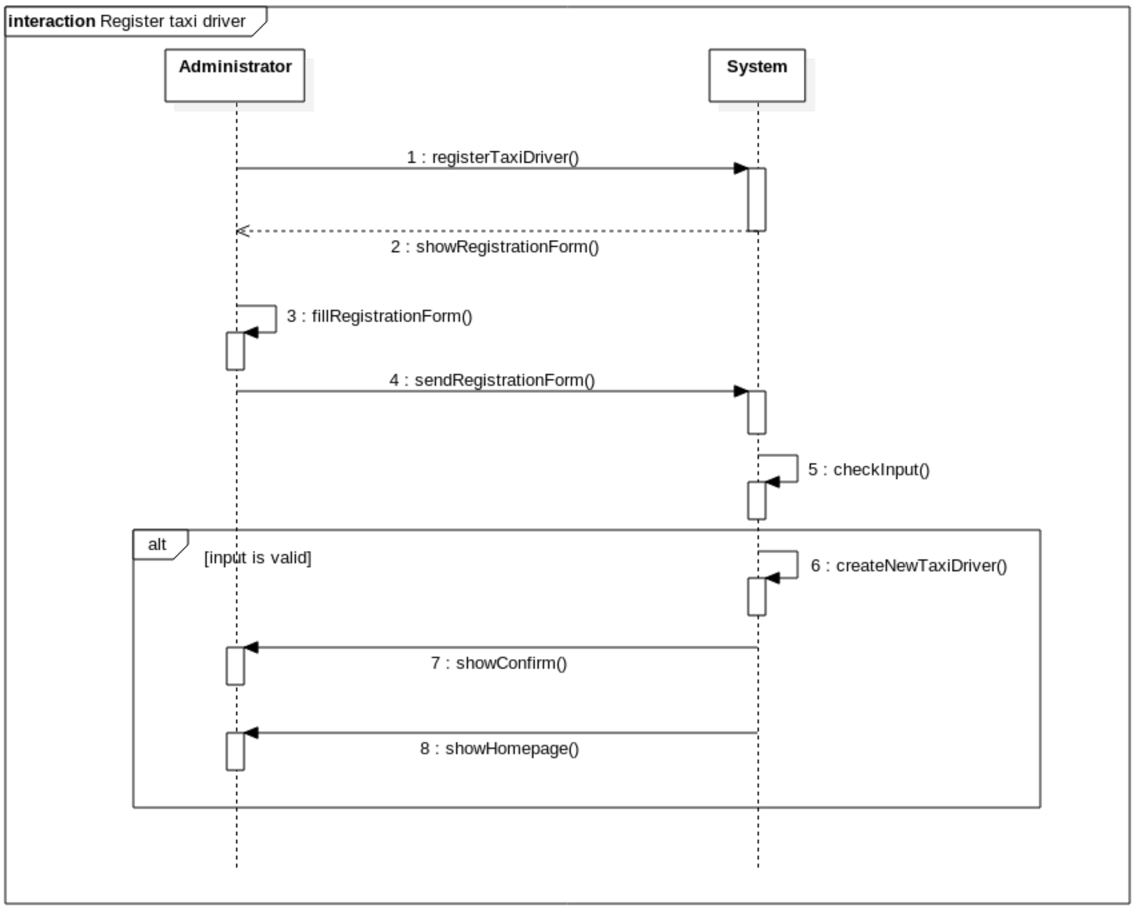


Figure 19: Use Case diagram - Overview

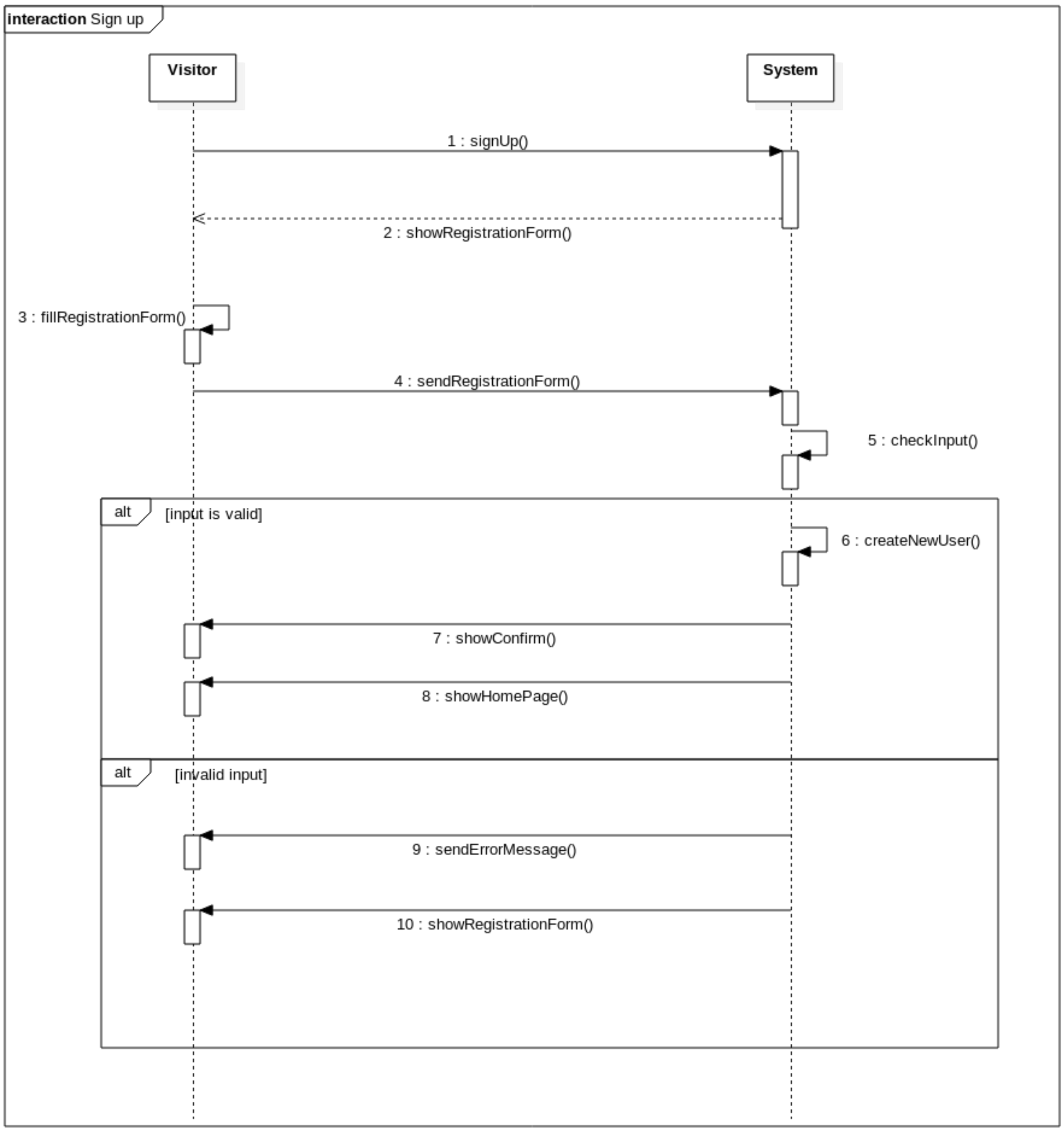
3.4.1.1 Register a taxi driver

Actors	Administrator
Goal	[G1]
Preconditions	1. Administrator is logged in. 2. The taxi driver has been hired by the company.
Flow of events	1. Administrator selects the "Register a taxi driver" function. 2. Administrator inserts taxi driver's information. 3. Administrator submits.
Postconditions	A new taxi driver is added to the company's fleet. His/her information are stored in the DB.
Exceptions	If taxi driver's email or license code already exists in the DB the system shows an error message to the Administrator reporting the causes of the abort.


interaction Register taxi driver
Administrator
System
1 : registerTaxiDriver()
2 : showRegistrationForm()
3 : fillRegistrationForm()
4 : sendRegistrationForm()
5 : checkInput()
alt
[input is valid]
6 : createNewTaxiDriver()
7 : showConfirm()
8 : showHomepage()

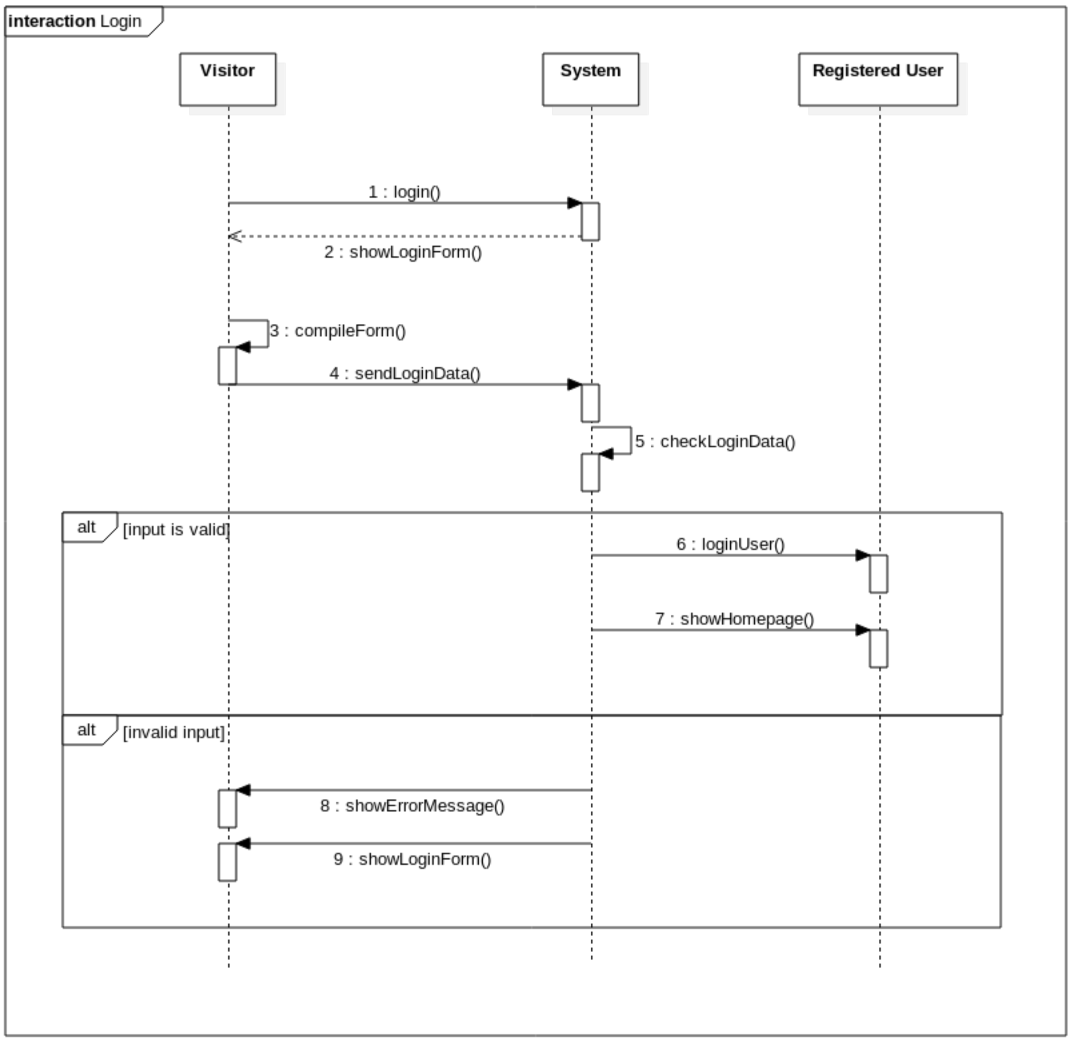
3.4.1.2 Sign Up

Actors	Visitor
Goal	[G1]
Preconditions	
Flow of events	1. Visitor accesses the "Sign Up" function via web application or mobile application. 2. Visitor enters his first name, last name, street address, number, ZIP code, country, email. 3. Visitor selects a username and a password. 4. Visitor submits. 5. Vistor's data are saved in the DB.
Postconditions	Visitor is successfully registered. Now he/she can log in the system.
Exceptions	1. Visitor is already a user. 2. Visitor chooses a username already possessed by an other user. 3. Visitor inserts an e-mail that is already associated with another user. In all this cases the system displays an error message and the visitor is redirected to the registration form.


interaction Sign up
Visitor
System
1 : signUp()
2 : showRegistrationForm()
3 : fillRegistrationForm()
4 : sendRegistrationForm()
5 : checkInput()
alt
[input is valid]
6 : createNewUser()
7 : showConfirm()
8 : showHomePage()
alt
[invalid input]
9 : sendErrorMessage()
10 : showRegistrationForm()

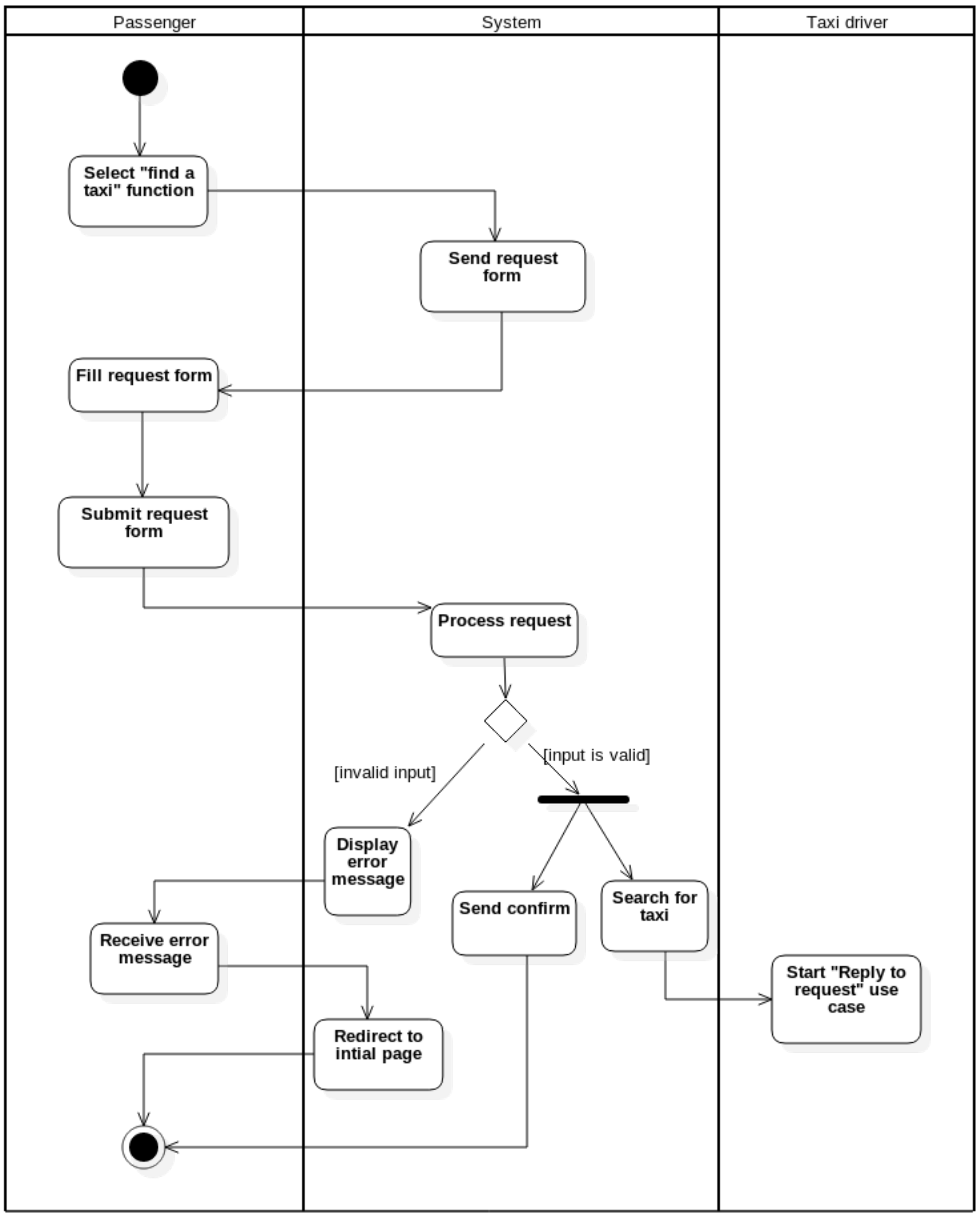
3.4.1.3 Login

Actors	Visitor, Registered User
Goal	[G1]
Preconditions	User should be registered
Flow of events	1. Passenger accesses the ”Login” function via web application or mobile application. Taxi driver can access only via mobile application. 2. User enters username and password. 3. User submits.
Postconditions	User is successfully logged in the system.
Exceptions	1. User inserts wrong credentials. 2. Visitor is not registered. In all this cases the system displays an error message and the visitor is redirected to the login page.


interaction Login
Visitor
System
Registered User
1 : login()
2 : showLoginForm()
3 : compileForm()
4 : sendLoginData()
5 : checkLoginData()
alt
[input is valid]
6 : loginUser()
7 : showHomepage()
alt
[invalid input]
8 : showErrorMessage()
9 : showLoginForm()

3.4.1.4 Find a taxi

Actors	Passenger
Goal	[G2]
Preconditions	Passenger should be logged in.
Flow of events	1. Passenger accesses the "Find a taxi" function via web application or mobile application. 2. Passenger fills in the form with pickup location, destination, number of seats needed, payment method. 3. Passenger submits. 4. The system selects a taxi driver according to his algorithm and forwards the request to him/her.
Postconditions	User's request has been sent to the system that is processing it.
Exceptions	If passenger inserts a non existing pickup location or destination the system displays an error message and the user is redirected to the "Find a taxi" page.

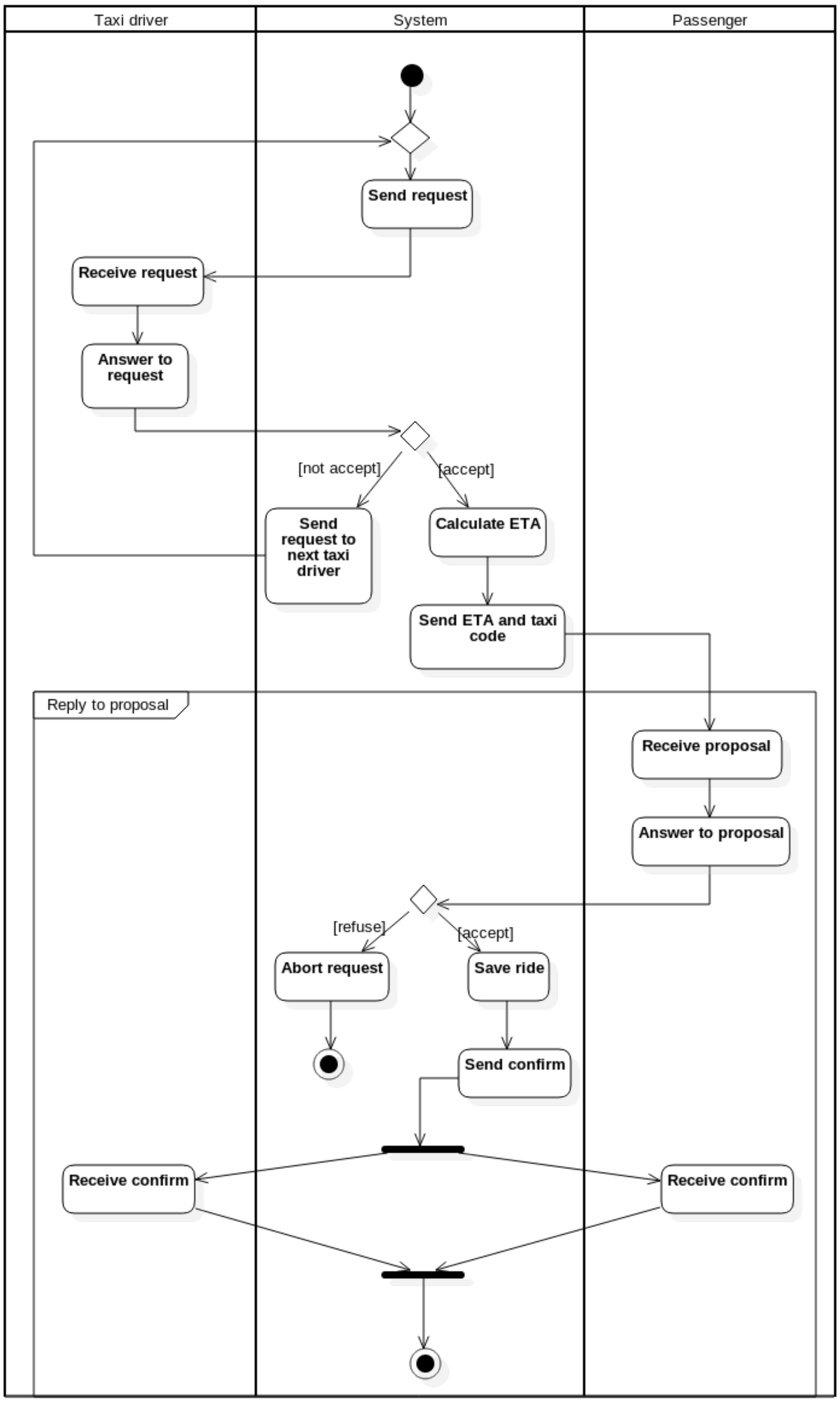

Passenger
System
Taxi driver
Select "find a taxi" function
Send request form
Fill request form
Submit request form
Process request
[invalid input]
[input is valid]
Display error message
Send confirm
Search for taxi
Receive error message
Start "Reply to request" use case
Redirect to intial page

3.4.1.5 Reply to request

Actors	Taxi driver
Goal	[G2]
Preconditions	1. Taxi driver is logged in. 2. Passenger has done a request. 3. The system has forwarded passenger's request to the taxi driver.
Flow of events	1. Taxi driver accepts the request. 2. The system elaborates the ETA and let the passenger know it (and also the code of the incoming taxi). 3. If passenger accepts the proposal the taxi driver moves towards the pickup location.
Postconditions	The system changes the status of the taxi driver from free to unavailable and removes him/her from the queue.
Exceptions	1. Taxi driver refuses the request. 2. Taxi driver doesn't reply to the request within 1 minutes. 3. Passenger refuses the proposal In all this cases the system doesn't change the status of the taxi driver (that remain free) and doesn't move him/her from the current queue he/she is into.

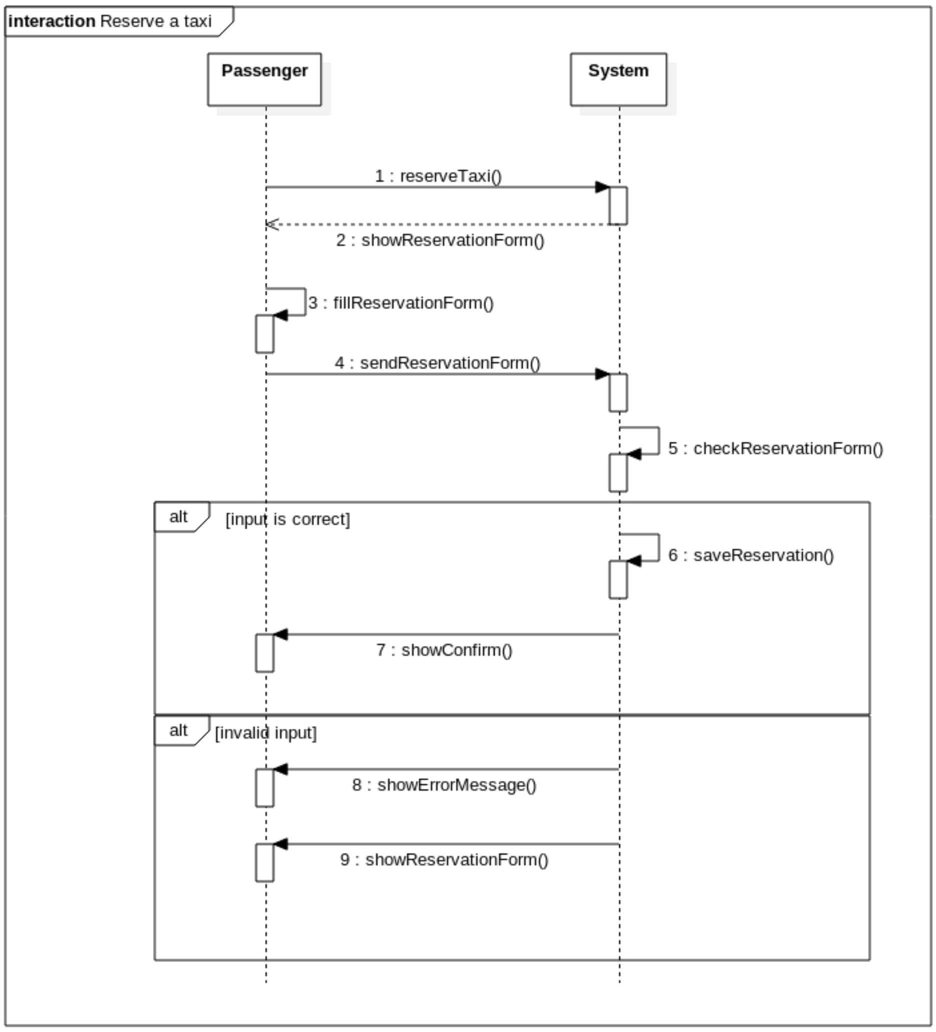
3.4.1.6 Reply to proposal

Actors	Passenger
Goal	[G2]
Preconditions	1. Passenger is logged in. 2. Passenger has performed a request. 3. Taxi driver has accepted the request.
Flow of events	1. Passenger accepts the request.
Postconditions	Passenger's ride is added to his/her rides history. Taxi driver is acknowledged that the proposal has been accepted
Exceptions	If passenger refuses the proposal the system : - doesn't change the status of the taxi driver (that remains free) and doesn't move him/her from the current queue he/she is into. - doesn't add the ride to the user's rides history.


Taxi driver
System
Passenger
Send request
Receive request
Answer to request
[not accept]
[accept]
Send request to next taxi driver
Calculate ETA
Send ETA and taxi code
Reply to proposal
Receive proposal
Answer to proposal
[refuse]
[accept]
Abort request
Save ride
Send confirm
Receive confirm
Receive confirm

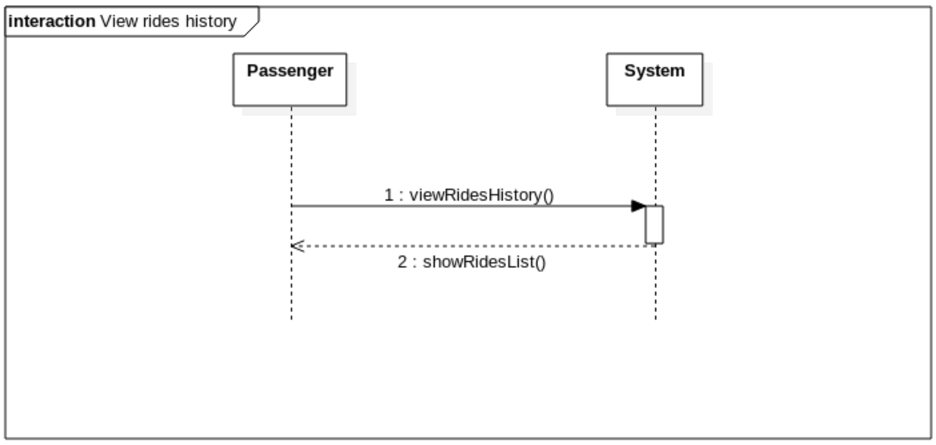
3.4.1.7 Reserve taxi

Actors	Passenger
Goal	[G2]
Preconditions	Passenger should be logged in.
Flow of events	1. Passenger accesses the "Book a taxi for future date" function via web application or the "Reserve a Taxi" via mobile application. 2. Passenger fills in the form with pickup location, destination, pickup date and time, number of seats needed, payment method. 3. Passenger submits. 4. The system stores the order in the DB. 5. Passenger is acknowledged that his/her reservation has gone fine.
Postconditions	Passenger's reservation is successfully stored in the DB.
Exceptions	1. Passenger selects a date that belongs to the past. 2. Passenger inserts a non existing pickup location or a destination. 3. Passenger inserts a pickup time that occurs within 2 hours from the current time. In all this cases the system pops up an error message with the relative information. The user is redirected to the request form.


interaction Reserve a taxi
Passenger
System
1 : reserveTaxi()
2 : showReservationForm()
3 : fillReservationForm()
4 : sendReservationForm()
5 : checkReservationForm()
alt
[input is correct]
6 : saveReservation()
7 : showConfirm()
alt
[invalid input]
8 : showErrorMessage()
9 : showReservationForm()

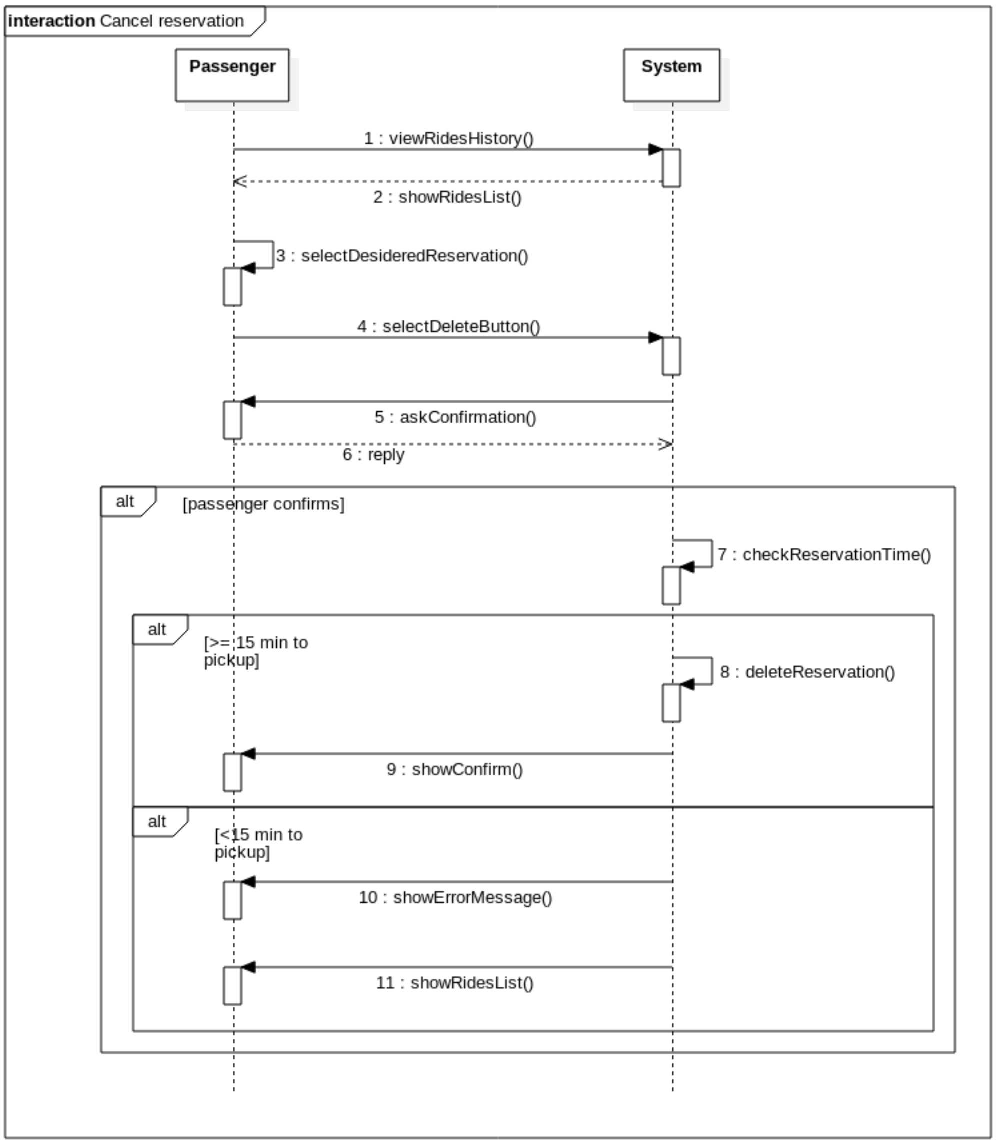
3.4.1.8 View rides history

Actors	Passenger
Goal	[G2]
Preconditions	Passenger is logged in the system.
Flow of events	1. Passenger selects ”Rides history” function.
Postconditions	The system shows on the passenger’s screen the complete list of rides (already made, then belonging to the past) and his/her reservation (future possible rides).
Exceptions	



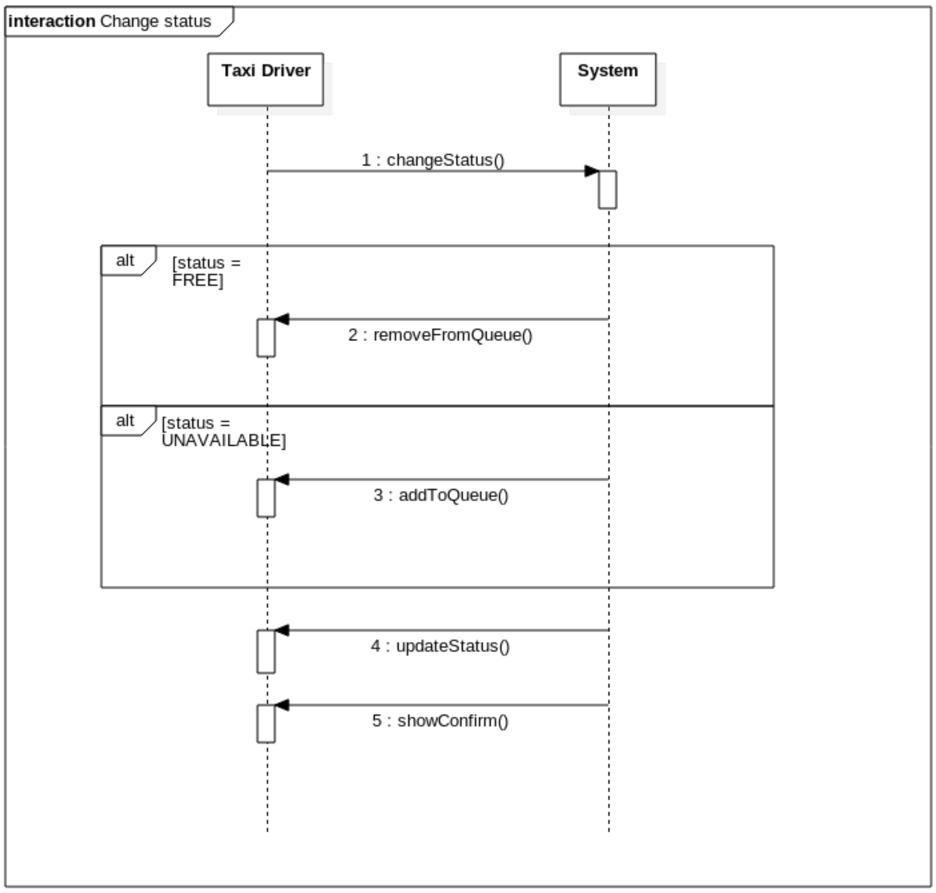
3.4.1.9 Cancel reservation

Actors	Passenger
Goal	[G2]
Preconditions	1. Passenger should be logged in. 2. Passenger should be in the "Rides history" page. 3. Passenger should have done a reservation that occurs in the future. 4. There has to be no less then 15 minutes left to the pick-up time.
Flow of events	1. Passenger selects the desired reservation and click the delete button. 2. Passenger confirms. 3. The system deletes the order from the DB.
Postconditions	Passenger's reservation is successfully removed from the DB.
Exceptions	If the passenger selects a reservation whose pickup time occurs within 15 minutes from the current time, the system pops up an error message with the relative information. The user is redirected to the "Rides history" page.


interaction Cancel reservation
Passenger
System
1 : viewRidesHistory()
2 : showRidesList()
3 : selectDesideredReservation()
4 : selectDeleteButton()
5 : askConfirmation()
6 : reply
alt
[passenger confirms]
7 : checkReservationTime()
alt
[>= 15 min to pickup]
8 : deleteReservation()
9 : showConfirm()
alt
[<15 min to pickup]
10 : showErrorMessage()
11 : showRidesList()

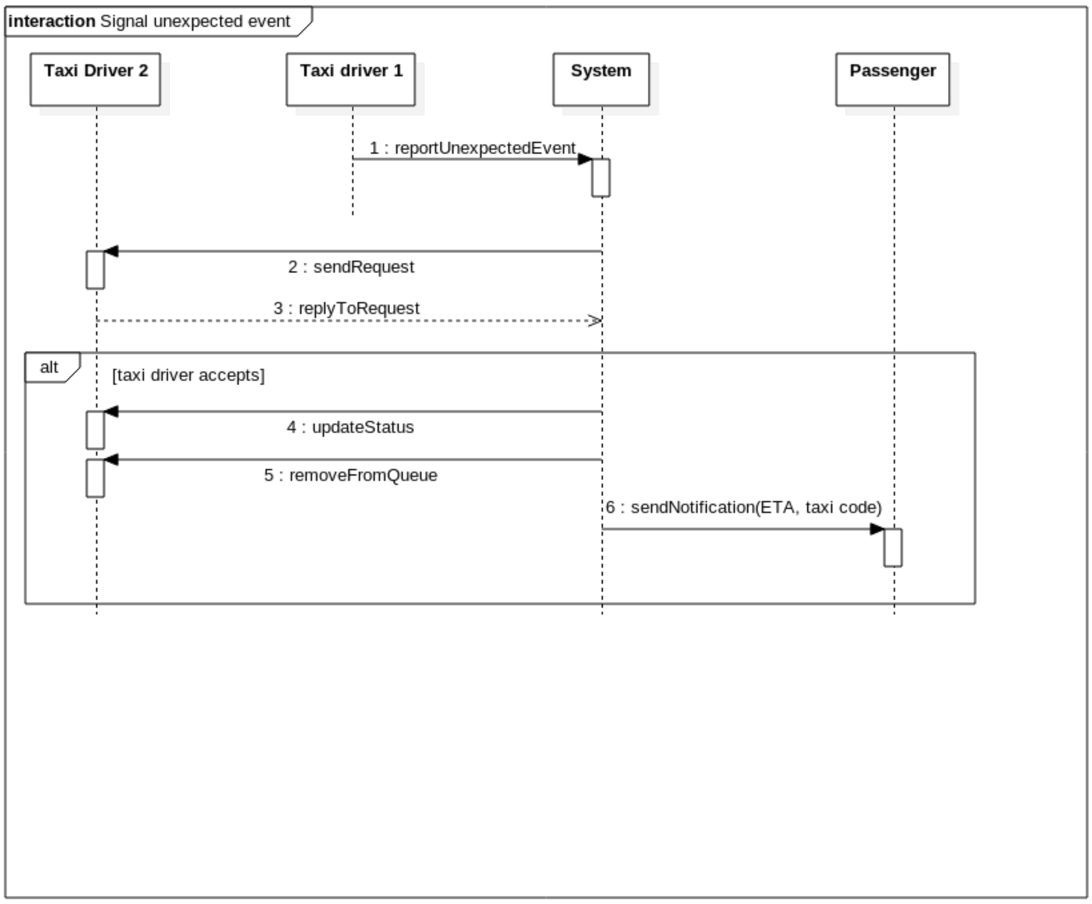
3.4.1.10 Change status

Actors	Taxi driver
Goal	[G4]
Preconditions	Taxi driver is logged in.
Flow of events	1. Taxi driver selects the ”Change status” function. 2. The system changes the status of the taxi driver and moves it into the queue according to the policy.
Postconditions	Taxi driver status has been changed correctly.
Exceptions	



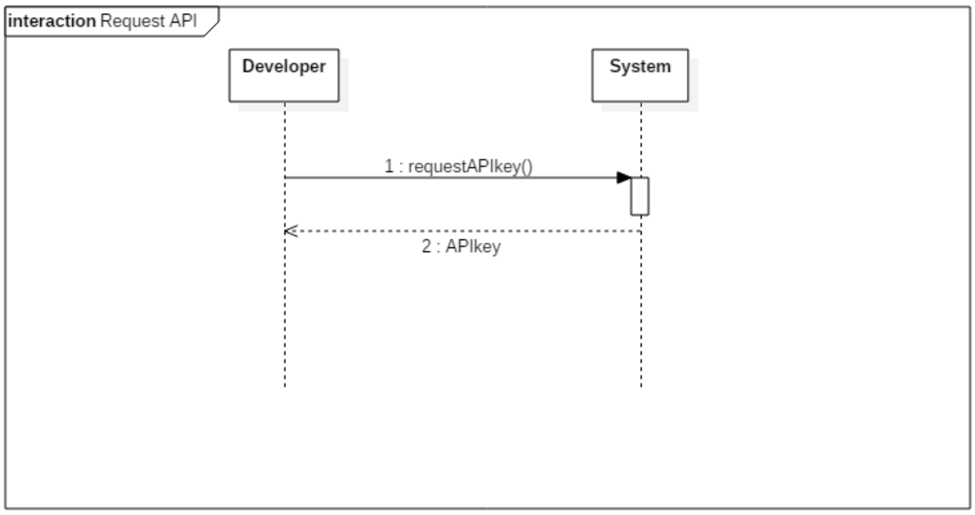
3.4.1.11 Signal unexpected event

Actors	Taxi driver
Goal	[G4]
Preconditions	Taxi driver is logged in the system.
Flow of events	1. When an unexpected event occurs, taxi driver selects the "Report unexpected event" function. 2. The system contacts an other taxi driver. 3. The system notifies the passenger that an unexpected event has occurred and that a new taxi driver is available (ETA and taxi code are displayed).
Postconditions	The unexpected event is reported correctly and the user has been notified about that.
Exceptions	



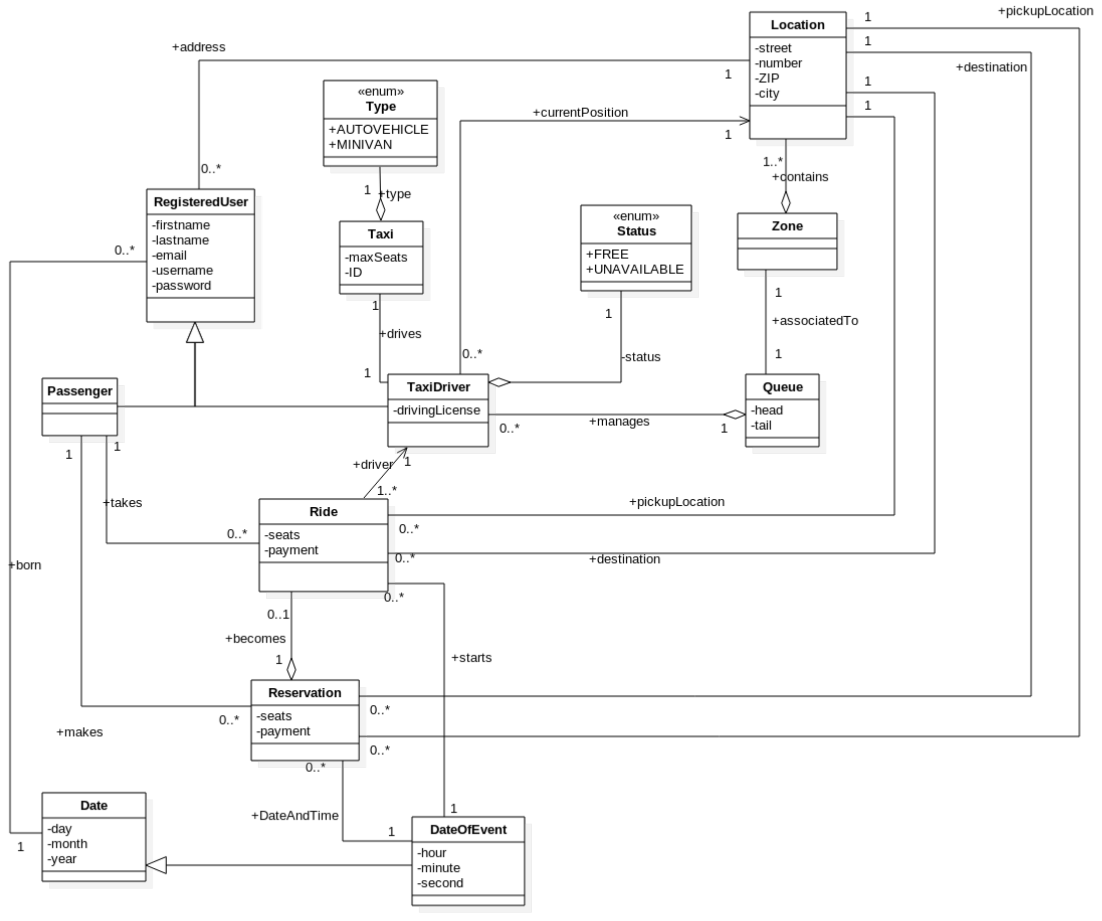
3.4.1.12 Request API

Actors	Developer
Goal	[G5]
Preconditions	Developer is logged in.
Flow of events	1. Developer selects the "Request API key" function. 2. The system generates a new API key and shows it on the screen.
Postconditions	The API key is visualized on the screen so that the developer can save it, and the system stores it in the database.
Exceptions	If the developer has already requested an API key, the system shows an error message preventing the creation of two keys for the same user.



3.4.2 Class Diagram

The following image represent the class diagram that supply a simplified vison of the entities that plays a major role in the application.



3.5 Performance Requirements

[NFR1] Performance must be acceptable to guarantee a good grade of usability, both for passengers and for taxi drivers.

[NFR2] The system has to answer to a passenger's request within a mean time of 30 seconds, assuming that there are no internet connection problems and that the first taxi driver in the queue of the current zone accepts the call within 25 seconds.

[NFR3] Users can access each function of the application within a maximum of 3 screens, so that he/she can rapidly complete every requests.

3.6 Design Constraints

[NFR4] The application will be developed with Java EE so it will inherit all language's constraints.

3.7 Software System Attributes

3.7.1 Reliability

[NFR5] The system will be based on a relational database, so it must keep all its data always accessible and consistent, even in case of unexpected problems or failures, like electric network failures or natural disasters.

[NFR6] The system must allow to plan periodical backups, possibly once per day (the actual frequency will be defined later on in the project).

[NFR7] In case of serious failures, it could be necessary a restart of the system; this is allowed, as the system is not a critical application.

3.7.2 Availability

[NFR8] The system has to be accessible in every moment, since it is intended for large cities so it's assumed that a lot of taxi requests are made every day.

[NFR9] In case of server failures, the availability of the service will be granted by a backup server that will substitute temporarily the main server.

[NFR10] The system can be temporarily unavailable only for a maximum of one hour per week, for ordinary maintenance.

3.7.3 Security

3.7.3.1 Interface Side

[NFR11] The system will implement an authentication process to identify users and protect their information.

[NFR12] Users' passwords will be stored safely in the Database of the system. Requirements on the strength of the password will not be too heavy: minimum length of 6, with no particular need of numbers, capital letter or special characters.

[NFR13] Passwords will also be static, meaning that users will not be asked to change their password after some time.

[NFR14] In order to receive response from the API Server the user must provide his/her API key in his/her request.

[NFR15] API users cannot exceed 250 requests per day. HTTP responses will return a 429 status code for any request until the rate limit has dropped below the required threshold.

3.7.3.2 Application Side

[NFR16] On the application side, the system will implement the https connection instead of http to guarantee communication confidentiality between users and the server and integrity of data sent in the network.

3.7.3.3 Server Side

[NFR17] In order to improve security, application logic will be divided from the database logic, so the software architecture will have an application server separated from the Database and from the web server.

[NFR18] All components will be divided by a firewall, in order to allow access to each zone only to authorized users and block possible malicious attacks.

3.7.4 Maintainability

[NFR19] The system will provide wide code documentation, in order to let future developers know how the application works and find possible weaknesses or problems.

[NFR20] Developers will be able also to request API of the application in order to not only develop additional features, but also correct bugs and keep the application up-to-date.

3.7.5 Portability

[NFR21] The application will be usable on any operative system which supports JVM and DBMS, as it will be written in java code.

3.8 Other Requirements

3.8.1 Accessibility

[NFR22] The system must be easily usable by every type of user, so no particular knowledge will be required.

[NFR23] All types of users must be able to access every function of the application without reading any instruction, anyway instructions will be available for those who request them.

4 Appendix

4.1 Alloy

4.1.1 .als file

```
/******************************
                    DATA TYPE
******************************/
abstract sig Status{}
one sig Free extends Status{}
one sig Unavailable extends Status{}

sig integer{}
sig string{}

sig Location{
        street: one string ,
        number: one integer,
        ZIP: one integer,
        city: one string,
}


sig Date{
        day: one integer,
        month: one integer,
        year: one integer
}

sig DateOfEvent extends Date{
        time: one Time,
}

sig Time{
        hour: one integer,
        minute: one integer,
        second: one integer
}

abstract sig Type{}
one sig Autovehicle extends Type{}
one sig Minivan extends Type{}

/*********************************
                       ENTITIES
```

```
*********************************/
abstract sig  RegisteredUser{
        firstname: one string,
        lastname: one string,
        email: one string,
        address: one Location,
        username: one string,
        password: one string
}

sig Passenger extends RegisteredUser{
        history: set Ride,
        reservations: set Reservation,
}

sig TaxiDriver extends RegisteredUser{
        taxi: one Taxi,
        current_position: one Location,
        driving_license: one string,
        status: one Status
}

sig Taxi{
        ID: one integer,
        seats: one integer,
        type: one Type
}

sig Queue{
        head : lone TaxiDriver,
        tail: lone TaxiDriver,
        taxis: set TaxiDriver,
        zone: one Zone
}{#taxis≥0 }

sig Zone{
        locations: set Location,
        queue : one Queue
}{#locations>0}

sig Ride{
        passenger: one Passenger,
        driver: one TaxiDriver,
        pickupLocation: one Location,
        destination: one Location,
        startDateTime : one DateOfEvent,
        seats: one integer,
        payment: one string
}

sig Reservation{
        ride: lone Ride,
        passenger: one Passenger,
        pickupLocation: one Location,
        destination: one Location,
        reservationDateTime: one DateOfEvent,
        seats: one integer,
        payment: one string
}

/*********************************
                       FUNCTIONS
**********************************/

fun getTaxiDriver[q:Queue]: one TaxiDriver {
        (q.taxis -q.head -q.tail)
}


/*********************************
                           FACTS
**********************************/
fact EntitiesExistance{
        #Zone>1
}
//No duplicate user
fact NoDuplicateUsers{
        no disj u,u1:RegisteredUser | (u.username=u1.
           ↪ username) or (u.email = u1.email)
}

//No taxi drivers with the same driving license
fact NoDuplicateLicense{
        no disj t,t1:TaxiDriver | (t.driving_license=t1.
           ↪ driving_license)
}

//No taxis with the same ID
fact NoDuplicateTaxiID{
        no disj t,t1 : Taxi | t.ID=t1.ID
}
```

```
//No taxi driver with the same taxi
fact NoTaxiDriverWithSameTaxi{
        no disj t,t1:TaxiDriver | t.taxi = t1.taxi
}

//If one zone contains one queue, then this queue is
   ↪ contained in that zone
fact ZonesQueuesAssociationIsUnique{
        all q:Queue, z:Zone | z=q.zone iff q=z.queue
}

//One location belongs to exactely one zone
fact ZoneStructure{
        no disj z,z1:Zone |
        (some loc:Location |
        (loc in z.locations) and (loc in z1.locations))
}

//All locations must belong to a zone
fact OneLocationOneZone{
        all loc:Location | one z:Zone | loc in z.
           ↪ locations
}

//No rides with the same date and time(hour,minute)
   ↪ coming from the same passenger
fact NoDifferentRides{
        no disj r,r1: Ride | (r.passenger=r1.passenger)
           ↪ and (r.startDateTime=r1.startDateTime)
}

//No taxi driver in different queue
fact NoTaxiDriverInDifferentQueue{
        no t:TaxiDriver, q,q1:Queue | (t in q.taxis) and
           ↪  (t in q1.taxis) and(q ≠ q1)
}

//Pickup location and destination in a ride and in a
   ↪ reservation should be different
fact PickupLocationDifferentFromDestination{
        all r:Ride | (r.pickupLocation ≠ r.destination)
    all r:Reservation | (r.pickupLocation ≠ r.
       ↪ destination)
}

//Unavailable taxi driver does not appear in any queue
```

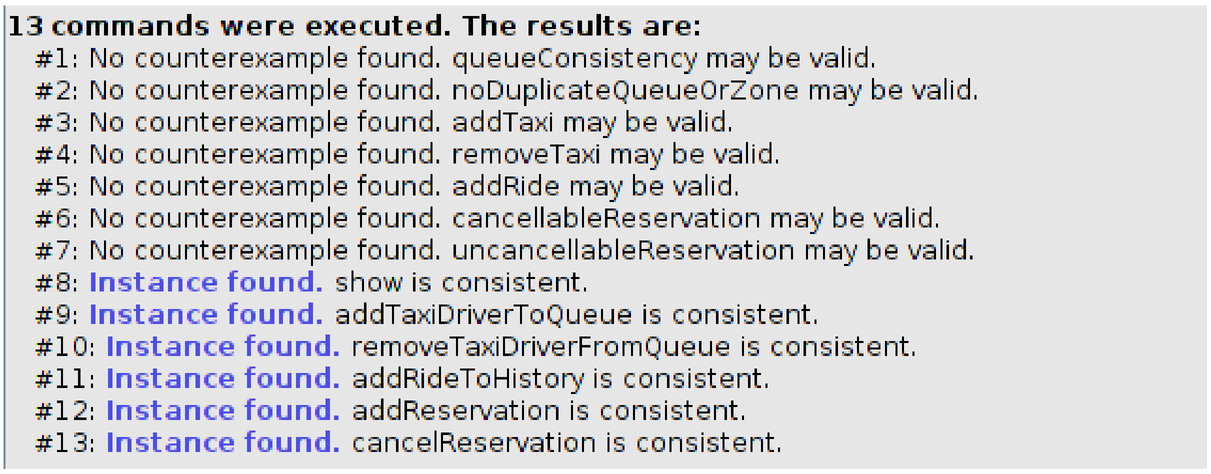
```
fact NoUnavailableTaxiInQueues{
        all q:Queue, t:TaxiDriver | (t in (q.taxis)) ⟹
           ↪ (t.status=Free)
}

//If a taxi's location belongs to a certain zone, then
   ↪ the taxi (if is available) must appear
//in the queue of that zone
fact TaxiDriverLocation{
        all t:TaxiDriver, z:Zone | (t.status=Free) and (
           ↪ t.current_position in z.locations) ⟹ (t
           ↪ in z.queue.taxis)
}

//If the queue is empty there is no head nor tail
fact EmptyQueue{
        all q:Queue | (#q.taxis=0) implies ((#q.head=0)
           ↪ and (#q.tail=0))
}

//If the queue has one element head=tail
fact OneElementinQueue{
        all q:Queue |  (#q.taxis=1) implies ((q.head=q.
           ↪ tail) and (#q.head=1))
}

//If the queue has more then one element head!=tail
fact MoreElementinQueue{
        all q:Queue |  (#q.taxis>1) implies ((q.head≠q.
           ↪ tail) and (#q.head=1) and (#q.tail=1))
}

//Head and Tail are taxi that appears in the queue
fact TaxiDriverExistanceInQueue{
        all q:Queue | (#q.taxis>0) implies ((q.head in q
           ↪ .taxis) and (q.tail in q.taxis))
}

//All rides appear only in the history of their
   ↪ passenger
fact HistoryConstraint{
        all p:Passenger, r:Ride | (p=r.passenger) iff (r
           ↪ in p.history)
}

//If a Reservation is not cancelled, it becomes a ride.
```

```
//That ride should have the same attributes of
   ↪ Reservation except for the time.
fact ReservationToRide{
        all res:Reservation | (#res.ride=1) implies ((
           ↪ res.passenger=res.ride.passenger)
                and(res.pickupLocation=res.ride.
                   ↪ pickupLocation)and(res.destination
                   ↪  = res.ride.destination)
                and(res.reservationDateTime.day = res.
                   ↪ ride.startDateTime.day)
                and (res.reservationDateTime.month = res
                   ↪ .ride.startDateTime.month)
                and(res.reservationDateTime.year = res.
                   ↪ ride.startDateTime.year)
                and (res.seats = res.ride.seats) and(res
                   ↪ .payment = res.ride.payment))
}

//All reservations belong to only one passenger
fact NoReservationWithDifferentPassenger{
        all p:Passenger, r:Reservation | (p=r.passenger)
           ↪  iff (r in p.reservations)
}

//there are no reservation that refer to the same ride
fact NoReservationWithSameRide{
        no disj r,r1:Reservation | r.ride = r1.ride
}

/**********************************
                        ASSERTIONS
***********************************/

//If a queue  has at least one element, its head and
   ↪ tail must be a taxi between those elements
assert queueConsistency{
        all q:Queue |  (#q.taxis>0) implies ((#q.head=1)
           ↪  and (#q.tail=1) and
                (q.head in q.taxis) and (q.tail in q.
                   ↪ taxis))
}
check queueConsistency

//No zones with the same queue and no queues with the
   ↪ same zones
assert noDuplicateQueueOrZone{
        no disj z,z1: Zone | z.queue = z1.queue
        no  disj q,q1: Queue | q.zone = q1.zone
}
check noDuplicateQueueOrZone


assert addTaxi{
    //if a taxi is not in a queue, it can be added to it
       ↪  and it becomes the new tail of that queue
        all q,q1:Queue, t:TaxiDriver |
        (!(t in q.taxis) and addTaxiDriverToQueue[q,q1,t
           ↪ ] and #q.taxis>0) implies ((#q1.taxis = #q
           ↪ .taxis +1) and (q1.head = q.head) and (q1.
           ↪ tail = t))

    //if a taxi is added to a queue that has no taxis in
       ↪  it, both the head and the tail of that queue
       ↪ are updated
        all q,q1:Queue, t:TaxiDriver |
        (!(t in q.taxis)and addTaxiDriverToQueue[q,q1,t]
           ↪  and #q.taxis=0) implies ((#q1.taxis = #q.
           ↪ taxis +1) and (q1.head = t) and (q1.tail =
           ↪  t))
}
check addTaxi


assert removeTaxi{
        //if a taxi is not in a queue, it can't be
           ↪ removed from it
        all q,q1:Queue, t:TaxiDriver |
        !(t in q.taxis) and removeTaxiDriverFromQueue[q,
           ↪ q1,t]  implies (#q1.taxis = #q.taxis)

        //if a taxi is in a queue, it can be removed
           ↪ from it
        all q,q1:Queue, t:TaxiDriver |
        (t in q.taxis) and removeTaxiDriverFromQueue[q,
           ↪ q1,t]  implies (#q1.taxis = #q.taxis -1)

        //if the head of a queue is removed from it, the
           ↪  head is updated and the tail remains the
           ↪ same
        all q,q1:Queue, t:TaxiDriver | (
           ↪ removeTaxiDriverFromQueue[q,q1,t]and (q.
           ↪ head=t and #q.taxis>2))
        implies ((q1.head in q.taxis) and !(q1.head = q.
          ↪ head)and (q1.tail = q.tail))
}
check removeTaxi

assert addRide{
        //a new ride can be added to a passenger's
          ↪ history if it is not already in it
        all p,p1:Passenger, r:Ride |
        (!(r in p.history) and addRideToHistory[p,p1,r])
          ↪  implies (#p1.history = #p.history +1)

        //if a ride is already in a queue, it can't be
          ↪ added again
        all p,p1:Passenger, r:Ride |
        ((r in p.history) and addRideToHistory[p,p1,r])
          ↪ implies (#p1.history = #p.history)

}
check addRide

//if a reservation has not already become a ride, it can
  ↪  be cancelled from the system
assert cancellableReservation{
         all p,p1:Passenger, r:Reservation |
         (r in p.reservations and #r.ride=0 and
           ↪ cancelReservation[p,p1,r]) implies ((#p1.
           ↪ reservations=#p.reservations - 1) and !(r
           ↪  in p1.reservations))
}
check cancellableReservation

//if a reservation has already become a ride, it can be
  ↪ cancelled
assert uncancellableReservation{
         all p,p1:Passenger, r:Reservation |
         ((r in p.reservations) and #r.ride=1 and
           ↪ cancelReservation[p,p1,r]) implies ((#p1.
           ↪ reservations=#p.reservations) and (r in
           ↪ p1.reservations))
}
check uncancellableReservation


/**********************************
                        PREDICATES
```

```
**********************************/
pred show{
        #Location >1
        #Zone>1
        #Queue>1
        #Ride>1
        #Reservation>1
        #TaxiDriver>1
        #Passenger>2

}
run show for 5

pred addTaxiDriverToQueue[q,q1:Queue, t:TaxiDriver] {
        !(t in q.taxis) implies (q1.taxis = q.taxis + t)
        (t in q.taxis) implies (q1.taxis = q.taxis)
        //Update head and tail
        (!(t in q.taxis) and #q.taxis>0) implies ((q1.
          ↪ head = q.head) and q1.tail = t)
                else (q1.head = t and q1.tail = t)
}
run addTaxiDriverToQueue for 8

pred removeTaxiDriverFromQueue[q,q1:Queue, t:TaxiDriver]
   ↪ {
        (t in q.taxis) implies (q1.taxis = q.taxis - t)
        !(t in q.taxis) implies (q1.taxis = q.taxis)
        //Update head and tail
        (q.head=t and #q.taxis>2) implies ((q1.head = q.
          ↪ getTaxiDriver) and (q1.tail = q.tail))
        (q.head=t and #q.taxis=2) implies ((q1.head = q.
          ↪ tail) and (q1.tail = q.tail))
        (q.head=t and #q.taxis=1) implies ((#q1.head =
          ↪ 0) and (#q1.tail=0))
        (t in q.taxis and q.head≠t and #q.taxis=2)
          ↪ implies ((q1.head = q.head) and (q1.tail=q
          ↪ .head))
        (t in q.taxis and q.head≠t and q.tail≠t and #q.
          ↪ taxis>2) implies ((q1.head = q.head) and (
          ↪ q1.tail=q.tail))
        (q.tail=t and #q.taxis>2) implies ((q1.head = q.
          ↪ head) and (q1.tail=q.getTaxiDriver))
}
run removeTaxiDriverFromQueue for 8

pred addRideToHistory[p,p1:Passenger, r:Ride]{
        (r in p.history) implies (p1.history=p.history)
        !(r in p.history) implies (p1.history=p.history
          ↪ +r)
}
run addRideToHistory for 8

pred addReservation[p,p1:Passenger, r:Reservation]{
    (r in p.reservations) implies (p1.reservations=p.
      ↪ reservations)
        !(r in p.reservations) implies (p1.reservations=
          ↪ p.reservations +r)
}
run addReservation for 8

pred cancelReservation[p,p1:Passenger, r:Reservation]{
    (r in p.reservations and #r.ride=0) implies (p1.
      ↪ reservations=p.reservations - r)
        //if the reservation doesn't exist or it has
          ↪ already become a ride, it is not cancelled
        (!(r in p.reservations) or #r.ride=1) implies (
          ↪ p1.reservations=p.reservations)
}
run cancelReservation for 8
```

4.1.2 Result

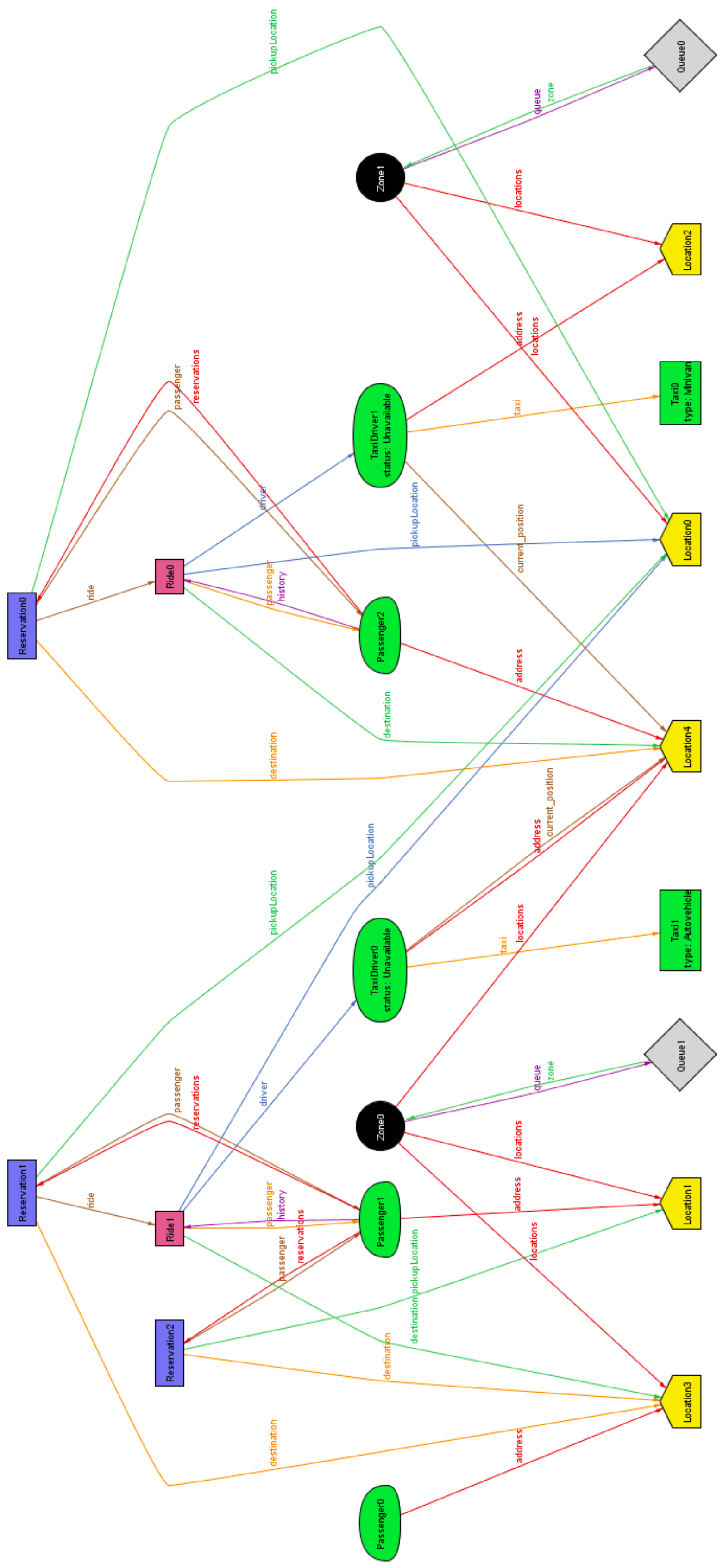
This is the screenshot of Alloy Analyzer 4.2 that shows the consistency of the model.



```
13 commands were executed. The results are:
  #1: No counterexample found. queueConsistency may be valid.
  #2: No counterexample found. noDuplicateQueueOrZone may be valid.
  #3: No counterexample found. addTaxi may be valid.
  #4: No counterexample found. removeTaxi may be valid.
  #5: No counterexample found. addRide may be valid.
  #6: No counterexample found. cancellableReservation may be valid.
  #7: No counterexample found. uncancellableReservation may be valid.
  #8: Instance found. show is consistent.
  #9: Instance found. addTaxiDriverToQueue is consistent.
  #10: Instance found. removeTaxiDriverFromQueue is consistent.
  #11: Instance found. addRideToHistory is consistent.
  #12: Instance found. addReservation is consistent.
  #13: Instance found. cancelReservation is consistent.
```

4.1.3 Generated World

This is the world generated with the given show command.


Reservation0
Reservation1
Reservation2
Ride0
Ride1
Passenger0
Passenger1
Passenger2
TaxiDriver0
status: Unavailable
TaxiDriver1
status: Unavailable
Zone0
Zone1
Queue0
Queue1
Taxi0
type: Minivan
Taxi1
type: Autovehicle
Location0
Location1
Location2
Location3
Location4
ride
passenger
reservations
history
driver
pickupLocation
destination
address
locations
current_position
taxi
queue
zone

4.2 Software Tool used

- TexMaker (http://www.xm1math.net/texmaker/): LaTeXeditor, used to redact this document.
- StarUML (http://staruml.io/) : used to build Use Case diagrams, Sequance diagrams, Class diagrams.
- Alloy Analyzer (http://alloy.mit.edu/alloy/): to check the consistency of our model.
- Moqups (http://www.moqups.com): to create mockups.

4.3 Hours of work

- Alberto Gasparin $\sim$ 32 h
- Vito Matarazzo $\sim$ 32 h