Academic Year 2015/2016

Software Engineering 2 Project

myTaxiService

Andrea Turconi
853589

Lorenzo Raimondi
859001

POLITECNICO
MILANO 1863

Requirements Analysis and Specification Document

Scope and Actors

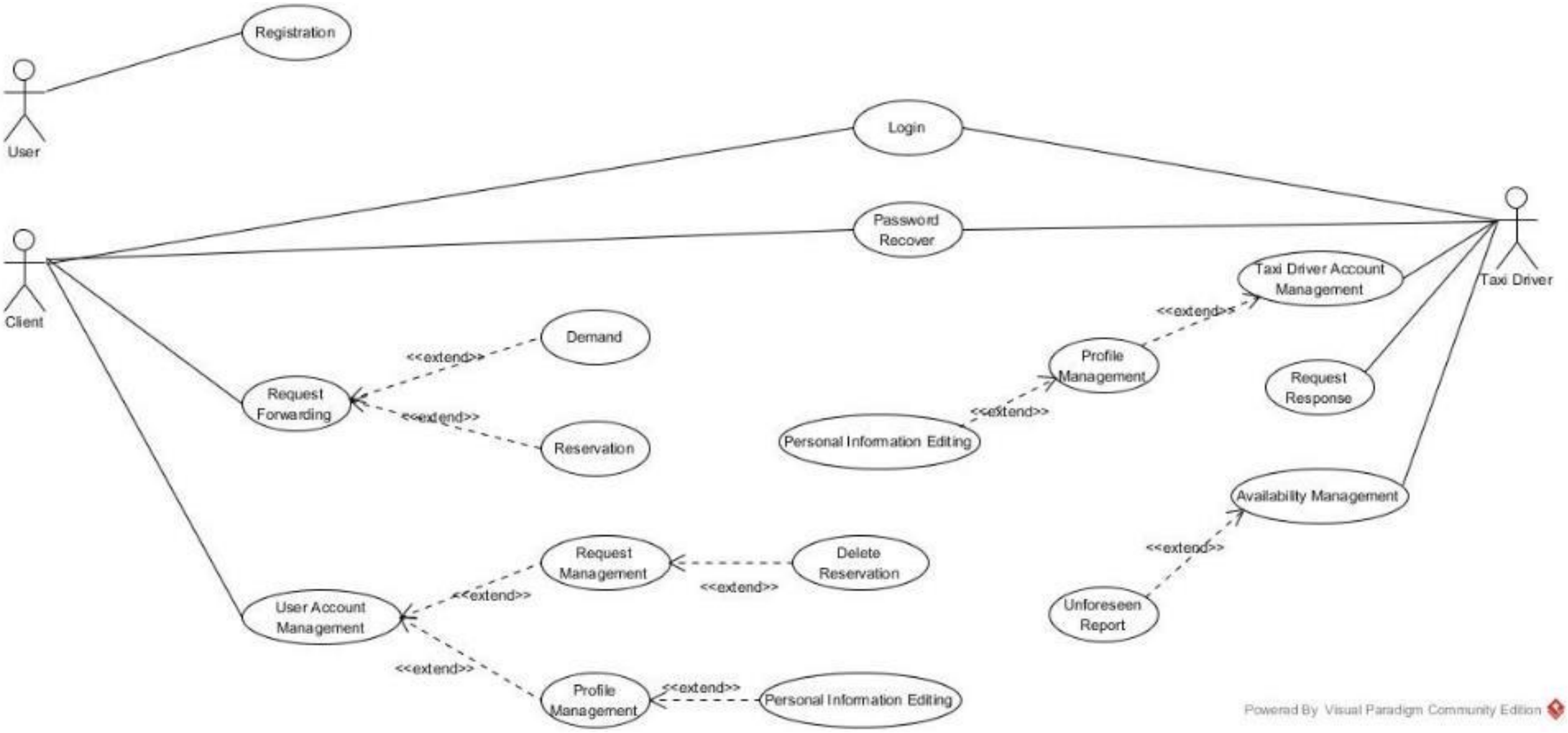
The system that will be developed is both an online software application and a mobile application, called myTaxiService.

The purpose of this product is to help people in calling for a taxi service, simplifying the access of passengers to the service, and to help the taxi drivers to manage their service, guaranteeing a fair management of taxi queues.

Actors:

- Visitors, all the user that doesn’t have an account. They can only view the login page and the registration form, in order to be able to access all the functionality provided by the applications as registered users.
- Registered users: all the users who have an account. They can make requests for taxi service, make reservations, specifying the origin and the destination of the ride, and manage their data.
- Taxi drivers: people who are authorized to practice that profession. They can accept or reject a request and communicate their availability.

Use Case Diagram



Goals and constraints

- [G1] allow a visitor to become a registered user
- [G2] allow a visitor to log in to the application
- [G3] allow user to make a request of taxi service, both reservations and for immediate service
- [G4] allow taxi drivers to communicate their availability
- [G5] allow taxi drivers to accept or reject a request
- [G6] allow registered members to view their reservations and change them.

Principal constraint:

- Taxi's position is computed through GPS localization, which is very sensible to the urban context, giving less accurate or even incorrect data nearby high buildings (e.g. skyscrapers), underground parking or tunnels.

Some Assumptions

- The first and most important assumption we make declares that once a user submits a request to the system, it is guaranteed that a taxi will fulfil his request. In other words, we assume that a user can never be left stranded; a taxi will always give him a shift.
- In addition, users that booked a ride in advance are ensured to get a taxi right at desired place and time. 10 minutes before the designed time the first taxi in the respective zone is appointed to the requesting user, and by means of assumption number one there always be a taxi to be assigned to the booked ride.
- We assume that once the user receives the confirmation notification reporting waiting time and taxi code, he will wait for that taxi, allocated to him.
- When an unforeseen occurs, the taxi driver notifies it to the system. Indeed only if the taxi driver does this the system can reassign the request to another driver and update the information for the user.

…

Example of Functional Requirements

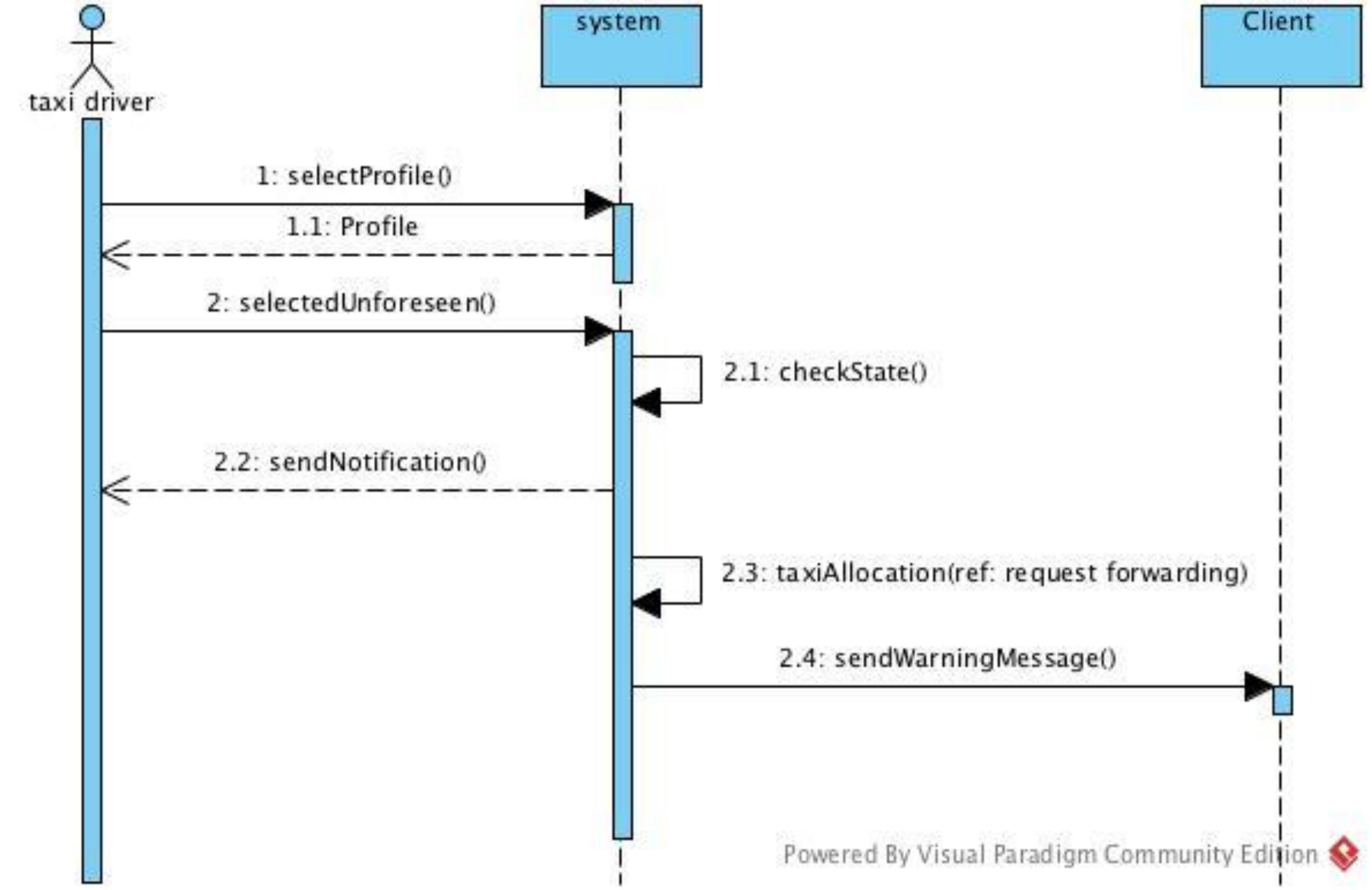
[G3] allow user to make a request of taxi service, both reservations and for demand service

- [R1] user must complete all the mandatory fields of the taxi request form
- [R2] a demand is rejected if there is already another not replied demand, performed by the same user
- [D1] Time must be included between 00.00 and 23.59
- [D2] Starting position and destination provided must exist

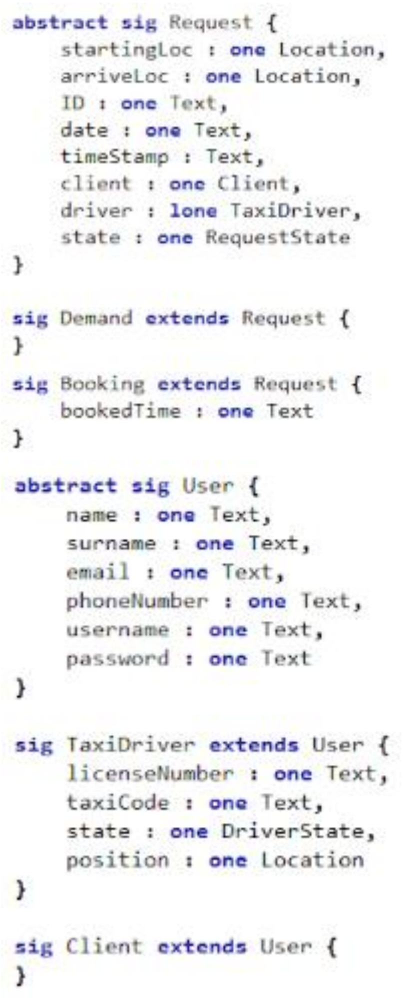
Non Functional Requirements

- **Reliability**, the system must guarantee all the required functionalities under several different situations. So each operation must operate with a transactional model, preventing data loss.
- **Availability**, as long as city taxi service is available 24/7, also myTaxiService aims to be available every day and every night
- **Performance**, it must ensure high performance interaction between user and system
- **Security**, myTaxiService uses an encryption system for preventing data stealing, in particular for data exchanged on networks
- **Portability**, refers to easiness to port the software to other hardware or software systems

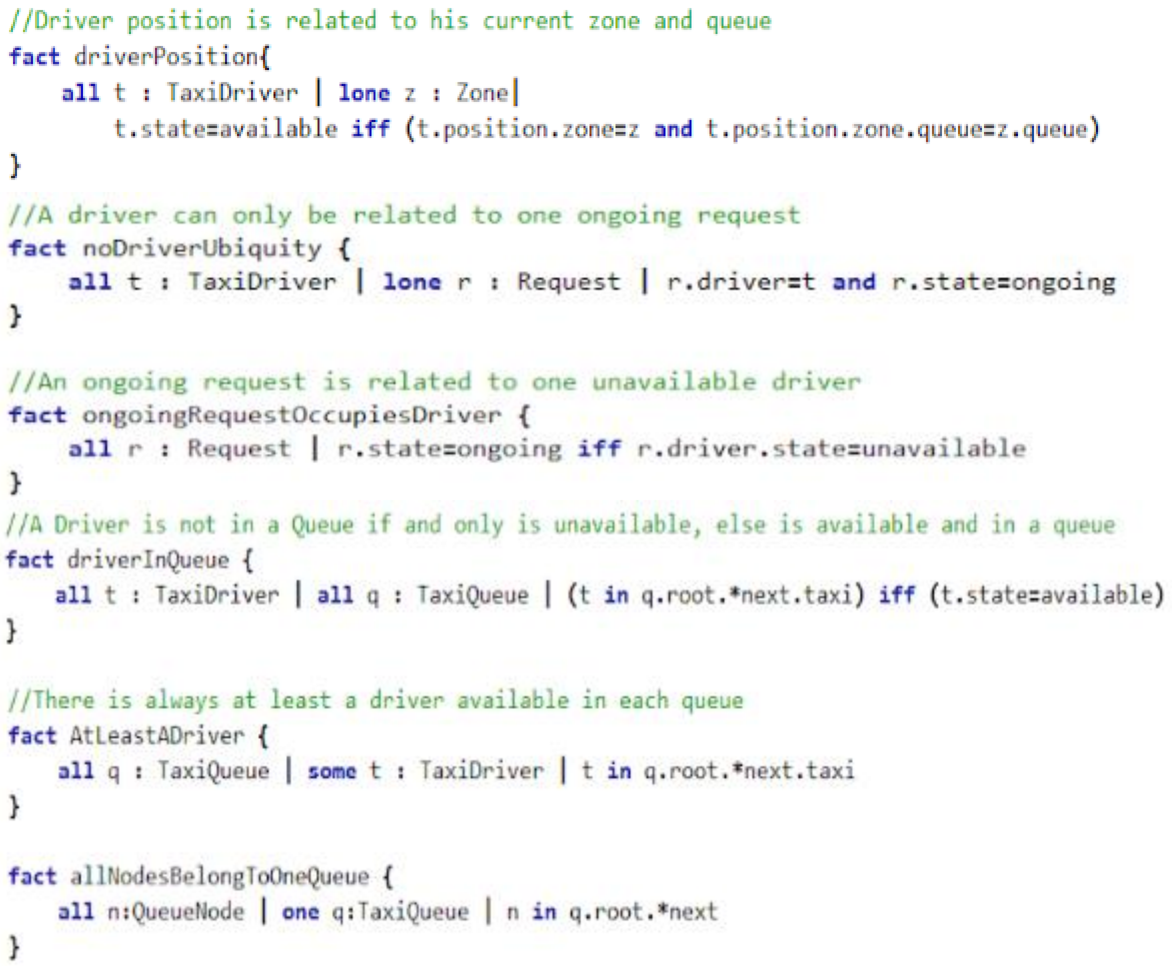
Unforeseen Management



Alloy Code

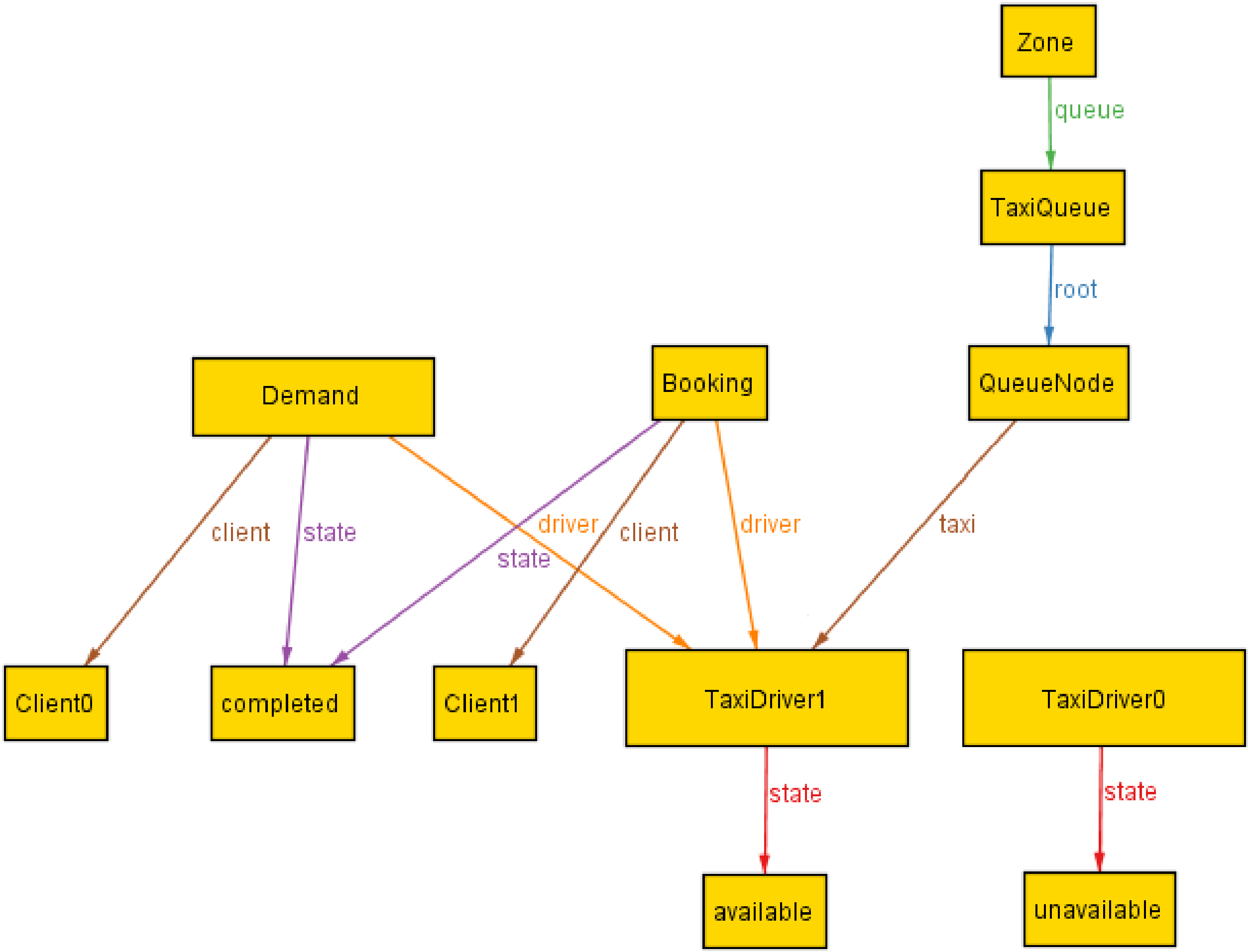


```
abstract sig Request {
    startingLoc : one Location,
    arriveLoc : one Location,
    ID : one Text,
    date : one Text,
    timeStamp : Text,
    client : one Client,
    driver : lone TaxiDriver,
    state : one RequestState
}

sig Demand extends Request {
}

sig Booking extends Request {
    bookedTime : one Text
}

abstract sig User {
    name : one Text,
    surname : one Text,
    email : one Text,
    phoneNumber : one Text,
    username : one Text,
    password : one Text
}

sig TaxiDriver extends User {
    licenseNumber : one Text,
    taxiCode : one Text,
    state : one DriverState,
    position : one Location
}

sig Client extends User {
}
```



```
//Driver position is related to his current zone and queue
fact driverPosition{
    all t : TaxiDriver | lone z : Zone|
        t.state=available iff (t.position.zone=z and t.position.zone.queue=z.queue)
}

//A driver can only be related to one ongoing request
fact noDriverUbiquity {
    all t : TaxiDriver | lone r : Request | r.driver=t and r.state=ongoing
}

//An ongoing request is related to one unavailable driver
fact ongoingRequestOccupiesDriver {
    all r : Request | r.state=ongoing iff r.driver.state=unavailable
}
//A Driver is not in a Queue if and only is unavailable, else is available and in a queue
fact driverInQueue {
    all t : TaxiDriver | all q : TaxiQueue | (t in q.root.*next.taxi) iff (t.state=available)
}

//There is always at least a driver available in each queue
fact AtLeastADriver {
    all q : TaxiQueue | some t : TaxiDriver | t in q.root.*next.taxi
}

fact allNodesBelongToOneQueue {
    all n:QueueNode | one q:TaxiQueue | n in q.root.*next
}
```

Alloy World


Zone
queue
TaxiQueue
root
Demand
Booking
QueueNode
client
state
driver
state
client
driver
taxi
Client0
completed
Client1
TaxiDriver1
TaxiDriver0
state
state
available
unavailable

Design Document

System Architecture

The Software System is based on a **3-tiered client-server** model, composed by:

- DBMS on a dedicated server
- Application Server, running software logics
- Web Server, hosting web application components

A 3-tiered structure grants **scalability**, **security**, **customization**.

An important architectural choice is to develop the application using the **MVC pattern** to manage software components organization, resulting in a decoupled but cohesive structure.

To allow asynchronous messages sending from server to the client is implemented also a topic-based **Publish/Subscribe** system, used by inform the clients with notifications and warnings, bypassing client-server limits.

Moreover, the system follows a **Service-Oriented-Architecture** idea, realized by a sharp division of components as services, dividing tasks and building interfaces between them.

High-Level Components (1)

We pointed out few components to accomplish application goals, by means of their tasks and relations

UI Components

Components in charge to manage the interaction between users and application. Organized in subcomponents, basing on the different supported devices, are related with communication components which grant communication with system logic components.

Communication Manager

Component which receive service requests from the users and sends back the relative response coming from the application logic. It acts like a command parser, forwarding request to the relative component.

High-Level Components (2)

Account Manager

Component that manage the registration of new users, creating new accounts, and also allowing the logging in for already registered users.

Data Manager

Component in charge of manage all the data stored in the DBMS. Its role is relevant because it is invoked every time a component need to access the database to retrieve or insert data. It is the intermediate between the logic and the data layer.

High-Level Components (3)

Request Manager

Core component that fulfill the ride requests coming from the users. It elaborates the request by saving them, finding a taxi driver and notifies the user, by which the relative components. In particular, for what regards reservations, it triggers automatically research and notification at the correct time.

Queue Manager

Component in charge of ensure a fair management of taxi queues. It updates taxi queues by inserting available drivers and removing unavailable ones, and provides a driver for every request, in every zone, after having realigned driver status.

High-Level Components (4)

Location Manager

Component that manages, acquires and offers data related to positioning and location. It uses Google Maps API to elaborate user position and estimate waiting times, depending on traffic conditions.

Taxi Manager

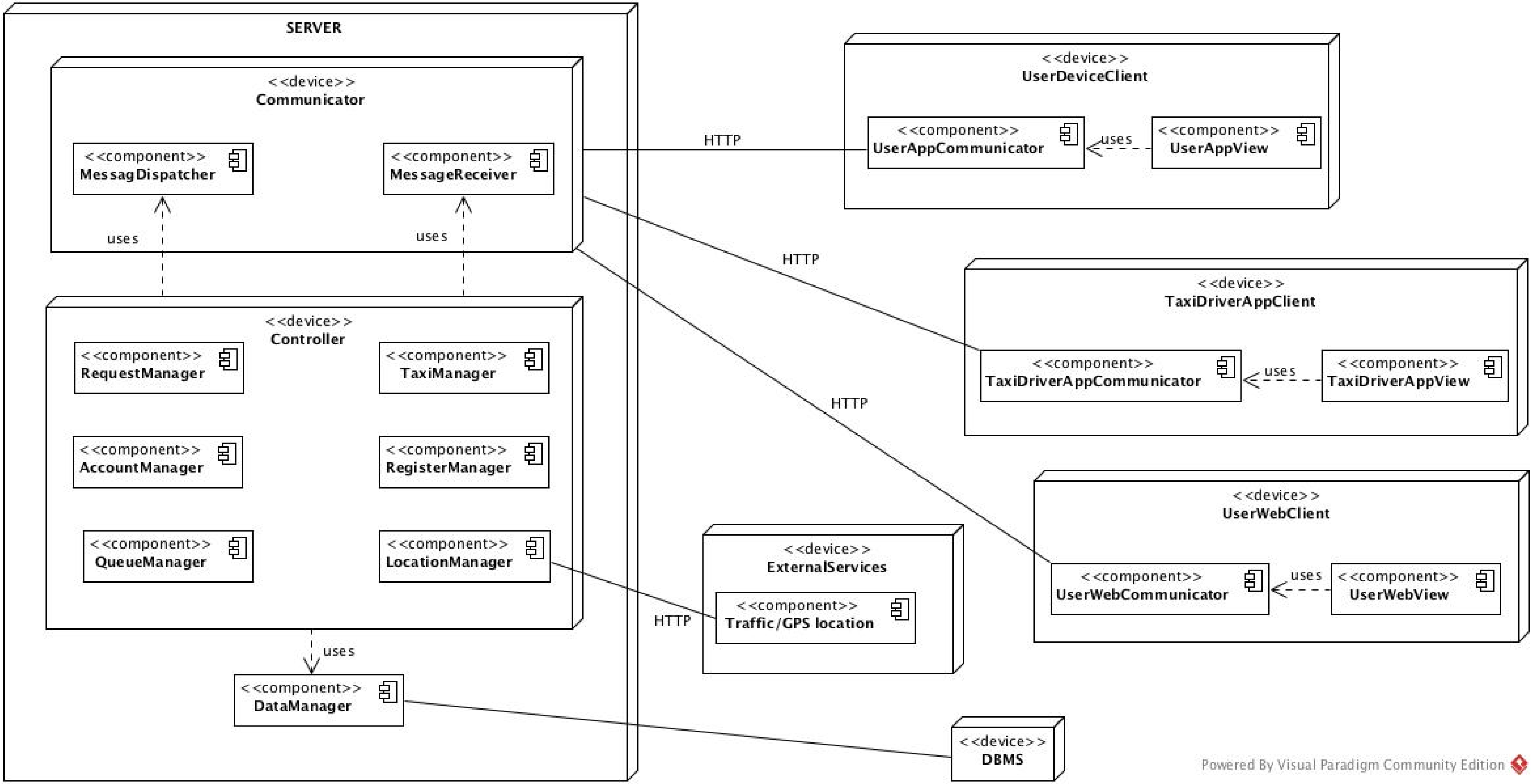
Component in charge to handle all the operation regarding driver's management, as driver availability status. It changes driver's status every time he starts or ends its service, and also handles unforeseen that can occur to the drivers during service.

High-Level Components (5)

Service Provider

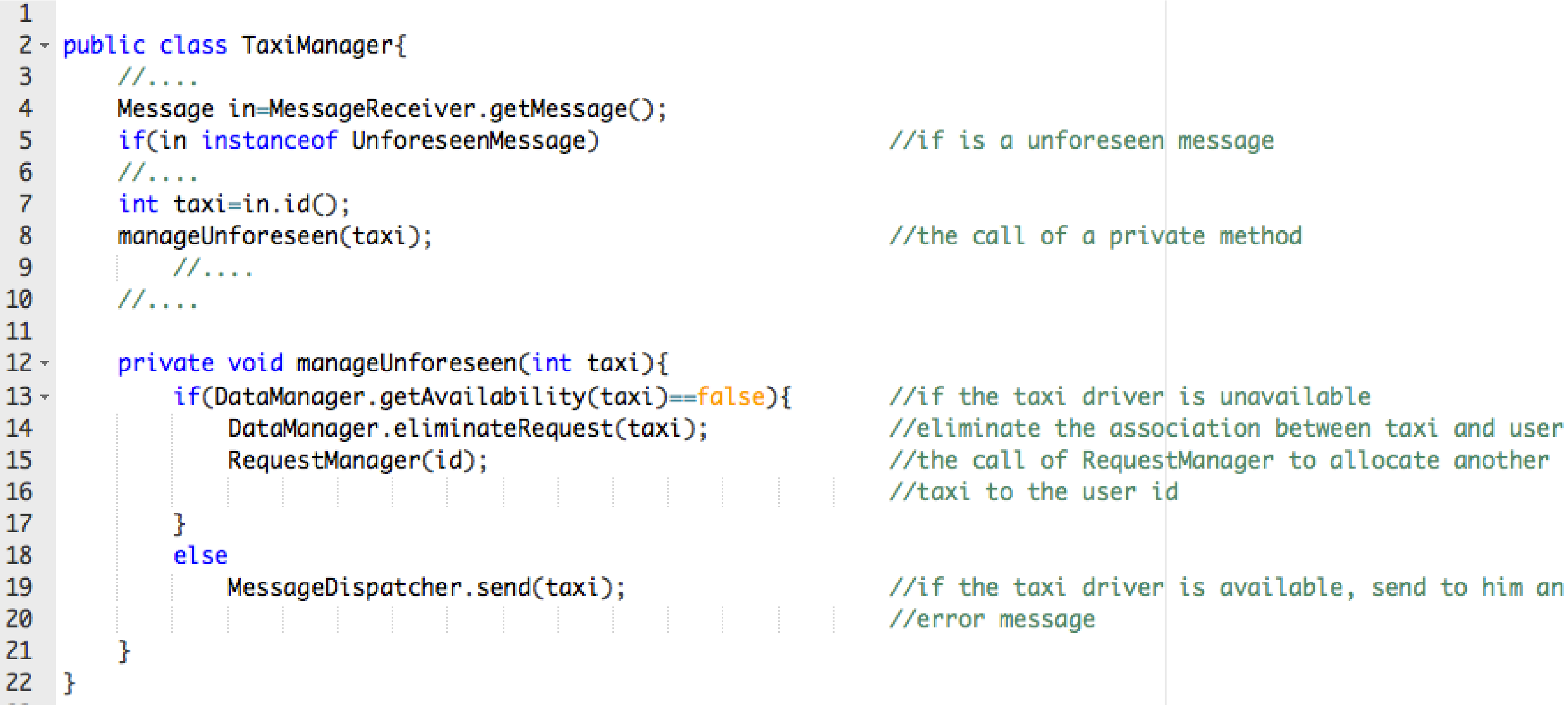
Component that allows external application to access system services. Using a dedicated component the managing of requests coming from external services can occur in a more sharp and filtered way. Acts like an interface from which "call" the desired services.

Deployment View



Algorithm Design

As regard Algorithm Design phase, the **manageUnforeseen** algorithm projected is useful to handle unforeseen that can occur to driver during their service, by allocating a new driver for the interested request.



```java

public class TaxiManager{
    //....
    Message in=MessageReceiver.getMessage();
    if(in instanceof UnforeseenMessage)                     //if is a unforeseen message
    //....
    int taxi=in.id();
    manageUnforeseen(taxi);                                 //the call of a private method
        //....
    //....

    private void manageUnforeseen(int taxi){
        if(DataManager.getAvailability(taxi)==false){       //if the taxi driver is unavailable
            DataManager.eliminateRequest(taxi);             //eliminate the association between taxi and user
            RequestManager(id);                             //the call of RequestManager to allocate another
                                                            //taxi to the user id
        }
        else
            MessageDispatcher.send(taxi);                   //if the taxi driver is available, send to him an
                                                            //error message
    }
}
```

Requirements Traceability

Requirement Traceability allow to map out the relation between every requirement and the relative system components which fulfill and ensure it.
As example here we report an extract of our traceability table, related to the previously presented requirements.

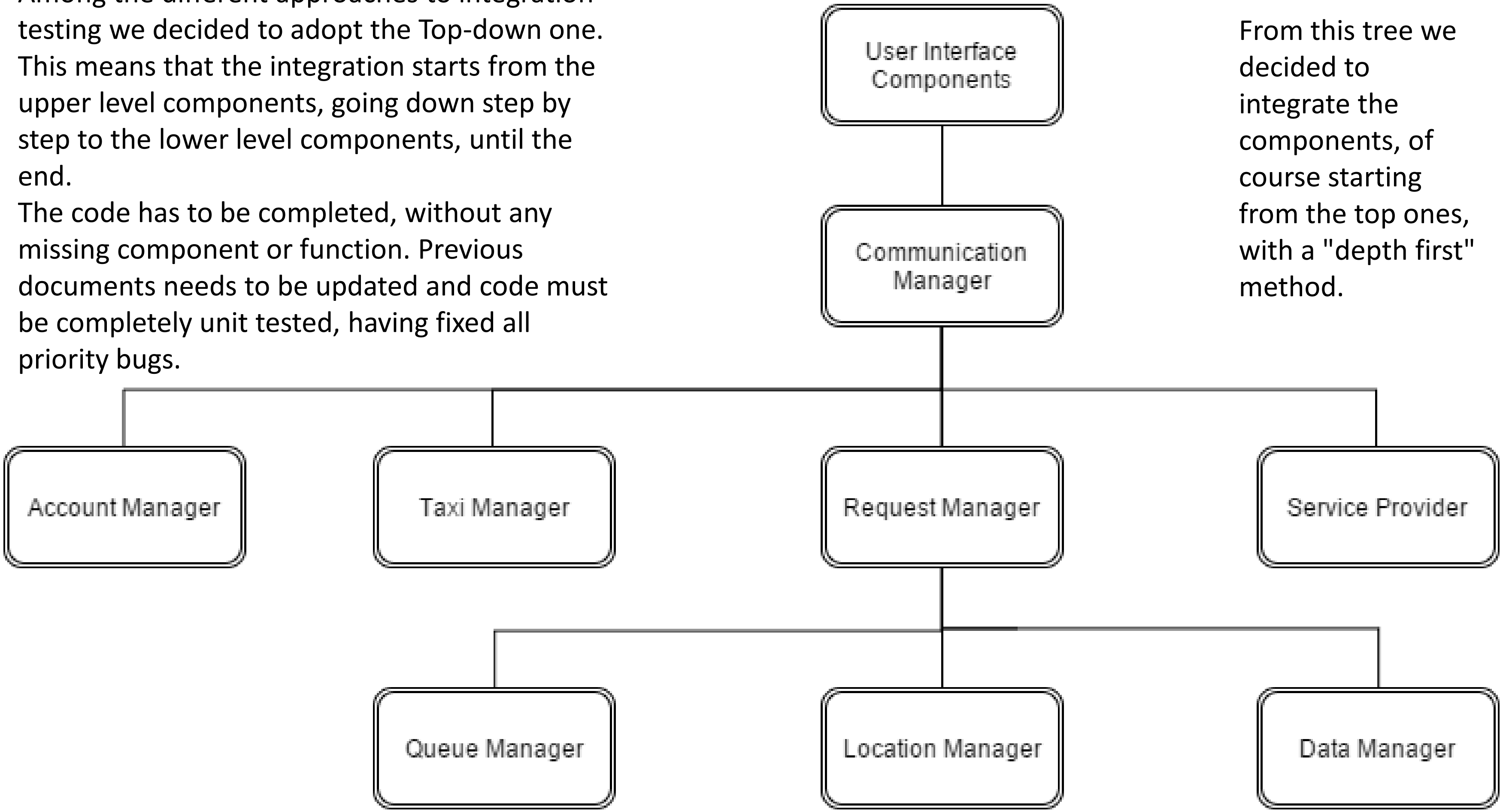
Requirement	Component
[G3.R1] user must complete all the mandatory fields of the taxi request form	User Interface Components
[G3.R2] a demand is rejected if there is already another not replied demand, performed by the same user	Request Manager

POLITECNICO
MILANO 1863

Integration Test Plan Document

Integration Strategy

Among the different approaches to integration testing we decided to adopt the Top-down one. This means that the integration starts from the upper level components, going down step by step to the lower level components, until the end.
The code has to be completed, without any missing component or function. Previous documents needs to be updated and code must be completely unit tested, having fixed all priority bugs.

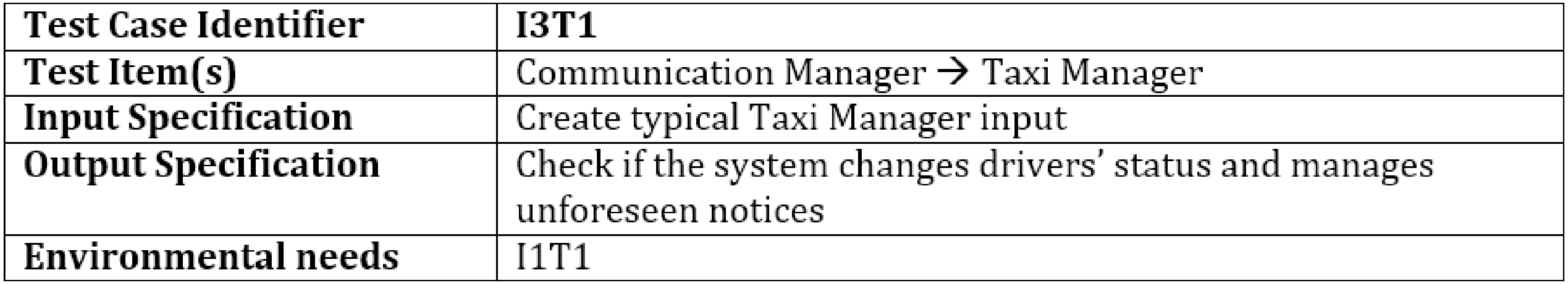


From this tree we decided to integrate the components, of course starting from the top ones, with a "depth first" method.

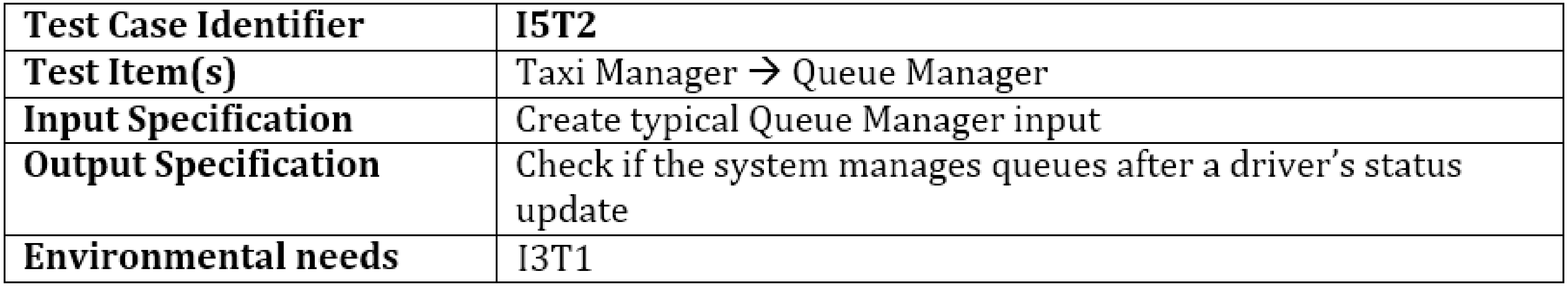
Elements to be integrated

- **User Interface Component** Responsible for the graphical interface interaction with the system.
- **Communication Manager** In charge of handle communication and messages between user and system.
- **Account Manager** Responsible for account logging and registration.
- **Data Manager** Grants managing and access to the database.
- **Request Manager** Fulfills all the request forwarded to the system.
- **Queue Manager** Responsible for taxi queues management.
- **Location Manager** In charge of retrieve and compute Google Maps data.
- **Taxi Manager** Grants a fair management of drivers availability status.
- **Service Provider** Grants access to system services for external applications.

Example of Integration Test cases (1)

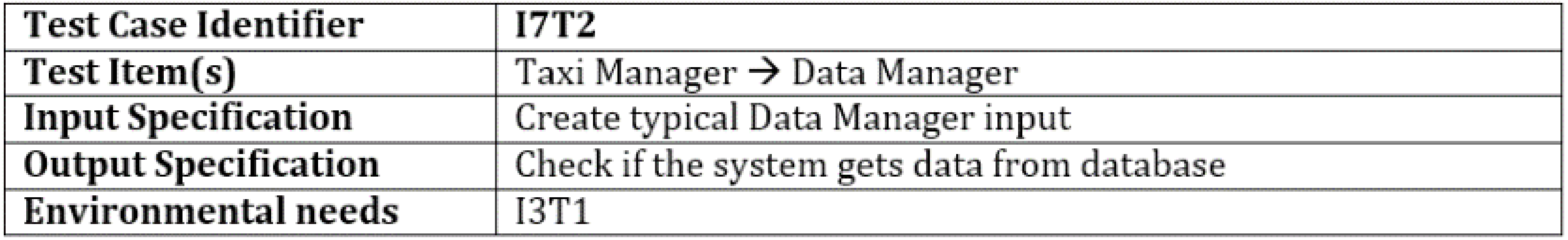


Test Case Identifier	**I3T1**
Test Item(s)	Communication Manager → Taxi Manager
Input Specification	Create typical Taxi Manager input
Output Specification	Check if the system changes drivers' status and manages unforeseen notices
Environmental needs	I1T1

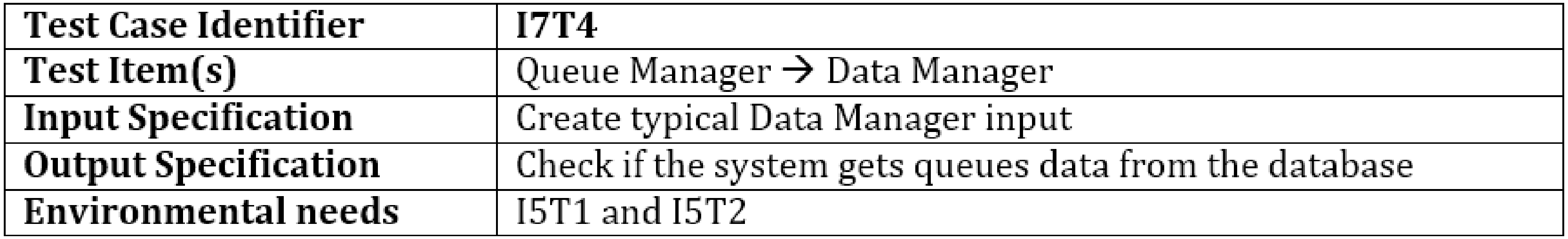


Test Case Identifier	**I5T2**
Test Item(s)	Taxi Manager → Queue Manager
Input Specification	Create typical Queue Manager input
Output Specification	Check if the system manages queues after a driver's status update
Environmental needs	I3T1

Example of Integration Test cases (2)

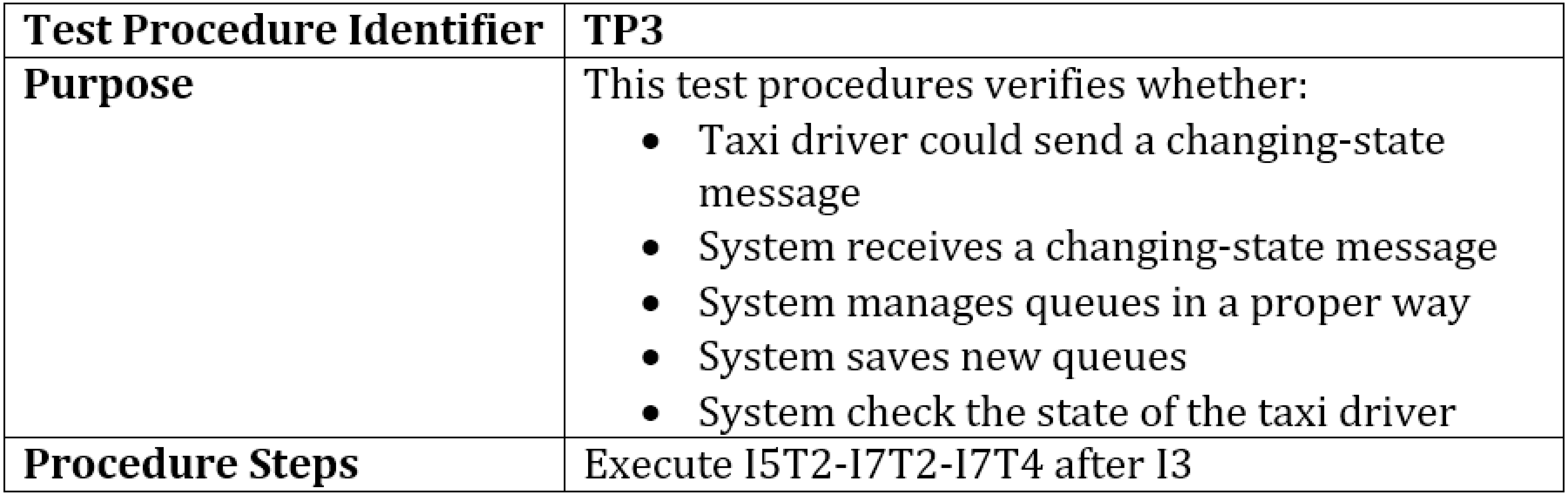


Test Case Identifier	**I7T2**
Test Item(s)	Taxi Manager → Data Manager
Input Specification	Create typical Data Manager input
Output Specification	Check if the system gets data from database
Environmental needs	I3T1



Test Case Identifier	**I7T4**
Test Item(s)	Queue Manager → Data Manager
Input Specification	Create typical Data Manager input
Output Specification	Check if the system gets queues data from the database
Environmental needs	I5T1 and I5T2

Example of Integration Test Procedure



Test Procedure Identifier	TP3
Purpose	This test procedures verifies whether: • Taxi driver could send a changing-state message • System receives a changing-state message • System manages queues in a proper way • System saves new queues • System check the state of the taxi driver
Procedure Steps	Execute I5T2-I7T2-I7T4 after I3

POLITECNICO
MILANO 1863

Project Plan

Project Estimations (1)

Project Estimation techniques allow to estimate the dimension and the effort related to the software and its development by performing some abstractions on the system functionalities.

The first analysis was performed using the **Function Points** approach: assigning to each system feature a score, on the base of their type, we computed a first rough indicator the project dimension.

Then we prosecute the estimation with **COCOMO II** method: we followed the official Model Definition Manual to obtain the *Scale Factors*, values that measures the presence of economies of scale, and the *Cost Drivers*, values that detail precisely the required effort.

Project Estimations (2)

Function Points

Internal Logic File	External Interface File	External Input	External Output	External Inquiry	Total
30	10	38	15	6	**108**

COCOMO Factors

Very Low	-									
Low	PREC	FLEX	RELY	STOR	PVOL	ACAP	PCAP	PLEX		
Nominal	RESL	PMAT	DATA	CPLX	DOCU	PCON	APEX	LTEX	TOOL	SCED
High	TEAM	RUSE	TIME							
Very High	SITE									

Estimation Result

$$UFP \times JavaEE\ Ratio = 108 \times 46 = \mathbf{4968}\ \boldsymbol{SLOC}$$

$$Effort = A \times Size^{E} \times \prod Cost\ Drivers = 2{,}94 \times 4{,}968^{1{,}11} \times 1{,}087$$

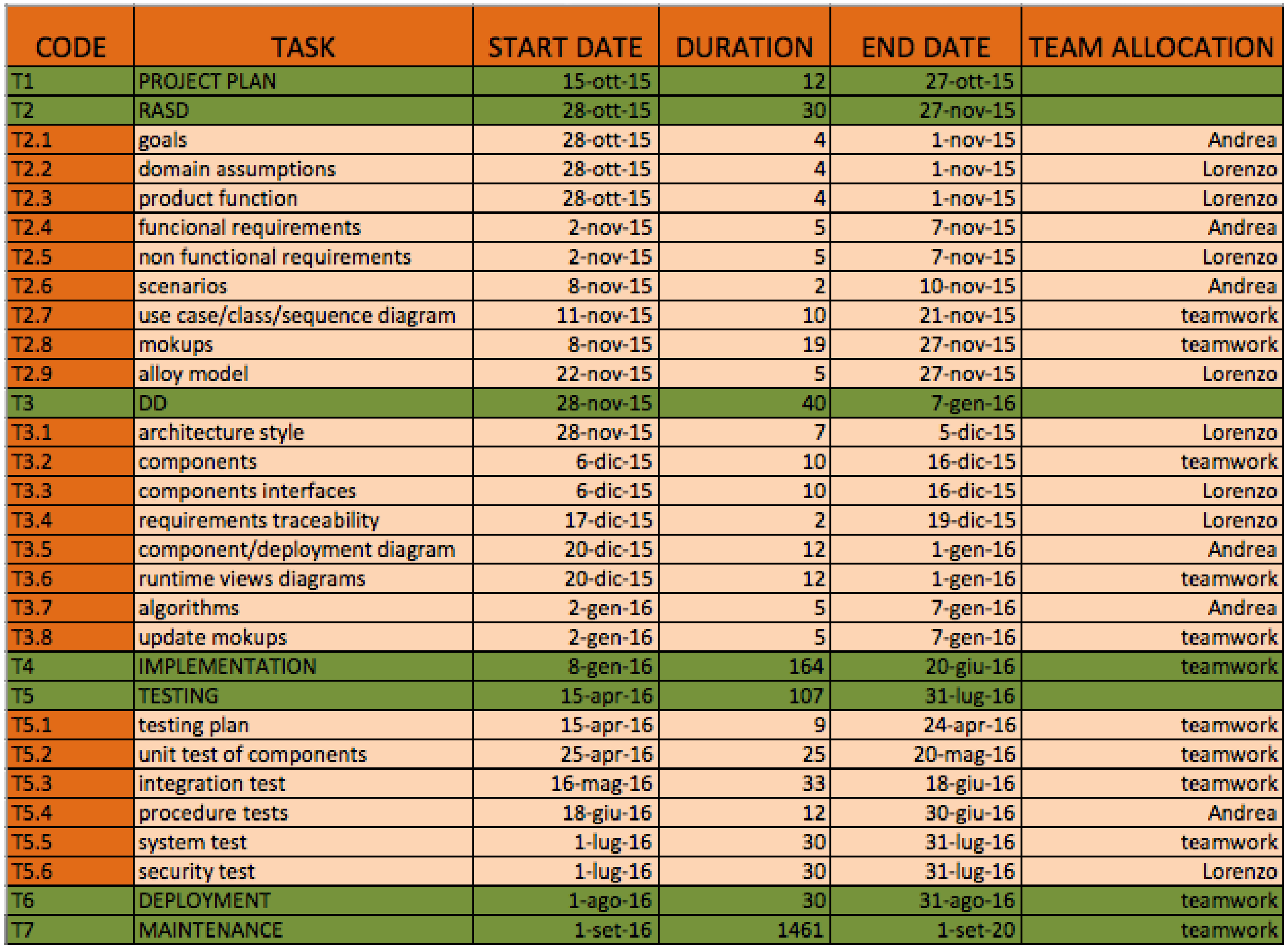
$$\boldsymbol{Effort = 18,94\ Person\ Month}$$

$$Duration = C \times Effort^{E} = 3{,}67 \times 18{,}94^{0{,}32}$$

$$\boldsymbol{Duration = 9,4\ Months}$$

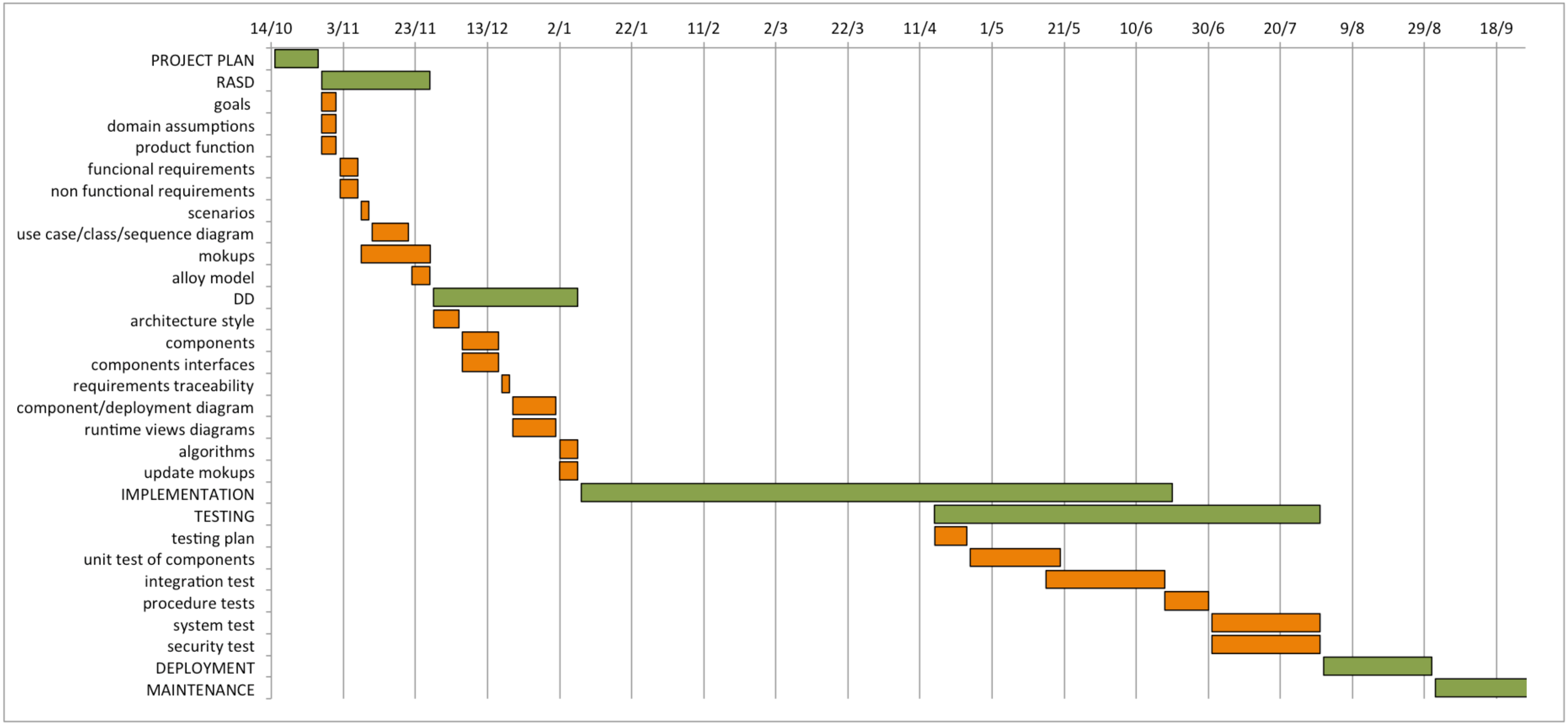
$$\boldsymbol{Required\ Personnel = \frac{Effort}{Duration} = 2,014 \longrightarrow 2\ people}$$

Task Identification and Allocation (1)



CODE	TASK	START DATE	DURATION	END DATE	TEAM ALLOCATION
T1	PROJECT PLAN	15-ott-15	12	27-ott-15	
T2	RASD	28-ott-15	30	27-nov-15	
T2.1	goals	28-ott-15	4	1-nov-15	Andrea
T2.2	domain assumptions	28-ott-15	4	1-nov-15	Lorenzo
T2.3	product function	28-ott-15	4	1-nov-15	Lorenzo
T2.4	funcional requirements	2-nov-15	5	7-nov-15	Andrea
T2.5	non functional requirements	2-nov-15	5	7-nov-15	Lorenzo
T2.6	scenarios	8-nov-15	2	10-nov-15	Andrea
T2.7	use case/class/sequence diagram	11-nov-15	10	21-nov-15	teamwork
T2.8	mokups	8-nov-15	19	27-nov-15	teamwork
T2.9	alloy model	22-nov-15	5	27-nov-15	Lorenzo
T3	DD	28-nov-15	40	7-gen-16	
T3.1	architecture style	28-nov-15	7	5-dic-15	Lorenzo
T3.2	components	6-dic-15	10	16-dic-15	teamwork
T3.3	components interfaces	6-dic-15	10	16-dic-15	Lorenzo
T3.4	requirements traceability	17-dic-15	2	19-dic-15	Lorenzo
T3.5	component/deployment diagram	20-dic-15	12	1-gen-16	Andrea
T3.6	runtime views diagrams	20-dic-15	12	1-gen-16	teamwork
T3.7	algorithms	2-gen-16	5	7-gen-16	Andrea
T3.8	update mokups	2-gen-16	5	7-gen-16	teamwork
T4	IMPLEMENTATION	8-gen-16	164	20-giu-16	teamwork
T5	TESTING	15-apr-16	107	31-lug-16	
T5.1	testing plan	15-apr-16	9	24-apr-16	teamwork
T5.2	unit test of components	25-apr-16	25	20-mag-16	teamwork
T5.3	integration test	16-mag-16	33	18-giu-16	teamwork
T5.4	procedure tests	18-giu-16	12	30-giu-16	Andrea
T5.5	system test	1-lug-16	30	31-lug-16	teamwork
T5.6	security test	1-lug-16	30	31-lug-16	Lorenzo
T6	DEPLOYMENT	1-ago-16	30	31-ago-16	teamwork
T7	MAINTENANCE	1-set-16	1461	1-set-20	teamwork

Task Identification and Allocation (2)



Risk Analysis

Risk	Dimension	Probability	Impact
Personnel shortfalls	Team	Medium	Serious
Changes in team membership	Team	Low	Critical
Bad team coordination	Team	Low	Serious
Unrealistic time/cost estimation	Planning & Control	Low	Critical
External services shortfalls	Planning & Control	Low	Serious
Software not fitting with purpose	Planning & Control	Low	Catastrophic
Design not flexible and scalable	Planning & Control	Moderate	Critical
Changes in Requirements	Requirements	Low	Catastrophic
Inadequate security features	Complexity	Moderate	Critical
Detached stakeholders	Corporation	Low	Catastrophic