POLITECNICO
MILANO 1863

Politecnico di Milano
A.A. 2016–2017
Software Engineering 2: "PowerEnJoy"
Integration Test Plan Document

Pietro Ferretti, Nicole Gervasoni, Danilo Labanca

January 15, 2017

Contents

1 Introduction

1.1 Revision History

- ITDP v1.0, published on January 15, 2017

1.2 Purpose and Scope

The purpose of this document is to provide a complete description of the integration testing plans for PowerEnjoy. It will focus on testing the proper behavior of the software by checking the interoperability between its components. It is intended for developers and for any team member involved in the testing process.

The aim of this project is to specify in detail a new digital management software for PowerEnJoy, a car-sharing service that employs electric cars only.

PowerEnjoy will offer a very valuable service to its users, letting them borrow cars to drive around the city freely, as an alternative to their own vehicles and public transport. Among the advantages of using PowerEnJoy we can note being able to find available cars in any place that is served by our system and having dedicated spots to park in (namely, PowerEnJoy's power grid stations). Furthermore, thanks to the fact that all the cars that we provide are electrically powered, PowerEnJoy is also very environmentally friendly.

1.3 List of Definitions and Abbreviations

1.3.1 Definitions

- *Guest*: a person that is not registered to the system.
- *User*: a person that is registered to the system. Users can log in to the system with their email or username and their password. Their first name, last name, date of birth, driving license ID are stored in the database.
- *Safe area*: a location where the user can park and leave the car. Users can end their ride and park temporarily only in these locations. The set of safe areas is predefined by the system.
- *Power grid station*: a place where cars can be parked and plugged in. While a car is plugged in a power grid station its battery will be recharged. Power grid stations are by definition safe areas.
- *Available car*: a car that is currently not being used by any user, and has not been reserved either. Available cars are in good conditions (not dirty nor damaged) and don't have dead batteries.
- *Reservation*:

  - the operation of making a car reserved for a user, i.e. giving permission to unlock and use the car only for that user, forbidding reservations by other users.
  - the time period between the moment a reservation is requested and the moment the user unlocks the car, or the reservation is canceled.

- *Ride*: the time period from the moment a reserved car is unlocked to the moment the user notifies that he wants to stop using the car and closes all the doors. A ride doesn't stop when a car is temporarily parked, but continues until the user chooses to leave the car definitely.
- *Temporary parking*: the act of parking a car in a safe area and, after notifying the system, locking it and leaving it for a finite amount of time. The user that does this retains the right to use the car and can unlock it later to use it again.
- *Bill*: a record of the money owed by the user at the end of a ride.
- *Outstanding bill*: a bill that hasn't been paid yet.
- *Suspended user*: a user that cannot reserve or use cars. Usually users are suspended because they have outstanding bills.
- *Payment method*: a way to transfer money from the user to the system. Our system will only accept credit cards and online accounts like Paypal.
- *Payment API*: an interface to carry out money transactions, offered by the external provider associated to the payment method used (e.g. a bank).

1.3.2 Acronyms

- **DD**: Design Document
- **RASD**: Requirements Analysis and Specification Document
- **DB**: Database
- **DOB**: Date of birth
- **PGS**: Power Grid Station
- **GPS**: Global Positioning System

1.3.3 Abbreviations

- **[Gx]**: Goal
- **[RE.x]**: Functional Requirement
- **[UC.x]**: Use Case

1.4 List of Reference Documents

- Requirements analysis and specification document: "RASD.pdf"
- Design document: "DD.pdf"
- Project description document: "Assignments AA 2016-2017.pdf"
- Example document: "Integration Testing Example Document.pdf"

2 Integration Strategy

2.1 Entry Criteria

Before starting the integration testing phase specific conditions concerning the whole project development must be met.

It is fundamental that the Requirements Analysis and Specification Document and the Design Document have been properly written and completed.

Regarding the code development, it is necessary to fully complete

- the Resource Subsystem
- the Model Component in the Application Subsystem.

All the other components will be tested, as soon as they are concluded, following the sequence stated in subsection 2.4. Furthermore, each component has to be successfully unit tested before being involved in the integration testing.

2.2 Elements to be Integrated

As stated in the Design Document (paragraph 2.2), the system is based on the cooperation of two parts, the *Application Subsystem* and the *Resource Management Subsystem*. The first handles all the operations related to the user applications and interfaces, while the second takes care of keeping track of all the automatic updates from the sensors in the cars and the charging stations. The testing process will verify the correctness of the integration between these macro-components after checking the proper cooperation of smaller components inside each subsystem. Furthermore, it will be tested the higher integration, client/server level, between

- the Web Application and the Application Subsystem
- the Car Application and the Application Subsystem

Concerning the external APIs, we expect them to work properly so we will only check, by unit testing, that the components execute correctly the API call.

Application Subsystem Components to be tested:

1. Registration
2. Account Manager
3. Car Search
4. PGS Search
5. Safe Areas Search
6. Payment
7. Reservation Manager
8. Unlocking
9. Parking Ride Manager
10. Ride Manager
11. Report Manager
12. Request Dispatcher
13. Model

It will be tested that the Request Dispatcher interacts with each component from 1 to 10 in the proper way and that these components cooperate as expected with the Model one. In addition, it will be tested the integration between the Model component and the DBMS.

Resource Management Subsystem Components to be tested:

1. Car Manager
2. PGS Manager
3. Car Handler

It will be tested the cooperation between the Car Manager and the Car Handler components and between the PGS Manager and the DBMS.

2.3 Integration Testing Strategy

We believe that the most efficient way to test the integration between PowerEnjoy's software components is the *bottom up approach*. Using this strategy, we will be able to test the cooperation between different units as soon as they are fully developed without waiting for the whole subsystem to be ready. This method will also highlight all the possible integration issues while the software is still under development, allowing quicker smaller fixes. We will start the process

from independent components which don't have any dependencies and we will built further integration test on the already tested software. As specified in the Design Document, several components of the Application Subsystem depend on the Resource Management Subsystem thus the testing phase will start in this subsystem.

2.4 Sequence of Component / Function Integration

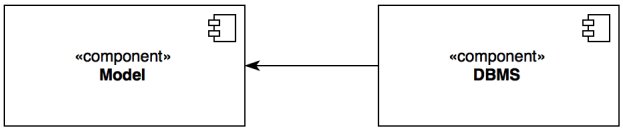
In this section we will describe in which order the components of our system will be coupled together to be tested and integrated. First we will integrate the components in each subsystem separately, then the two subsystems (the Application Subsystem and the Resource Management Subsystem) will be combined together.

We will use diagrams to make the dependencies between components clearer. Arrows from a component to another imply that the first component is necessary for the second one to work correctly.

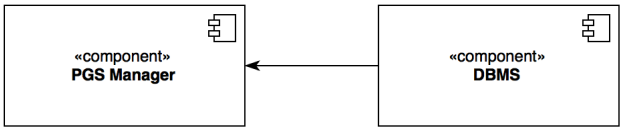
2.4.1 Software Integration Sequence

Following our choice of using a bottom-up approach, integration testing will start from two base components: the DBMS and the Car Internal System. The DBMS is free from dependencies and can be easily used as a starting point. The Car Internal System on the other hand, even if it is a prerequisite for many of the system's functions to work, requires an interface to the Resource Management Subsystem that we haven't integration tested yet. Fortunately the interface is used only for sending updates about its status, so the interface functions can be easily replaced by an appropriate stub.

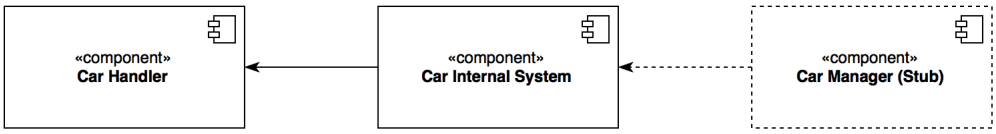
Model The first two elements to be integrated will be the Model and the DBMS. The Model is the foundation of the Application Subsystem and integrating it will let us test all the other components in the subsystem.



Power Grid Stations Manager On the Resource Management Subsystem we can start from integrating the Power Grid Stations Manager with the DBMS. The PGS Manager is almost fully independent from the other components and can be tested easily.

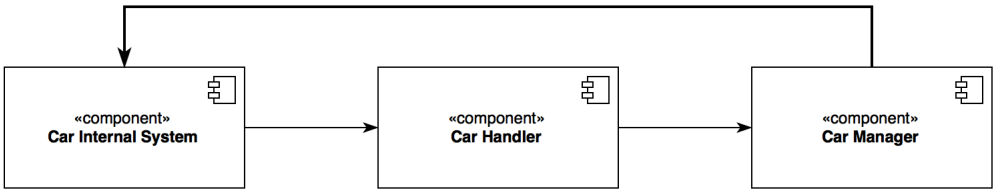


Car Handler The Car Handler is the component that is in charge of issuing commands to the car. We can start testing its behavior as soon as the Car Internal System is fully unit tested. We will make use of a stub to simulate the Resource Subsystem Interface.

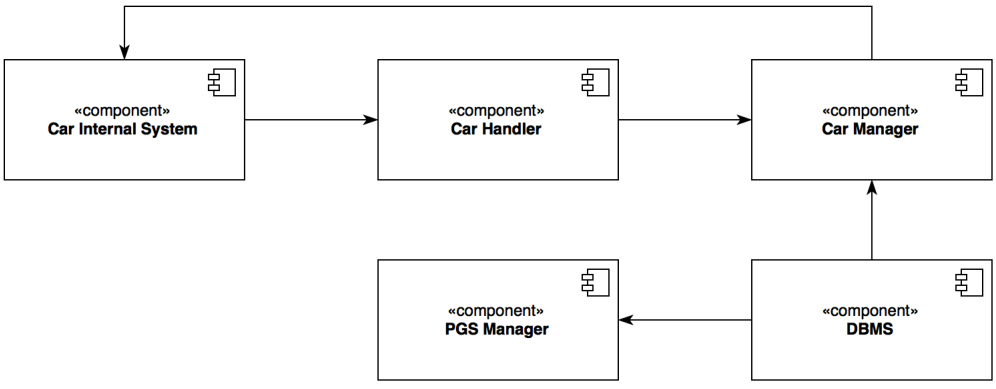


Car Manager After the Car Handler is integrated we can finally incorporate the Car Manager, the component doing the heavy lifting in the Resource Management Subsystem.

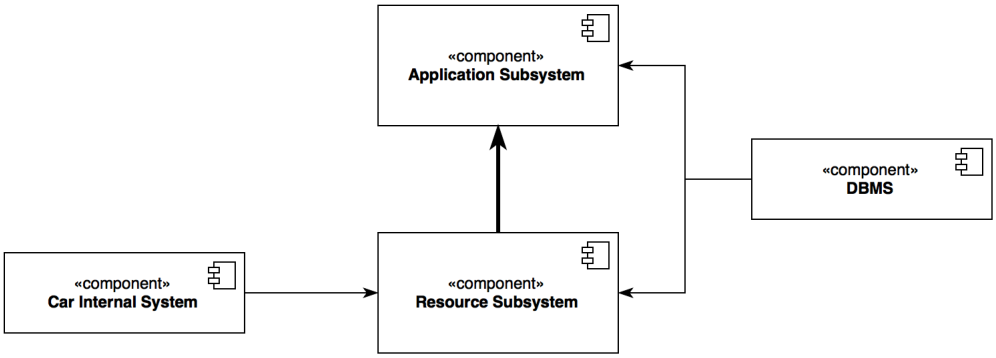
Once the Car Manager has been integration tested, we can actually test the interaction between the Car Internal System and the Resource Subsystem Interface.



At this point the Resource Management Subsystem is fully integrated.

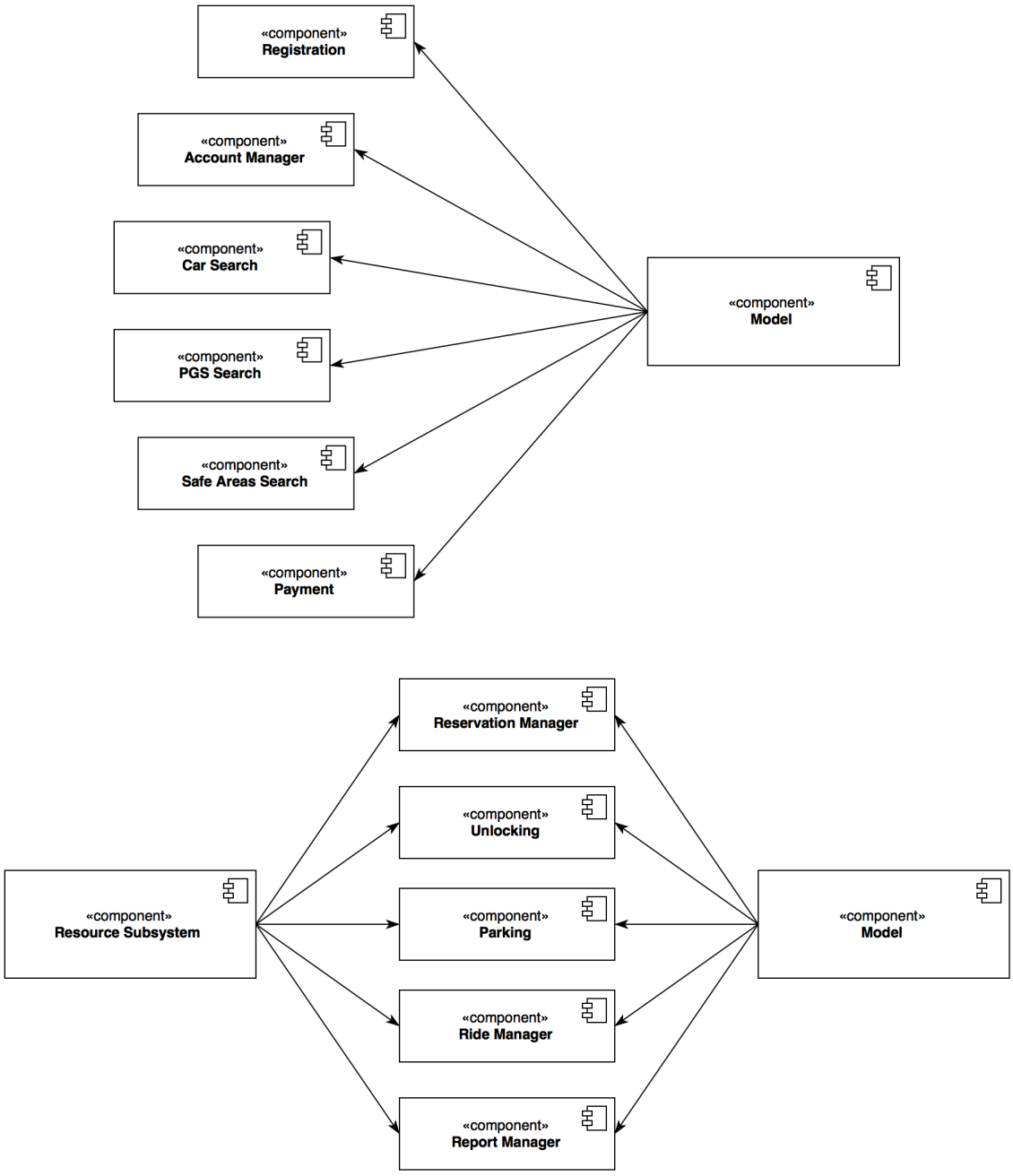


Application and Resource Subsystems Integration We can now proceed to integrating the Application Subsystem and the Resource Subsystem, again following the bottom-up approach.



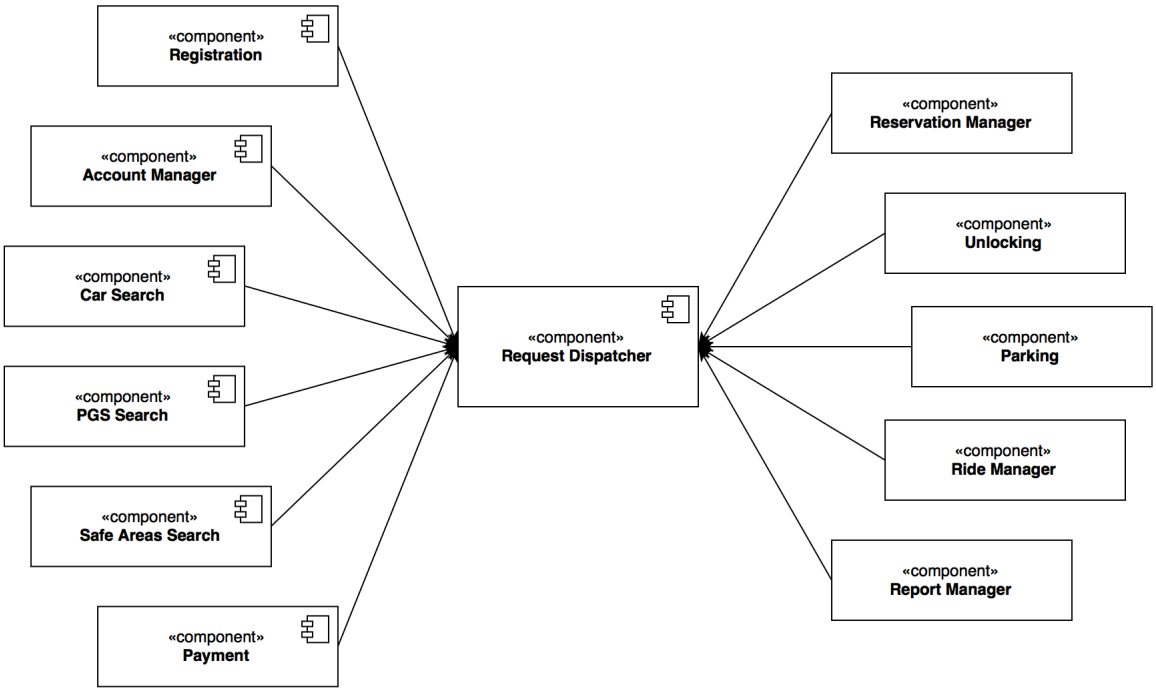
Application Subsystem Components Most of the components in the Application Subsystem are independent from each other, therefore they can be freely tested in parallel.

The Registration Component, the Account Manager, the Car Search Component, the PGS Search Component, the Safe Areas Search Component and the Payment Component only need the Model, while the Reservation Manager, the Unlocking Component, the Parking Component, the Ride Manager and the Report Manager also need to interface with the Resource Subsystem.

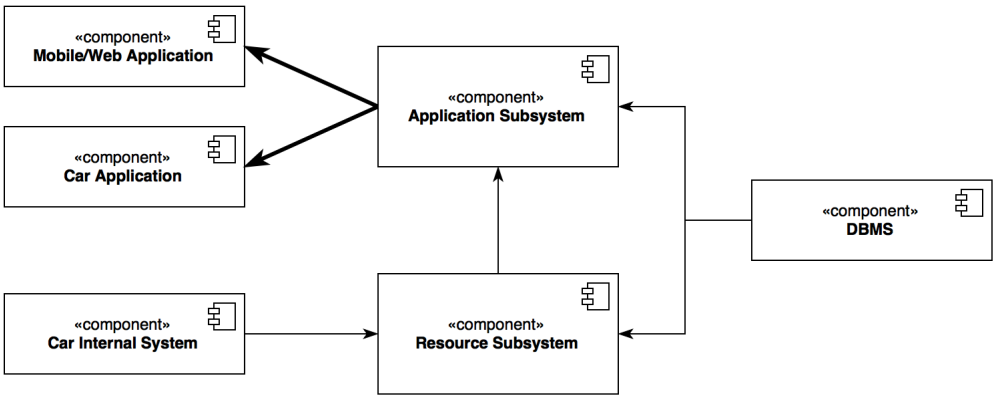


Request Dispatcher The Request Dispatcher acts as a wrapper for all of the requests to the application API, processing and distributing them to the corresponding components.

This allows us to integrate it with thread based testing: when testing for one of the intermediate components has been finished, we can test the corresponding functionality on the Request Dispatcher, that will interact with the specific component. The request dispatcher will be integrated one by one with all the other components, until the Application Subsystem is complete.



Client Applications Integration Finally, after the whole central system is tested and has been successfully integrated, we will test the interaction between the client applications and the system.



3 Individual Steps and Test Description

Also, try all combinations of null values. Also, try passing parameters of the wrong type.

AUTHENTICATION! - maintenance - pgs - car update - application components?

3.1 Model - DBMS

azioni da componenti sul model - le azioni di delete - richieste di tutte le reservation/ride esistenti (serve?)

Insert User	
Input	*Result*
A username, an email, the user's personal data, a license id number and a payment method (e.g. a valid credit card number).	A new user is inserted in the database.
Some of the parameters are missing.	The insertion fails.
Invalid parameters?	error

Update User	
Input	*Result*
correct request	user data updated
updating with unacceptable data	fail
trying to update a nonexistent user	fail

Insert Bill	
Input	*Result*
A valid user and a well-formed bill.	The bill is added to the user's bills in the database.
A valid user and an ill-formed bill	*An error is raised.*
A non-existent user and a bill	*An error is raised*

List Bills	
Input	*Result*
A valid user	Returns the user's bills
A valid user with no bills	Returns an empty set
A non-existent user	*An error is raised*

List Cars	
Input	*Result*
Nothing	Returns a set of all cars and *information about them, position, battery ecc.*

List Power Grid Stations	
Input	*Result*
Nothing	Returns a set of all power grid stations and *their position, their availability*

List Safe Areas	
Input	*Result*
Nothing	Returns a set of all safe areas and *their position*

Insert Reservation	
Input	*Result*
A valid user, an available car	A new reservation is created in the database.
A valid user, a car that is not available	*An error is raised*
A non-existent user, a car	*An error is raised*
A suspended user, a car	*An error is raised*

Update Reservation	
Input	*Result*
An existent ongoing reservation, valid data to update	The reservation is updated with the new data.
An existent ongoing reservation, invalid data to update	error
A canceled reservation, data to update	error
An ended reservation, data to update	error
A non-existest reservation, data to update	error

Insert Ride	
Input	*Result*
A valid user, a car currently reserved by the user	A new ride is created in the database.
A valid user, a car not currently reserved by the user	error
A valid user, a non-existent car	error
A non-existent user, a car	error
A suspended user, a car	error

Update Ride	
Input	*Result*
A valid ride and valid data to update	The ride is updated with the new data.
A valid ride, invalid data to update	error
A non-existent ride, data to update	error

3.2 PGS Manager - DBMS

azioni da interfaccia esterna - update plugs -> disponibile -> non disponibile
in teoria non dovrebbero esserci richieste errate perché sono solo messaggi delle stazioni, ma mettiamo casi di errore

Update Plug Availability	
Input	*Result*
A valid plug id and its availability (different from the current one)	The plug's availability is updated in the database.
A valid plug id and its availability (the same as the current one)	Nothing changes, returns a "Not Modified"?
An invalid plug id and an availability	error

3.3 Car Handler - Car Internal System

azioni da car manager a car handler - richiesta di informazioni - lock - unlock

Request Update from Car	
Input	*Result*
A valid car id and a valid *request type*	The request is sent to the car and the car sends an update to the Car Manager.
A valid car id and an invalid *request type*	error
A non-existent car id and a request type	error

Lock Car	
Input	*Result*
A valid car id	The car is locked.
A non-existent car id	error

Unlock Car	
Input	*Result*
A valid car id	The car is unlocked.
A non-existent car id	error

3.4 Car Manager - Car Handler - DBMS

3.4.1 Maintenance

List Cars in Need of Maintenance	
Input	*Result*
Valid authentication credentials	Returns a set with all cars in need of maintenance, along with *information about the issues and their position*
Invalid authentication credentials	error (access forbidden)

Set Car as Available	
Input	*Result*
Valid authentication credentials, a valid car id	The car is set as available.
Valid authentication credentials, a non-existent car id	error
Invalid authentication credentials, a car id	error

Set Car as Unavailable	
Input	*Result*
Valid authentication credentials, a valid car id and a reason for making it unavailable	The car is set as unavailable and the reason is saved on the database.
Valid authentication credentials, a valid car id and no reason to for making it unavailable	error?
Valid authentication credentials, a non-existent car id	error
Invalid authentication credentials, a car id	error

3.4.2 Interface to the Application Subsystem

Request to Lock Car	
Input	*Result*
A valid car id	The car is locked.
A non-existent car id	error

Request to Unlock Car	
Input	*Result*
A valid car id	The car is unlocked.
A non-existent car id	error

3.4.3 Car Interface

Car Status Update	
Input	*Result*
A car's valid authentication credentials, well-formed info about the car's status	The car's status is updated in the database.
A car's valid authentication credentials, ill-formed info about the car's status	error
Invalid car authentication credentials and info about a car's status	error

3.5 Application Subsystem Components - Model

azioni da request dispatcher

3.5.1 Registration Component - Model

- registrazione utente

Register User	
Input	*Result*
A not-yet-taken username, an email, valid personal data, a valid payment method and a valid driving license id number	A new user is created in the database and an email containing the user's password is sent to the email address provided.
A not-yet-taken username, an email, valid personal data, a valid payment method and a picture of a valid driving license	A new user is created in the database and an email containing the user's password is sent to the email address provided.
A not-yet-taken username, an email, valid personal data, a valid payment method and an invalid driving license	error
A not-yet-taken username, an email, valid personal data, an invalid payment method and a driving license	error
A not-yet-taken username, an email, invalid personal data, a payment method and a driving license	error
An already taken username, an email, personal data, a payment method and a driving license	error
An ill-formed username, an email, personal data, a payment method and a driving license	error

3.5.2 Account Manager - Model

- login - getinfo utente - update info utente - list bills

Login	
Input	*Result*
A valid username and the correct corresponding password	Creates a session for the user and returns a session token.
A valid username and a wrong password	error
A non-existent username and a password	error

Get User Info	
Input	*Result*
A valid session token	Returns all the session user's info
An invalid session token	error

Update User Info	
Input	*Result*
A valid session token and valid new data to update	The user info is updated with new data on the database.
A valid session token and invalid new data	error
An invalid session token and new data	error

List User Bills	
Input	*Result*
A valid session token	Returns a set with all the user's bills
An invalid session token	error

3.5.3 Car Search Component- Model

- find cars with position - find cars with address

Find Cars with Position	
Input	*Result*
A valid geographic position	Returns a list of all the cars close to that position, along with *their info*.
An invalid geographic position	error

Find Cars with Address	
Input	*Result*
A valid well-formed address	Returns a list of all the cars close to that address, along with *their info*.
An invalid address	error

3.5.4 PGS Search Component- Model

- find stations with position - find stations with address

Find Stations with Position	
Input	*Result*
A valid geographic position	Returns a list of all the charging stations close to that position, along with *their info*.
An invalid geographic position	error

Find Stations with Address	
Input	*Result*
A valid well-formed address	Returns a list of all the charging stations close to that address, along with *their info*.
An invalid address	error

3.5.5 Safe Areas Search Component - Model

- find safe areas with position - find safe areas with address

Find Safe Areas with Position	
Input	*Result*
A valid geographic position	Returns a list of all the safe areas close to that position.
An invalid geographic position	error

Find Safe Areas with Address	
Input	*Result*
A valid well-formed address	Returns a list of all the safe areas close to that address.
An invalid address	error

3.5.6 Payment Component- Model

- pay a bill

Pay a Bill	
Input	*Result*
A valid session token, a bill that needs to be paid and a valid payment method	The transaction is carried out and the bill is marked as paid.
A valid session token, a bill that needs to be paid and an invalid payment method	error
A valid session token and a bill that needs to be paid	The system uses the payment method saved for the user to carry out the transaction; if it succeeds the bill is marked as paid, otherwise *error*
A valid session token and a bill that has already been paid	error
A valid session token and a non-existent bill	error
An invalid session token and a bill	error

3.5.7 Reservation Manager - Model - Resource Subsystem

- crea reservation - annulla reservation

Create Reservation	
Input	*Result*
A valid session token for a user that has no current reservation, an available car	A new reservation is created.
A valid session token for a user that has no current reservation, a car that is not available	error
A valid session token for a user that has no current reservation, a non-existent car	error
A valid session token for a user with an ongoing reservation, a car	error
An invalid session token and a car	error

Cancel Reservation	
Input	*Result*
A valid session token and an ongoing reservation made by the user	The reservation is canceled.
A valid session token and an ongoing reservation made by another user	error
A valid session token and a reservation that has already ended	error
An invalid session token and a reservation	error

3.5.8 Unlocking Component - Model - Resource Subsystem

- sblocca macchina (-> nb fa partire il conto del tempo)

Unlock Car with Code	
Input	*Result*
A valid session token, the code of the car that is currently being reserved by the user	The car is unlocked.
A valid session token, the code of a car that is not currently being reserved by the user	error
A valid session token, an invalid car code	error
An invalid session token	error

Unlock Car with Position	
Input	*Result*
A valid session token for a user that has an ongoing reservation, a geographic position close to the car that the user has reserved	The car is unlocked.
A valid session token for a user that has an ongoing reservation, a geographic position far from the car that the user has reserved	error
A valid session token for a user that has no ongoing reservation, a geographic position	error
An invalid session token	error

3.5.9 Parking Component - Model - Resource Subsystem

- metti la macchina in sosta

Park Car	
Input	*Result*
A valid session token for a user that has an ongoing ride	If the car has been stopped and the car's position is in a safe area, then the car is locked, otherwise *error*
A valid session token for a user that has no ongoing ride	error
An invalid session token	error

3.5.10 Ride Manager - Model - Resource Subsystem

- fine ride

End Ride	
Input	*Result*
A valid session token for a user that has an ongoing ride	If the car has been stopped and the car's position is in a safe area, then the car is locked and the ride is marked as ended, otherwise *error*
A valid session token for a user that has no ongoing ride	error
An invalid session token	error

3.5.11 Report Manager - Model - Resource Subsystem

- aggiungi report

Report Issue	
Input	*Result*
A valid session token for a user with an ongoing reservation or ride, an explanation of the issue	The issue is recorded and the car is set as unavailable as soon as possible
A valid session token for a user with no ongoing reservation or ride	error
An invalid session token	error

request dispatcher:

3.6 Request Dispatcher - Application Components

- questa è l'api descritta nel DD, diciamo soltanto che deve comportarsi nel modo che abbiamo già detto nei dettagli

client applications:

3.7 Mobile/Web Application - Application Subsystem

- qui diciamo che proviamo l'applicazione in tutte le sue funzionalità e devono funzionare tutte

3.8 Car Application - Application Subsystem

- stessa cosa qui

3.9 Integration between subsystems

4 Tools and Test Equipment Required

4.1 Tools

In order to test the software in the most efficient way, we will use some tools for automated testing.

Mockito It is an open source testing framework for Java released under the MIT License. It allows the *creation of mock objects* in automated unit tests. It supports *scaffolding and stubs*.
`http://site.mockito.org/`

JUnit JUnit is a simple, open source framework used *to write and run repeatable tests*. It is an instance of the xUnit architecture for unit testing frameworks. JUnit features include: assertions for testing expected results, test fixtures for sharing common test data and test runners for running tests.
`http://junit.org/junit4/`

Arquillian It is a test framework that can be used to perform testing inside a remote or embedded container, or deploy an archive to a container so the test can interact as a remote client; in addition Arquillian integrates with other testing frameworks (e.g., JUnit 4), allowing the use of IDE and Maven test plugins. It will be used mainly for *testing dependency injections* and *transaction control*.
`http://arquillian.org/`

4.2 Test Equipment Required

In order to ensure our software to work properly in the majority of cases, we will test our client components on:

- an Android smartphone or tablet,
- an Apple smartphone or tablet,
- a Windows smartphone or tablet.

Since we are developing a responsive web application, it is enough to check the software behavior with different mobile OS and thus browsers. Therefore we will also test the application on a computer using different browsers (Chrome, Explorer, Safari, Firefox). Additionally we will test our car application on each model of car owned by the PowerEnjoy company.

5 Program Stubs and Test Data Required

6 Effort Spent

- Pietro Ferretti: hours of work
- Nicole Gervasoni: hours of work
- Danilo Labanca: hours of work