POLITECNICO DI MILANO

POWERENJOY

SOFTWARE ENGINEERING 2

Integration Test Plan Document

Authors:
Giancarlo COLACI
Giulio DE PASQUALE
Francesco RINALDI

Supervisor:
Elisabetta DE NITTO

January 15, 2017

Contents

1 Introduction

1.1 Revision History

Version	Date	Author(s)	Summary
1.1	14/01/2017	Giancarlo Colaci, Giulio De Pasquale, Francesco Rinaldi	Name refactoring
1.0	08/01/2017	Giancarlo Colaci, Giulio De Pasquale, Francesco Rinaldi	Initial Release

1.2 Purpose and Scope

The Integration Test Plan Document (**ITPD**) describes how integration tests are to be performed. The tests here described focus on the information's flow between every module as a whole. Specifically it describes the adopted methodologies ranging from the sets of all tests to be performed to the tools used throughout the whole process. The system will be an optimization of a pre-existing system for renting cars already in use in some cities. The new system will let users to check reservability and status of available cars, rent or reserve them through a mobile or a web application in a more simple and effective way.

1.3 Definitions, Acronyms, Abbreviations

Below there are definitions of some terms that will be used in the document, in order to avoid any ambiguity in their use and their understanding.

Definitions

Component: each of the low level components realizing the functionalities of a subsystem

Subsystem: a high-level functional unit of the system

Requests: reservations and rents

Acronyms

DBMS: DataBase Management System, a software that control the creation, maintenance and use of a database. (e.g. MySQL)

API: Application Programming Interface

RASD: Requirement Analysis and Specification Document

DD: Design Document

MSO: Money Saving Option

RMSS: Request Management Subsystem

ITPD: Integration Test Plan Document

1.4 Reference documents

- Requirements Analysis and Specification Document produced before
- Design Document produced before
- Specification Document
- JUnit [1]
- JMockit [2]
- Arquillian[3]

1.5 Document Structure

1. **Introduction**: this section introduces the Integration Test Plan Document. It contains a justification of his utility and its main use.
2. **Integration Strategy**: this section is divided into different parts.
    - *Entry Criteria*: This section lists all the prerequisites that need to be met *before* any integration testing begins in order achieve valid and worthwhile results.
    - *Elements to be Integrated*: this sections gives a global view of the components of the application to be integrated, in a consistent way with our design.
    - *Integration Testing Strategy*: this sections gives a more detailed view of the integration testing approach, such as top-down, bottom-up, functional groupings, etc.., and the main reasons behind it.
    - *Component / Subsystem Testing:*
        - Component Testing: this section illustrates how every component will be integrated along in order to constitute a subsystem.
        - Subsystem Testing: this section shows the order in which subsystems will be integrated.
3. **Individual Steps and Test Description**: this section describes the type of tests that will be used to verify that the elements integrated perform as expected.
4. **Tools and Test Equipment Required**: this section presents all the tools and test equipment needed to accomplish the integration.
5. **Program Stubs and Test Data Required**: this section aims to identify any program stubs or special test data required for each integration step, based on the testing strategy and test design.
6. **Appendix**: in this section will be listed the different tools we used and the hours of work spent by each member of the team.

[1] http://junit.org/junit4/javadoc/latest/
[2] http://jmockit.org/api1x/overview-summary.html
[3] http://arquillian.org/guides/

2 Integration Strategy

2.1 Entry Criteria

This section lists all the prerequisites that need to be met *before* any integration testing begins in order achieve valid and worthwhile results.

All the classes and methods will be **tested** against several unit tests to detect major faults in algorithms and classes' structure. Each unit test has to cover at least the 90% of lines of code and will be run automatically on each build. However unit testing is not in the scope of this document and will not be specified in further detail.

The whole project will be **constantly inspected** to ensure maintainability, detect possible issues and coding conventions breakages which could increase the testers' effort in next testing phases. Continuous code inspection must be performed using automated tools as much as possible: manual testing should be reserved for the most difficult features to test.

Finally, the **documentation** has to be complete and up-to-date to be used as a reference for integration testing development. In particular, the public interfaces of each class and module should be well referenced. Where necessary, a formal specification language can be used.

The following documents must be delivered before integration testing can begin:

- *Requirement Analysis and Specification Document of PowerEnjoy*
- *Design Document of PowerEnjoy*
- *Integration Testing Plan Document of PowerEnjoy*

2.2 Elements to be Integrated

The following list represents every component grouped into several subsystems:

Core Data

- Data Manager
- DataBase Management System

Account Management

- Authentication Manager
- AccountInformation Manager

Car Management

- ADS_Application Manager
- Car Manager

Request Management

- CheckAvailability Manager
- Reservation Manager

Events Handler

- Payment Manager
- Notification Manager

Interfaces

- Web GUI
- App (Mobile) GUI

For a detailed description of each components function and interaction refer to the **Design Document**, section 2.3.

2.3 Integration Testing Strategy

The chosen strategy for the integration testing is the **bottom-up** approach.

It allows the testers to focus on each main component as little as possible since each test will be developed in great detail starting from the inner classes in our specification: from that point on, each component will rely on a strong codebase which reduces the time spent on reviewing the project internals.

2.4 Component / Subsystem Testing

Due to the complex nature of testing an entire system, we planned the integration testing following two point of views, both of them catalogued through a **dependency-driven** order: the detailed *component* testing and the top-view *subsystem* testing.

2.4.1 Component Testing

This section illustrates how every component will be integrated along in order to constitute a subsystem.

Core Data

The first two elements to be integrated are the **Data Manager** and the **Database Management System** components. We start from here because every other component relies on Data Manager to perform queries on the underlying data structure.

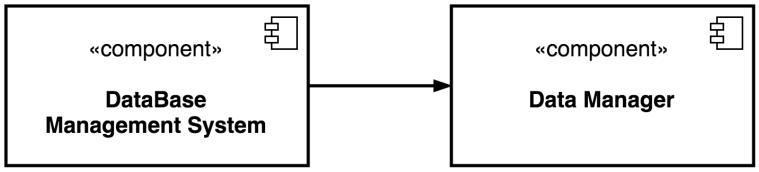

Figure 1: Core Data

Account Management

The *Account Management* subsystem relies on the **Authentication Manager** and the **AccountInformation Manager**. The first one manages the correctness of user data submitted by the clients to provide access to the PowerEnjoy services; the latter, instead, edits and provides access to every account in the DBMS.

They both need the **Core Data** subsystem to operate correctly.

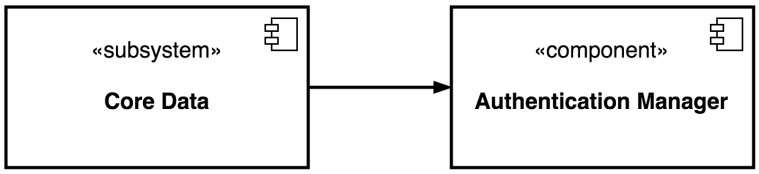

Figure 2: Authentication Manager

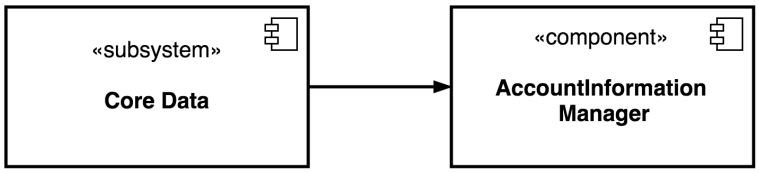

Figure 3: AccountInformation Manager

Car Management

The *Car Management* subsystem relies on the **ADS_Application Manager** and the **Car Manager**. The **ADS_Application Manager** has to guarantee the correct communication between the **Car Manager** and every ADS installed on each car. The **Car Manager** handles the status of each car by communicating the ADS installed on each car through the **ADS_Application Manager**.

They both need the **Core Data** subsystem to operate correctly.

Figure 4: ADS_Application Manager

Figure 5: Car Manager

Request Management

The *Request Management* subsystem relies on the **CheckAvailability Manager** and the **Reservation Manager**. The **CheckAvailability Manager** is responsible for searching available cars in a specific location. The **Reservation Manager** handles each reservation by creating new entries, checking active entries information and terminating active ones.

They both need the **Core Data** and the **Car Management** subsystems to operate correctly.

Figure 6: CheckAvailability Manager

Figure 7: Reservation Manager

Events Handler

The *Event Handler* subsystem relies on the **Payment Manager** and the **Notification Manager**. The **Payment Manager** calculates the fees for the rides at the end of each reservation and send a payment request to an external service. The **Notification Manager** handles every notification sent by the system to both server and client side.

They both need the **Core Data** and the **Request Management** subsystems to operate correctly.

Figure 8: Payment Manager

Figure 9: Notification Manager

Interfaces

The *Interface* subsystem relies on the **App GUI** and the **Web GUI**.

They both need the **Request Management** and **Account Management** subsystems to operate correctly.

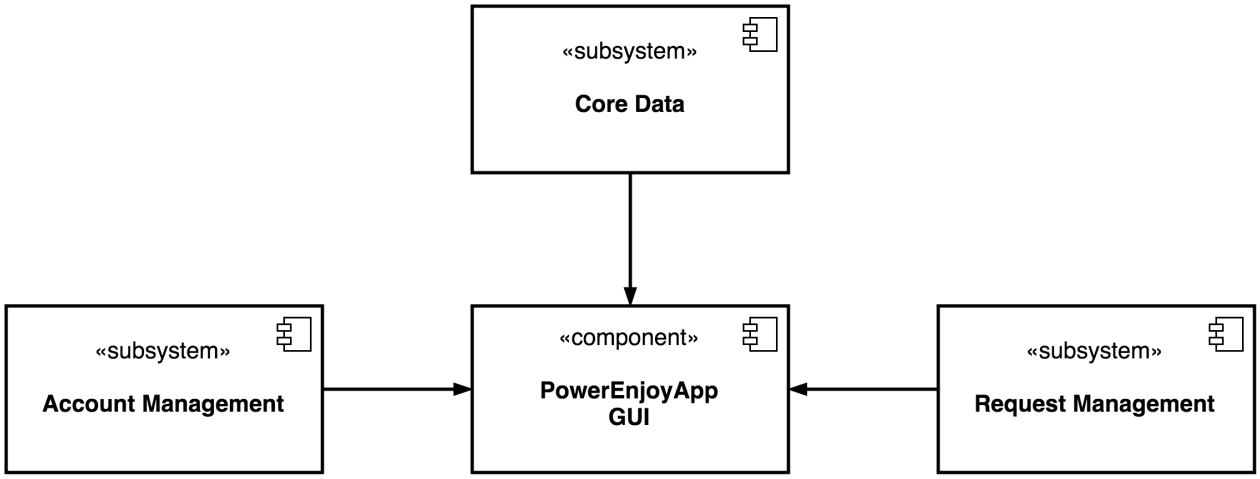

Figure 10: PowerEnjoyApp GUI

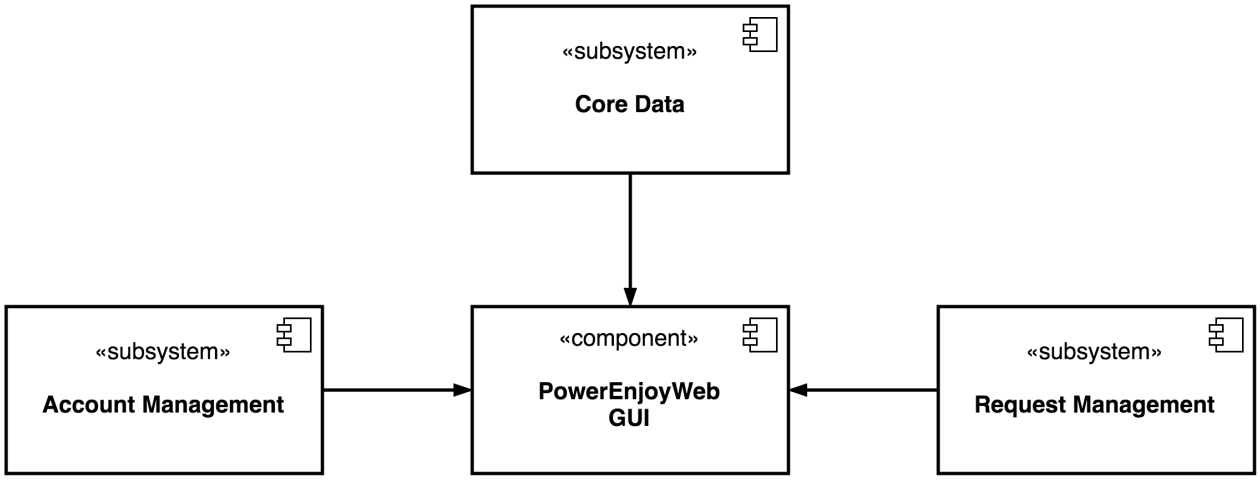

Figure 11: PowerEnjoyWeb GUI

2.4.2 Subsystem Testing

This section shows the order in which subsystems will be integrated.

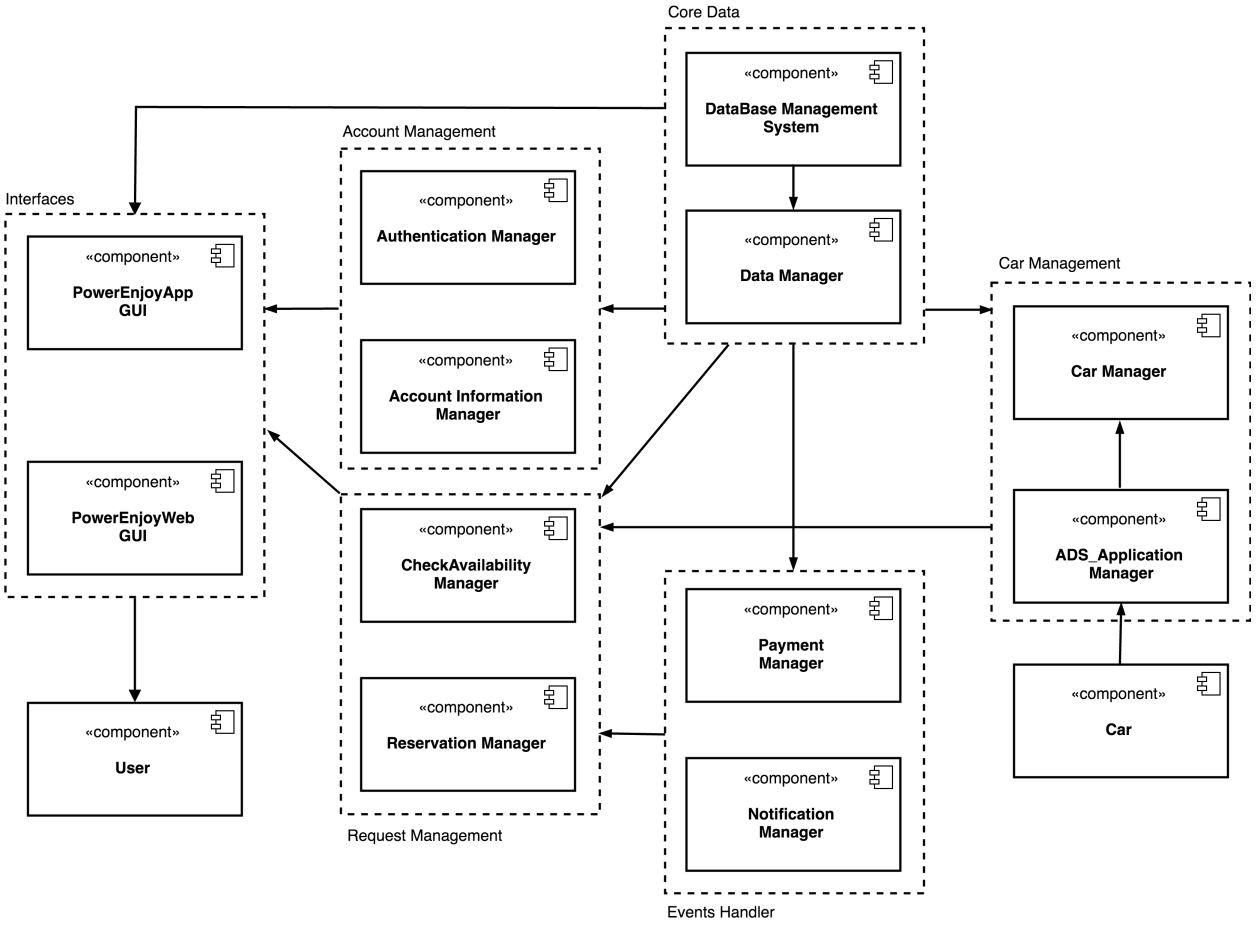

Figure 12: Subsystem Integration Sequence

3 Individual Steps and Test Description

This section describes the type of tests that will be used to verify that the elements integrated perform as expected. For each subsytem will be given a view of the main functions implemented, with a short description of the working principles and the expected effects for different inputs. In the Sequence Diagrams shown in the Section 2.5 of the Design Document can be found a detailed representation of the components interaction.

3.1 Account Management

Main functions implemented by the **Authentication Manager**:

`register(UserData)`	
Description	This function creates a new `User` instance in the system with all the information provided by the user during the registration phase.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An incomplete / duplicate set of information about the new user	An `InvalidArgumentException` is raised.
A full set of information about the new user (*name*, *surname*, *address*, *email*, *username*, *password*, ...)	The user data is correctly inserted in the database.

`login(Credentials)`	
Description	This function allows any registered user to log into the system using his *username* and *password*.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Credentials` object (with the correct combination of *username* and *password* of the user)	The user is logged in.

`checkCredentials(Credentials)`	
Description	This function verifies the *username - password* combination is correct.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A `Credentials` instance with a wrong combination of *username* and *password*	Returns `false`.
A valid `Credentials` instance (with the correct combination of *username* and *password* of the user)	Returns `true`.

Main functions implemented by the **AccountInformation Manager**:

`removeUser(User)`	
Description	This function deletes the `User` from the system.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
An existent and valid `User` instance	The `User` is correctly removed.

`activate(User)`	
Description	This function restores the privileges of the `User`.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
An active `User` instance	An `InvalidArgumentException` is raised.
A deactivated `User` instance	The `User` is no more a deactivated `User`.

`deactivate(User)`	
Description	This function removes some of the privileges from the `User`.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A deactivated `User` instance	An `InvalidArgumentException` is raised.
An active `User` instance	The `User` is no more an active User.

`editProfile(User, UserData)`	
Description	This function updates the existent `User` instance in the system with all the new information provided by the `UserData` object.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An incomplete set of information	An `InvalidArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A full set of information about the `User` instance (*address*, *password*, *billing information*, *license number*, ...)	The `User` data is correctly updated in the database.

`getHistory(User)`	
Description	This function allows a registered `User` to consult his complete requests history.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
An existent and valid `User` instance	Returns a list of all the requests of the `User`.

`enableMSO(User)`	
Description	This function enables the Money Saving Option for the `User`.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
An existent and valid `User` instance	The MSO of the `User` is enabled.

`disableMSO(User)`	
Description	This function disables the Money Saving Option for the `User`.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
An existent and valid `User` instance	The MSO of the `User` is disabled.

3.2 Car Management

Main functions implemented by the **ADS_Application Manager**:

`getStatus()`	
Description	This function asks the ADS to check the current `Status` (*available*, *reserved*, *in_use*, *unavailable*) of the car.
Input Specification	*Effect*
None	Returns the car `Status`.

`getDamages()`	
Description	This function queries the ADS to check the eventual car's damages through the sensors installed in there.
Input Specification	*Effect*
None	Returns a report with the eventual car's damages.

`getPosition()`	
Description	This function queries the ADS to check the current position of the car.
Input Specification	*Effect*
None	Returns the current `Location` of the car.

`getPassengers()`	
Description	This function queries the ADS the number of passengers actually inside the car.
Input Specification	*Effect*
None	Returns the number of passengers actually inside the car.

`checkPowerGrid()`	
Description	This function queries the ADS to check if the car is plugged into the power grid.
Input Specification	*Effect*
None	Returns `true` if the car is plugged into the power grid, `false` otherwise.

`lockDoors()`	
Description	This function queries the ADS the lock the car's doors and to change the `Status` of the car into *available*.
Input Specification	*Effect*
None	The doors are locked and the status of the car is updated into *available*.

`unlockDoors()`	
Description	This function queries the ADS the unlock the car's doors and to change the `Status` of the car into *in_use*.
Input Specification	*Effect*
None	The doors are unlocked and the status of the car is updated into *in_use*.

`updateCarStatus(Status)`	
Description	This function queries the ADS to change the car's `Status` into the one passed as argument.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Status` object	The car's status is updated according to the `Status` object.

Main functions implemented by the **Car Manager**:

`contactMaintainanceService(Car)`	
Description	Every time the Car Manager retrieve information about a car's status, if necessary, it will send a maintainance request to an external service through this function.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Car` object	A maintainance request to an external service is successfully sent for the specified `Car`.

`ping(Car)`	
Description	This function queries a `Car` to check whether it is online.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Car` object	A valid query is sent to the `Car`.

`carsInRadius(Location)`	
Description	This function retrieves all the cars near the `Location` passed as argument.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Location` object	Returns a list of *available* cars parked around the `Location`.

3.3 Requests Management SubSystem

Main functions implemented by the **CheckAvailability Manager**:

`getUserPosition(User)`	
Description	This function will retrieve the information about the user's position through the Google Maps API.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `User` instance	A `Location` object is returned representing the current position of the user.

`getAvailableCars(Location)`	
Description	This function will retrieve the needed information through the Car Manager and it will return a list of *available* cars near the user's `Location`.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Location` object	Returns a list of available cars parked near the `Location`.

Main functions implemented by the **Reservation Manager**:

`checkVerificationCode(User, int)`	
Description	This function verifies the `User` - *verification code* combination is correct.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A wrong *verification code* of the `User`	Returns `false`.
The correct *verification code* of the `User`	Returns `true`.

`getUserPosition(User)`	
Description	This function will retrieve the information about the user's position through the Google Maps API.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `User` instance	Returns a `Location` object representing the current position of the `User`.

`getReservableCars(Location)`	
Description	This function will retrieve the needed information through the Car Manager and it will return a list of *reservable* cars near the user's `Location`.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Location` object	Returns a list of *reservable* cars parked near the `Location`.

`startReservation(Car, User)`	
Description	This function creates a new instance of `Reservation`, tying together the `Car` and `User` instances passed as argument, and notifies it to the `User` through the Notification Manager.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
Valid `User` and `Car` instances	A new instance of `Reservation` is correctly created.

checkReservationStatus(Reservation)	
Description	This function retrieves all the information about the `Reservation` instance passed as argument, and shows them to the `User` through the Notification Manager.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Reservation` instance	Returns the current status of the `Reservation`.

endReservation(Reservation)	
Description	This function terminates the current `Reservation`, notifies it to the user through the Notification Manager and sends a payment request to the Payment Manager.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Reservation` instance	The state of the `Reservation` is changed into *terminated*, and the reservation is no more active.

startRent(Reservation)	
Description	This function creates a new instance of `Rent`, updating the `Reservation` instance passed as argument, and notifies it to the `User` through the Notification Manager.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Reservation` instance	A new instance of `Rent` is correctly created.

checkRentStatus(Rent)	
Description	This function retrieves all the information about the `Rent` instance passed as argument, and shows them to the `User` through the Notification Manager.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Rent` instance	Returns the current status of the `Rent`.

isTerminable(Rent)	
Description	This function will check through the Car Manager if all the conditions to end the `Rent` are respected (for example if nobody is still in the car, and so on..) and it will return `true` if the rent is terminable, `false` otherwise.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Rent` instance	Returns `true` if the rent is terminable, `false` otherwise.

endRent(Rent)	
Description	This function terminates the current `Rent`, notifies it to the user through the Notification Manager and sends a payment request to the Payment Manager.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Rent` instance	The state of the `Rent` instance is changed into *terminated*, and the rent is no more active.

3.4 Events Handler

Main functions implemented by the **Notification Manager**:

notify(User, Notification)	
Description	This function is used to notify a message to the user.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
Valid `User` and `Notification` instances	The notification message is correctly displayed to the user.

Main functions implemented by the **Payment Manager**:

`applyReservationFees(Reservation)`	
Description	This function applies the fees for a `Reservation` instance. Now the user can terminate a reservation for free, but in the future if PowerEnjoy would decide to apply a fee for the termination of a reservation, it would be easy to modify this value.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Reservation` instance	A payment request to an external service is sent for the `User` related to the current `Reservation`.

`calculateRentFees(Rent)`	
Description	This function calculates the fees for a `Rent` instance taking in account eventual discounts and other additional fees before sending the payment request.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Rent` instance	This function first updates the total cost of the `Rent` and then sends a payment request to an external service for the `User` related to the current `Rent`.

`calculateRentDiscount(Rent)`	
Description	This function will calculate the eventual discount for each `Rent` instance.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Rent` instance	The total cost of the `Rent` will be successfully updated according to the eventual discounts.

`calculateAdditionalFees(Rent)`	
Description	This function will calculate the eventual additional fees for each `Rent` instance.
Input Specification	*Effect*
A `null` parameter	A `NullArgumentException` is raised.
An invalid object	An `InvalidArgumentException` is raised.
A valid `Rent` instance	The total cost of the `Rent` will be successfully updated according to the eventual additional fees.

4 Tools and Test Equipment Required

This section presents all the **tools and test equipment** needed to accomplish the integration.

JUnit

JUnit is a simple, open source framework 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
- Test runners for running tests

JMockit

JMockit is open source library meant to be used together with JUnit's testing framework. It includes APIs for mocking, faking, and integration testing, and a code coverage tool.

Arquillian

Arquillian is a highly extensible testing platform for the JVM that ease the creation of automated integration, functional and acceptance tests for Java middleware.

Arquillian is an integration testing framework for business objects that are executed inside a container or that interact with the container as a client. It also integrates with JUnit, allowing tests to be launched using existing IDE, Ant and Maven test plugins. Arquillian shines by turning integration testing no more complicated than unit testing.

Cellular Phones

The application will be deployed also as a mobile application for customers using **Android** and **iOS**. In order to test these two radically different environments, at least two devices running these aforementioned OSes are needed. Specifically:

- Android phones will run **Kitkat** or superior
- iOS phones will run **iOS 8** or superior
- Each phone must have a **working Internet connection** and a **GPS** system enabled

PCs

In addition to the mobile application, a web app will be deployed. In order to test it, a personal computer must be used; several browsers are used, such as **Google Chrome, Mozilla Firefox, Internet Explorer / Edge** and **Safari**.

5 Program Stubs and Test Data Required

This section presents all the **program stubs** and **test data** needed to accomplish the integration.

Test Server

A working Glassfish test server is needed in order to properly host the Application server.

Test Database

The target environment must have a fully working and configured DBMS along with test data and tables reflecting the classes and the relations described in the ER diagram shown in the Design Document. This database will contain mixed valid and invalid data.

Fake GPS data

A set of fake GPS coordinates is needed to test both users and cars in our environment which will span inside and outside of mock cities.

Test e-mail confirmation

An email sender/receiver is needed in order to test and automate the email confirmation process when a user signs up for the service.

API client

It is also necessary to simulate a client application which interacts with the server through HTTP / HTTPS requests. A command-line application should suffice.

6 Appendix

6.1 Tools used

We used the following tools to produce this document:

- **LaTex** as typesetting system to write this document
- **LyX** as editor
- **Visio Professional** and **draw.io** to draw all the diagrams

6.2 Hours of work

Date	Colaci	De Pasquale	Rinaldi
28/12/16	1	/	1
29/12/16	3	3	3
30/12/16	1	/	/
31/12/16	/	3	3
3/1/17	/	1	3
4/1/17	/	3	/
6/1/17	3	/	1
7/1/17	5	5	5
8/1/17	3	1	1
9/1/17	5	3	/
10/1/17	1	1	2
11/1/17	2	1	3
14/1/17	1	2	1