Politecnico di Milano
A.Y. 2016-2017
Software Engineering II: “PowerEnJoy”

Integration **T**est **P**lan **D**ocument

version 1.0

January 15, 2017

Pozzati Edoardo (mat. 809392), Stefanetti Nadia (mat. 810622)

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this document, the Integration Test Plan Document (ITPD), is to describe how integration tests are to be performed in sufficient detail. Therefore, we intend to provide a guideline for the development team, in order to effectively verify that all of the components of PowerEnJoy are properly assembled and ensure that the unit-tested modules interact correctly.
The tests that will be presented here will focus on the modules themselves and afterwards on the flow of information between different modules.
Finally, this document also includes determining the adopted methodologies, which tools are needed and will be used during the whole process, as well as the required stubs, drivers and data structures that will be useful during said process.

1.2 Scope

PowerEnJoy is a car-sharing service based on mobile and web applications that exclusively employs electric-cars and allows users to easily find a car, reserve it and use it. The main accent is to optimize the access of users to the system and guarantee the support the management of the electric cars, as well as the booking and payments for the service itself.
The application logic must be designed and allocated into components that should improve software maintainability and ease future extensions.

1.3 Overview

In the second section firstly the criteria that must be met before integration testing may begin will be specified (e.g., functions must have been unit tested), secondly the components to be integrated are identified, thirdly the integration testing approach and the rationale for choosing it will be described and finally , for each subsystem, we will identify the sequence in which the software components will be integrated within the subsystem and the order in which subsystems will be integrated.
In the third section, for each step of the integration process presented before, we will describe in general the expected results of the test set.
In the fourth section, we will identify all tools and test equipment needed to accomplish the integration and explain why and how we are going to us them.
In the fifth section, based on the testing strategy and test design, we will identify any program stubs or special test data required for each integration step.

Finally, the last section includes appendices, where we will add information about the tool used to redact this document, the number of hours each group member has worked towards the fulfillment of this project, the glossary and the bibliography.

2 Integration Strategy

2.1 Entry Criteria

This section defines some entry criteria, i.e. the pre-requisites that must hold before the integration testing phase takes place, in order to ensure feasibility and correctness of the integration tests themselves.
First of all, we must have a code-complete project, all modules must be available and their performances and memory requirements have to fit with the specifications.
Secondly, all the classes and methods must pass thorough **unit tests** which should reasonably discover major issues in the structure of the classes or in the implementation of the algorithms. Every tested component must be bug-free and code-complete, i.e. all functionalities that are going to be tested must have already been completely implemented.
In order to assure a properly correct behaviour of the internal mechanics of the individual components, unit tests should have a minimum coverage of 90% of the code lines. In addition, test cases must be written with continuity and should be run at each consecutive build using J-Unit, to ensure that newly added lines do not interfere with the stability of the rest of the code. Unit testing is not in the scope of this document and will not be specified in further details.
Moreover, both automated data-flow and **code inspection** has to be performed on the whole project classes. This will ensure maintainability, respect of conventions and will reduce the risk that, during the integration test phase, any code-related issues or bugs rise, which could lead to complex problematic situations and therefore could increase the testers' effort in next testing phases. 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** of all classes and methods must be complete and up-to-date, in order to be used as a reference for integration testing development and to make it easier to reuse classes and understand their functioning. In particular, the public interfaces of each class and module should be well specified. Where necessary, a formal language specification of the classes' behaviours can be used (such as JML - Java Modelling Language).
In order to provide all the documentation that we need for proceeding in the succeeding steps, the following documents must be completed and delivered before integration testing begins:
• Requirement Analysis and Specification Document (**RASD**) of PowerEnJoy;
• Design Document (**DD**) of PowerEnJoy;

• Integration Testing Plan Document (this document).
Here we list the main functionalities that each subcomponent must fulfil:

Guest Controller	newRegistration()
	createPassword()

User Controller	submitUserLoginCredentials()
	viewHistoryRides()
	editProfile()
	getUserPosition()

Reservation Controller	createReservation()
	showPendingReservation()
	deleteReservation()
	getNearbyAvailableCars()
	checkExpiration()

Map Controller	getNearbyCars()
	getPosition()

Car Status Controller	getInfoCar()
	setCarStatus()
	unlockCar()
	lockCar()
	isCarInSafeArea()

Ride Controller	startRide()
	endRide()
	setInfoFee()

Fee Controller	computeNewFare()

Notification Dispatcher	sendLicenseInformation()
	sendPaymentInformation()
	sendPaymentCharges()
	askHelp()

Operator Controller	submitOperatorLoginCredentials()
	selectOperationRequest()

Table 2.1: Main functionalities of each subcomponent.

2.2 Elements to be Integrated

In the following paragraph we're going to provide a list of all the components that need to be integrated together.
The integration test phase for the PowerEnJoy system will be structured based on the architectural division in tiers that is described in the Design Document, as well as the indication of the elements of which said components are composed of.
In the Design Document, we outlined four main high-level components, corresponding to the tiers of the system, that are going to be integrated in this phase:

Client Tier. The client tier consists of the Web Browser Application Client, the Mobile Application Client and the On-Board Computer Application Client. Because single clients must behave properly with respect to their internal structure, they have to be tested individually and then they have to be integrated with their respective tier.

Web Server Tier. This includes all the components in charge of the web interface and the communicates with the application server and the client web browser.

Application Server Tier. This component implements all the application logic, includes the data access components and communicates with the front-ends through the interface components towards external systems and clients. All its subcomponents must be tested individually before being integrated and afterwards all the interactions among internal logic subcomponents must be tested as well.

Database Tier. This component includes all the database structures that will be used for the management of the system and data storage; the data layer components are already developed to work properly when coupled together, and the only component to be integrated is the DBMS.

The system is built upon the interactions of many high-level components, each one implementing a specific set of functionalities. For the sake of modularity, each component is further obtained by the combination of several lower-level subcomponents.
Because of this software architecture, the integration process of our software will be performed in two steps:
1. one by one integration, inside the same component, of the different subcomponents (i.e. Java classes and Java Beans) that depend strongly on one another to offer the higher level functionalities of PowerEnJoy (see table in Section 2.1);
2. integration of the above specified higher level components, after having ensured a proper internal behaviour.

2.3 Integration Testing Strategy

The integration testing strategy that will be adopted is the so-called bottom-up approach.
This choice comes natural since we will start the integration testing from the lowest-level components, which are assumed to be already tested at a unit level and which do not depend on other subcomponents or only depend on already developed subcomponents.
There are several advantages brought by this strategy. Firstly, it is easier to create the test conditions individually for each subcomponent and thus the observation of test results is simpler. Secondly, starting at the bottom of the hierarchy means that the critical modules are generally built and tested first and therefore any errors or mistakes in these forms of modules are identified early in the process. In order to avoid waiting until all modules are complete, an incremental approach is fundamental because it takes into account that localizing the faulty components can sometimes be difficult and that some

interfaces could be missed easily during testing.
We decided to choose bottom-up over top-down because there are many components at the lower levels; this means that only few drivers, if any, are needed.
Moreover, the higher-level components outlined in section 2.2 are fairly independent and loosely coupled since they correspond to different tiers; they also communicate through well-defined interfaces (RESTful API, HTTP), so they will not be hard to integrate at a later time.
In this way it will be possible to limit the total amount of stubs needed to accomplish the integration, because the specific subcomponents do not use the general ones, so they do not require stubs.
In the integration testing phase we also selected the order in which the subcomponents are going to be analysed, following the critical-module-first approach. This strategy allows us to concentrate our testing efforts on the riskiest components first, in order to avoid issues related to the failures of the core functionalities and threats to the correct implementation of the whole PowerEnJoy system. By proceeding this way, we are able to discover bugs earlier in the integration progress and take the necessary measures to correct them on time.
It should be noted that the DBMS is a commercial component already developed and can thus be used directly in the bottom-up approach without any explicit dependency.
At this level of integration testing, the communication functionalities with external systems must be covered as well, especially considering the relevance of said interaction in the context of the application. With respect to this, stubs and drivers will be used appropriately.

2.4 Sequence of Component/Function Integration

As the system consists of several different parts interrelated one to the other, we decided to plan integration testing under two different points of view, in accordance to what was written in [3, p. 6].
In particular, Section 2.4.1 will identify, for each subsystem, the sequence in which the main software subcomponents will be integrated within the subsystem, as they were defined in the DD in the High Level Components section, whilst Section 2.4.2 will identify the order in which subsystems will be integrated.

2.4.1 Software Integration Sequence

The components are tested starting from the most to the least independent one. This gives the opportunity to avoid the implementation of useless stubs, because when least independent components are tested, the components which they rely on have already been integrated. The components are integrated within their classes in order to create an integrated subsystem which is ready for subsystem integration.

First of all, all transactions with the DB must be operative in order to proceed with all other integrations: for this reason, all modules interacting with the DBMS are tested first.

In order to do so, the DBMS will need a driver for each Entity Bean to carry out queries and verify their correctness on a “shell” database, containing a greatly reduced number of test information. Said test database will be structured based on the E-R schema that will be adopted for the final implementation of the data layer.
The Database subcomponents are easily grouped into five steps: CreditCard, License, Car, SafeArea and Operator are the only subcomponents with primitive data attributes so these make the first step; after that Fare, Registration and SpecialSafeArea can be tested; Registration unlocks User which, with Car, unlocks Reservation, at the same time also Relocation can be tested; subcomponent Ride terminates our Database integration.
The next step involves the integration of the Session Beans which take advantage of said Entity Beans and are in charge of accessing them in the final application.
The following figures (fig.2.1, fig.2.2) show the subcomponents respectively of the Database and the Application Server component of the PowerEnJoy system, with the numbers as indication of the order of integration.

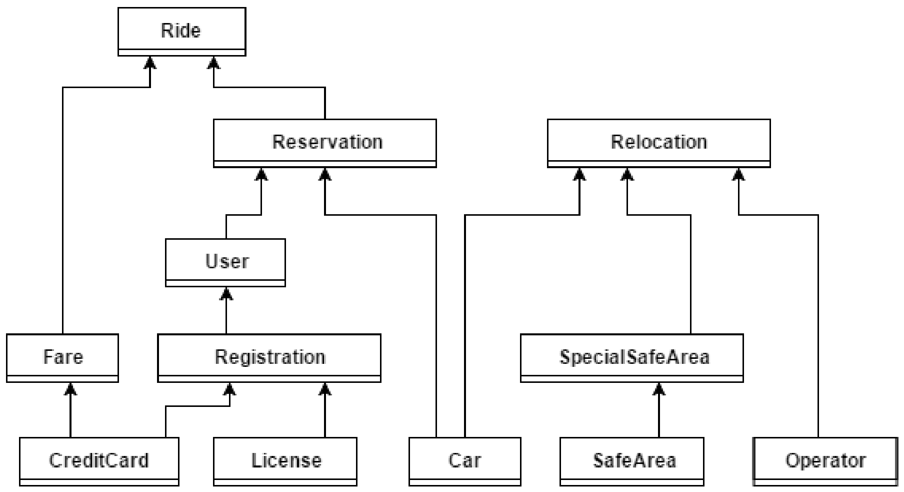
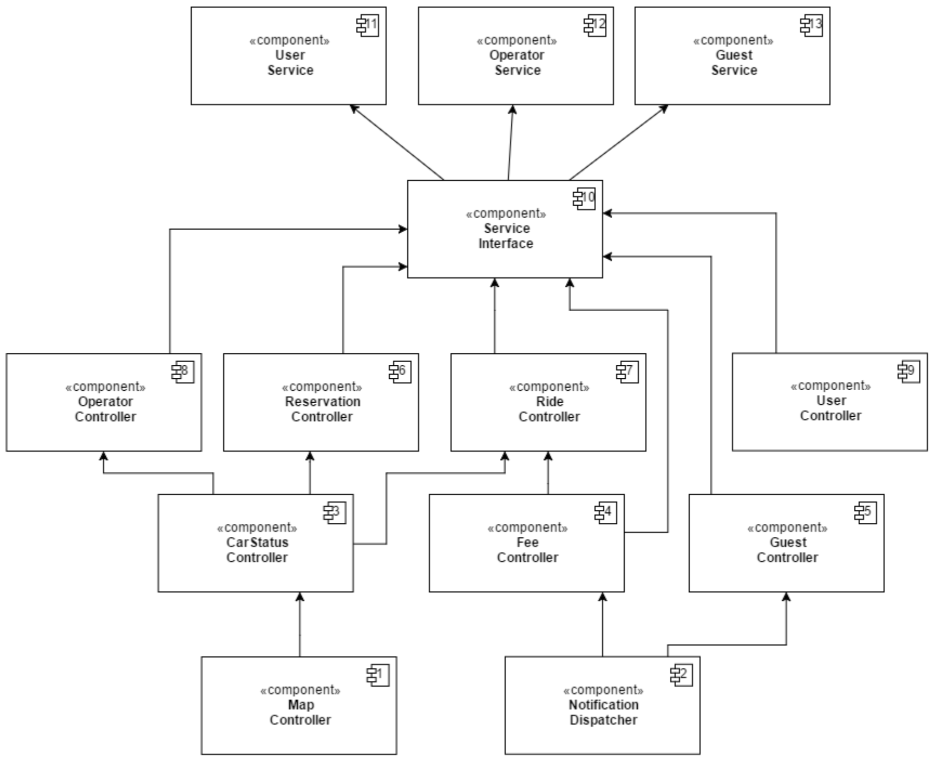


Figure 2.1: Integration order for the Database subcomponents.

The most critical features of the application revolve around the management of rides and reservations. Within this context, the most independent functionality is provided by the CarStatusController, since no other subcomponent depends on it apart from the already integrated MapController. In order to integrate the CarStatusController, there will be the need of two drivers: invoking methods in place of the ReservationController and of the RideController. In its turn, the RideController itself needs a driver in order to be integrated, which will represent the FeeController and call the methods exposed by RideManager in its place. FeeController, like GuestController, needs NotificationDispatcher's methods. Then we can integrate OperatorController (thanks to CarStatusController) and UserController to be ready to create a unique great integration into the ServiceInterface. Lastly we use ServiceInterface to customize an Interface for Guest, Operator and User.



«component» User Service
«component» Operator Service
«component» Guest Service
«component» Service Interface
«component» Operator Controller
«component» Reservation Controller
«component» Ride Controller
«component» User Controller
«component» CarStatus Controller
«component» Fee Controller
«component» Guest Controller
«component» Map Controller
«component» Notification Dispatcher

Figure 2.2: Integration order of the Application Server subcomponents.

N.	Subsystem	Component	Integrates with
I01	DataBase, AppServer	(JEB) CreditCard	DBMS
I02	DataBase, AppServer	(JEB) License	DBMS
I03	DataBase, AppServer	(JEB) Car	DBMS
I04	DataBase, AppServer	(JEB) SafeArea	DBMS
I05	DataBase, AppServer	(JEB) Operator	DBMB
I06	DataBase, AppServer	(JEB) Fare	DBMS (JEB) CreditCard
I07	DataBase, AppServer	(JEB) Registration	DBMS (JEB) CreditCard (JEB) License
I08	DataBase, AppServer	(JEB) SpecialSafeArea	DBMS (JEB) SafeArea
I09	DataBase, AppServer	(JEB) User	DBMS (JEB) Registration
I10	DataBase, AppServer	(JEB) Reservation	DBMS (JEB) User (JEB) Car

I11	DataBase, AppServer	(JEB) Relocation	DBMS (JEB) Car (JEB) SpecialSafeArea (JEB) Operator
I12	DataBase, AppServer	(JEB) Ride	DBMS (JEB) Fare (JEB) Reservation
I13	AppServer	(SB) MapController	(EXT) GoogleMapsAPI (JEB) SafeArea (JEB) SpecialSafeArea
I14	AppServer	(SB) NotificationDispatcher	(EXT) LicenseValidation (EXT) PaymentInformation (EXT) CallGateway
I15	AppServer	(SB) CarStatusController	(SB) MapController (JEB) Car (EXT) OnBoardApplication
I16	AppServer	(SB) FeeController	(SB) NotificationDispatcher (JEB) Fare (JEB) CreditCard
I17	AppServer	(SB) GuestController	(SB) NotificationDispatcher (JEB) Registration (JEB) User
I18	AppServer	(SB) ReservationController	(JEB) Car (JEB) Reservation (JEB) User (SB) CarStatusController
I19	AppServer	(SB) RideController	(JEB) Ride (JEB) Fare (JEB) Reservation (SB) CarStatusController (SB) FeeController
I20	AppServer	(SB) OperatorController	(JEB) Operator (JEB) Relocation (SB) CarStatusController
I21	AppServer	(SB) UserController	(JEB) User (JEB) Reservation
I22	AppServer	(SB) ServiceInterface	(SB) ReservationController (SB) RideController (SB) OperatorController (SB) UserController
I23	MobileClient	(I) MobileServices	(SB) ServiceInterface

Table 2.2: Integration order of the system subcomponents. External system interfaces to be integrated with PowerEnJoy's subsystems are marked with (EXT).

2.4.2 Subsystem Integration Sequence

Four main subsystems make up the whole architecture, listed in section 2.2. All components described in the previous section make up the DataBase and Application Server subsystem, the last one containing the main logic of the service; as stated before, the first important integration to be done is the one with the DBMS. The Application Server Tier comes before all kinds of clients, since a working business logic is mandatory to have properly functioning clients. The business tier, in fact, can be tested without any client, by making API calls also in an automated fashion. When the core system is ready, we will procede with the integration process from the server side towards the client applications, testing in conjunction the mobile application for users and operators with the main service. This choice of integrating the mobile app before the web tier is because the integration of the web tier and browser client is heavier and more complex. By integrating the mobile application before the web tier, we aim to obtain a fully operational client-server system as soon as possible. The web tier is less essential and can be integrated after the app.

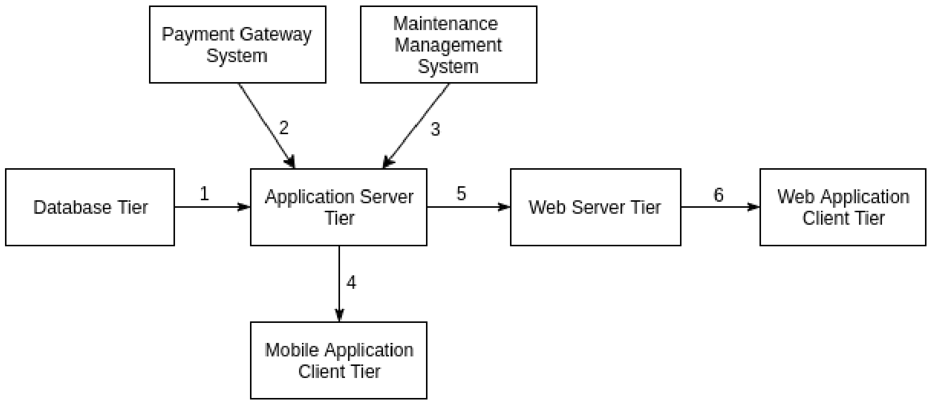

Payment Gateway System
Maintenance Management System
2
3
Database Tier
1
Application Server Tier
5
Web Server Tier
6
Web Application Client Tier
4
Mobile Application Client Tier

Figure 2.3: Graph representing the order of integration of the subsystems.

N.	Subsystem	Integrates with
SI1	Application Server Tier	DataBase Tier
SI2	Application Server Tier	(EXT) PaymentGateway
SI3	Application Server Tier	(EXT) MaintenanceManagement
SI4	Mobile Application Client	Application Server Tier
SI5	Web Server Tier	Application Server Tier
SI6	Web Browser Client	Web Server Tier

Table 2.3: Integration order of the subsystems described in Section 2.2.

3 Individual Steps and Test Description

This chapter describes the individual test cases to be carried out. Each test case is identified with a code and is directly mapped with Table 2.2 for the integration among subcomponents and with Table 2.3 for the integration among subsystems.
Test cases whose code starts with **SI** are integration tests among subsystems; test cases whose code starts with **I** are integration tests among subcomponents.

3.1 Subcomponent Tests

From now on the following notation will be used: C1 → C2 indicates that C2 is necessary for C1 to work properly.

3.1.1 Integration Test Case I01, I02, I03, I04, I05, I06, I07, I08, I09, I10, I11, I12: components that integrate with the DBMS.

The following test cases concern the integration among the Java Entity Beans and the Database Tier running the DBMS.

Test Case Identifier	I01T1
Test Item(s)	CreditCard → DBMS
Input Specification	Typical queries on table CreditCard.
Test Case Identifier	I02T1
Test Item(s)	License → DBMS
Input Specification	Typical queries on table License.
Test Case Identifier	I03T1
Test Item(s)	Car → DBMS
Input Specification	Typical queries on table Car.
Test Case Identifier	I04T1
Test Item(s)	SafeArea → DBMS
Input Specification	Typical queries on table SafeArea.
Test Case Identifier	I05T1
Test Item(s)	Operator → DBMS
Input Specification	Typical queries on table Operator.

Test Case Identifier	I06T1
Test Item(s)	Fare → DBMS, CreditCard
Input Specification	Typical queries on table Fare.
Test Case Identifier	I07T1
Test Item(s)	Registration → DBMS, CreditCard, License
Input Specification	Typical queries on table Registration.
Test Case Identifier	I08T1
Test Item(s)	SpecialSafeArea → DBMS, SafeArea
Input Specification	Typical queries on table SpecialSafeArea.
Test Case Identifier	I09T1
Test Item(s)	User → DBMS, Registration
Input Specification	Typical queries on table User.
Test Case Identifier	I10T1
Test Item(s)	Reservation → DBMS, User, Car
Input Specification	Typical queries on table Reservation.
Test Case Identifier	I11T1
Test Item(s)	Relocation → DBMS, Operator, SpecialSafeArea, Car
Input Specification	Typical queries on table Relocation.
Test Case Identifier	I12T1
Test Item(s)	Ride → DBMS, Fare, Reservation
Input Specification	Typical queries on table Ride.
Output Specification	Our model of Database is ready to be useful for Controllers.
Environment Needs	RDBMS.
Test Description	The purpose of these tests is to check that the correct methods of the Entity Beans are called, and that they execute the correct queries to the DBMS.
Testing Method	Automated with JUnit.

3.1.2 Integration Test Case I13

Test Case Identifier	I13T1
Test Item(s)	MapController → GoogleMapsAPI, SafeArea, SpecialSafeArea
Input Specification	Calls to the GoogleMapsAPI methods to get location data of the Coverage Area and calls from

	MapController to SafeArea and SpecialSafeArea entities, in order to locate power grids and boundaries of SafeAreas based on input sets of coordinates.
Output Specification	Coverage Area location data and lists of values to be provided to other subcomponents needing such information shall be returned, or a meaningful error status.
Environment Needs	GoogleGeolocationAPI, JDBC.
Test Description	The purpose of the test is to check that our controller (MapController) can correctly get the position from the corresponding GoogleMapsAPI. Error statuses shall also be checked.
Testing Method	Automated with JUnit.

3.1.3 Integration Test Case I14

Test Case Identifier	I14T1
Test Item(s)	NotificationDispatcher → PaymentPermission, LicenseValidation
Input Specification	Calls to the PaymentPermission and LicenseValidation methods to check the validity of the credential.
Output Specification	A Boolean value of the validity shall be returned.
Environment Needs	External Motorization and Bank Database.
Test Description	The aim of the test is to check that the connection among these entities is enabled.
Testing Method	Manual.

3.1.4 Integration Test Case I15

Test Case Identifier	I15T1
Test Item(s)	CarStatusController → MapController, Car, OnBoardApplication
Input Specification	Calls to the MapController's methods: -getPosition() to retrieve information about the car's GPS -getNearbyCars(UserPosition) to obtain a set of cars nearby the submitted location. Calls from CarStatusController to methods of the OnBoardApplication and Car entity to update the car status information according to specific conditions.

Output Specification	These information are ready to be processed by ReservationController, OperatorController and RideController. The car status information updates must be correctly memorized and made persistent.
Environment Needs	Glassfish, JDBC, Linux, RESTful APIs.
Test Description	MapController should be able to return the correct GPS data in a universal and consistent format independently from the architecture. Every update to the status attribute of the Car entities performed upon requests from the CarStatusController should be applied correctly.
Testing Method	Automated with JUnit.

3.1.5 Integration Test Case I16

Test Case Identifier	I16T1
Test Item(s)	FeeController → NotificationDispatcher, Fare, CreditCard
Input Specification	Calls to the NotificationDispatcher's method sendPaymentCharges() to complete the payment of a ride (also a Fee).
Output Specification	This Controller is now ready to receive information from RideController.
Environment Needs	Glassfish, JDBC.
Test Description	NotificationDispatcher can return the correct payment information to compute the fare.
Testing Method	Automated with JUnit.

3.1.6 Integration Test Case I17

Test Case Identifier	I17T1
Test Item(s)	GuestController → NotificationDispatcher, Registration,User.
Input Specification	Calls from the GuestManager to the Notification-Dispatcher's methods sendLicenseInformation() and send PaymentInformation() to verify the validity. It also check on the Database that he is not already registered.
Output Specification	A new instance of Registration is returned. The target information must be updated and the change made persistent, and a notification e-mail must be sent to the corresponding user.

Environment Needs	Glassfish, JDBC, mocked e-mail sender and receiver.
Test Description	The purpose of the test is to check that our Controller is able to confirm the validity of all the credentials inserted by the Guest, create a new instance of Registration and create unique password that, connected with the Registration, create a new instance of User.
Testing Method	Automated with JUnit and Mockito.

3.1.7 Integration Test Case I18

Test Case Identifier	I18T1
Test Item(s)	ReservationController → CarStatusController, Car, Reservation, User.
Input Specification	Calls to the CarStatusController's method: -getInfoCar() to show information about battery, position, etc. -setCarStatus() to request updates to the status of cars.
Output Specification	New instance of Reservation can be created. Update to cars and reservation must be performed correctly and the changes must be made persistent. The requests of car status modification must be performed correctly in relation to the context in which they happen.
Environment Needs	Glassfish, JDBC.
Test Description	The target is to check that ReservationController is able to create a new Reservation using a (User, Car) tuple and behaves correctly during the car reservation process, with respect to generating and managing reservations and requesting car status updates to support reservation activities.
Testing Method	Automated with JUnit.

3.1.8 Integration Test case I19

Test Case Identifier	I19T1
Test Item(s)	RideController → CarStatusController, FeeController, Reservation, Fare, Ride
Input Specification	Calls to the CarStatusController's methods: -setCarStatus() to request updates to the status of cars; -unlockCar() to start a new ride;

	-lockCar() to terminate the ride; -getInfoCar() to get information for a possible discount/charge. Calls to the FeeController's method: -computeNewFare() to create a new updated fare to be applied.
Output Specification	A new instance of Ride is returned; the status update request for cars must be performed correctly based on the final conditions of rides.
Environment Needs	Glassfish, JDBC.
Test Description	The goal of the test is to verify the correct creation of a Ride with a (Reservation, Fare) tuple and to ensure a correct behaviour of the system in managing the rides and the consequent status variations of cars through the CarStatusController.
Testing Method	Automated with JUnit.

3.1.9 Integration Test Case I20

Test Case Identifier	I20T1
Test Item(s)	OperatorController → Operator
Input Specification	Usual calls to the methods of the Operator entity in order to match user authentication information with a provided set of test inputs;
Output Specification	The match between authentication information and values in the database is carried out correctly.
Environment Needs	Glassfish, JDBC.
Test Description	The purpose of this test is to check that the authentication process is performed correctly and all the required controls are carried out.
Testing Method	Automated with JUnit.

Test Case Identifier	I20T2
Test Item(s)	OperatorController → CarStatusController, Operator, Relocation.
Input Specification	Call to the getInfoCar() and isCarInSafeArea() method to get information about all cars that need assistance/relocation and setCarStatus() when the Operator starts/ends the operation.
Output Specification	A list of all needy cars is returned.
Environment Needs	Glassfish, JDBC.

Test Description	The aim is to check that this controller is able to create a list of cars and to create a new Relocation instance.
Testing Method	Automated with JUnit.

3.1.10 Integration Test Case I21

Test Case Identifier	I21T1
Test Item(s)	UserController → User, Reservation
Input Specification	Call from UserController to the methods of the User entitiy to get his login credentials or to edit his/her profile information according to user requests, and to the Reservation’s methods to get past reservation of the applicant.
Output Specification	The target information must be updated correctly and the change must be made persistent. An User can login, user's possible actions are returned.
Environment Needs	Glassfish, JDBC.
Test Description	The intent of this test is to check that every modification requested by a user to his/her personal information is performed correctly and to prove that this controller is able to manage the user login and homepage.
Testing Method	Automated with JUnit.

3.1.11 Integration Test Case I22

Test Case Identifier	I22T1
Test Item(s)	ServiceInterface → ReservationController, RideController, OperatorController, UserController
Input Specification	Calls to the methods of all Controllers to group these Controllers into a one AppServer.
Output Specification	Create an Interface ready to be customize for each architecture.
Environment Needs	Glassfish, RESTful APIs.
Test Description	The goal is to join all the application functionalities and create an Interface for the clients. Multiple requests for one specific Session Bean have to be simultaneously carried out in order to avoid concurrency troubles during the system usage.
Testing Method	Automated with Arquillian and JUnit.

3.1.12 Integration Test Case I23

Test Case Identifier	I23T1
Test Item(s)	MobileServices → ServiceInterface
Input Specification	Calls to the Interface's methods to create a mobile interface. Typical requests of functions used by the application components to the application controller in order for it to forward appropriately the requests to the correct underlying Mobile Application component.
Output Specification	An Interface ready to be customize for each client (Operator, User, Guest).
Environment Needs	RESTful APIs, Cordova.
Test Description	The test aims is to check that MobileServices is able to fit to each client.
Testing Method	Automated with Android and iOS testing suites. Manual testing on physical devices. .

3.2 Subsystem Tests

3.2.1 Integration Test Case SI1

Test Case Identifier	SI1T1
Test Item(s)	Application Server Tier → Database Tier
Input Specification	Calls to the methods of the JPA Entities, that are mapped on tables in the Database tier.
Output Specification	The Database Tier shall respond by doing the correct queries on the Test Database. It must also react in the right way both if the requests are made correctly and if they come from unauthorized sources that are maliciously trying to access the data.
Environment Needs	Complete implementation of the Java Entity Beans, Java Persistence API, the Test Database and all drivers that calls the methods of the Java Entity Beans.
Test Description	The response will be compared with the expected output of the queries. Several components in the application server use DB for their functionalities, by either inserting, updating or deleting information. For more information, see the first tests in section Section 3.1
Testing Method	Automated with JUnit.

3.2.2 Integration Test Case SI2

Test Case Identifier	SI2T1
Test Item(s)	Application Server Tier → (EXT) PaymentGateway System
Input Specification	Invocations of methods provided by the payment gateway API.
Output Specification	The methods are properly invoked with the right parameters.
Environment Needs	Stub of the Payment Gateway endpoint to record dummy transactions and report success/failure. Application Server Tier fully developed up to the PaymentPermission functionalities.
Test Description	The Application Logic Tier shall correctly invoke methods offered by the Payment Gateway endpoint, which is replaced by a proper stub.
Testing Method	Automated with Mockito and JUnit.

3.2.3 Integration Test Case SI3

Test Case Identifier	SI3T1
Test Item(s)	Application Server Tier → (EXT) Maintenance-Management System
Input Specification	Methods invocation to establish a bidirectional communication between the two systems.
Output Specification	The requests and responses are carried out correctly over the established communication.
Environment Needs	Driver and stub simulating the behaviour of the MaintenanceManagement System endpoint to call methods offered by the dedicated RESTful API and to receive intervention requests from the PowerEnJoy system over the same API. Application Server Tier fully developed up to the CarStatusController functionalities.
Test Description	The RESTful API methods needed to establish a communication between the two systems are called and a series of request/responses is performed and compared with the expected results.
Testing Method	Automated with Mockito and JUnit.

3.2.4 Integration Test Case SI4

Test Case Identifier	SI4T1
Test Item(s)	Mobile Application Client → Application Server Tier
Input Specification	Typical RESTful API calls (both correct and intentionally invalid ones) to the Application Server Tier.
Output Specification	The Application Server Tier shall respond accordingly to the API specification. Also, it must react correctly if the requests are malformed or maliciously crafted.
Environment Needs	Complete implementation of the Application Server Tier; RESTful API client (driver) that mocks the actual Mobile Application Client.
Test Description	The clients should make typical API calls to the Application Server Tier; the responses are then evaluated and checked against the expected output. The driver of this test is a standard RESTful API client that runs on Java.
Testing Method	Automated with JUnit.

Test Case Identifier	SI4T2
Test Item(s)	Mobile Application Client → Application Server Tier
Input Specification	Multiple concurrent (typical, correct) requests to the RESTful API of the Application Server Tier.
Output Specification	The Application Server Tier must answer the requests in a reasonable time with the applied load.
Environment Needs	Glassfish, fully functional and developed Application Server Tier, Apache JMeter.
Test Description	This test case assesses whether the Application Server Tier fulfills the performance requirement stated in the RASD [1].
Testing Method	Automated with Apache JMeter.

3.2.5 Integration Test Case SI5

Test Case Identifier	SI5T1
Test Item(s)	Web Server Tier → Application Server Tier
Input Specification	Requests for services offered by the Application Server Tier, both valid and invalid ones.
Output Specification	The Web Server Tier must call the proper RESTful

	APIs or report an error if the requests were blocked or not recognized.
Environment Needs	Glassfish, fully developed Application Server Tier.
Test Description	This test has to ensure the right translation from HTTPS requests into RESTful APIs calls, reporting errors when needed.
Testing Method	Automated with JUnit.

Test Case Identifier	SI5T2
Test Item(s)	Web Server Tier → Application Server Tier
Input Specification	Multiple concurrent API calls to the Application Server Tier.
Output Specification	Web requests should be served without problems when a reasonable load is applied on the Application Server Tier.
Environment Needs	Glassfish, fully functional and developed Application Server Tier, Apache JMeter.
Test Description	This test case assesses whether the business tier fulfills the performance requirement stated in the RASD [1].
Testing Method	Automated with Apache JMeter.

3.2.6 Integration Test Case SI6

Test Case Identifier	SI6T1
Test Item(s)	Web Application Client → Web Server Tier
Input Specification	Typical and well-formed HTTPS requests from web client browser; incomplete, malformed and maliciously crafted requests.
Output Specification	If the requests are valid, then the Web Server Tier shall display the requested pages; if the requests are invalid, then it shall display a generic error message.
Environment Needs	Glassfish, fully developed Web Server Tier, HTTP driver to simulate the behaviour of a client web browser.
Test Description	This test should emulate HTTPS requests from typical users of the service and also incorrect requests.
Testing Method	Automated with JUnit.

Test Case Identifier	SI6T2
Test Item(s)	Web Application Client → Web Server Tier
Input Specification	Multiple and concurrent requests to the Web Server Tier.
Output Specification	Web pages should be provided without problems when a reasonable load is applied on the Web Server Tier.
Environment Needs	Glassfish, fully developed Web Server Tier, Apache JMeter.
Test Description	This test case assesses whether the Web Server Tier fulfills the performance requirement stated in the RASD [1].
Testing Method	Automated with Apache JMeter.

4 Tools and Test Equipment Required

To make sure that this product will be on market without unexpected behavior and that it can be reliable, two types of testing will be done before final commercial release: automatic and manual.

The objective of automated testing is to simplify as much of the testing effort as possible with a minimum set of scripts. Automated testing tools are capable of executing tests, reporting outcomes and comparing results with earlier test runs. They will be used when need to execute the set of test cases tests repeatedly. This type of testing will be very useful to catch regressions in a timely manner when the code is frequently changed and also will be carried out simultaneously on different machine with different OS platform combination.
For this project, in order to test the various components of PowerEnJoy in a more effective way, we provide the following software tools which will be used to automate the integration testing:

Apache JMeter. JMeter is an open source software tool which is used to test the performance of our subsystems: on the Web Tier, it will be used to simulate a heavy load, in order to check if the requirements on the maximum number of simultaneously connected users and on the response times stated in the RASD are respected. On the Application Server Tier it will be used to simulate a heavy load on the REST API. Please note that a stress test on the web tier as described before can also overload the business tier; tests on both sides are useful to identify the bottlenecks.

JUnit. It is the most used framework to write repeatable tests in Java. We plan to use it for unit tests of the single components (not covered by this document), but it will be also used to do integration testing together with Mockito and Arquillian, allowing us to verify that the interactions between components are producing the expected results. In particular, we are going to use it in order to verify that the correct outputs are returned after a method invocation, that appropriate exceptions are raised and caught when invalid parameters are passed to a method and other issues that may arise when components interact with each other.

Mockito. This is an open-source test framework useful to generate mock objects, stubs and drivers. We use it in several test cases to mock stubs and drivers for the components to test.
Since complex environments must be described within this project (for

example all interactions between a module and the DBMS), a mock framework is necessary. Among several and similar products (JMock, EasyMock, Powermock...), we decided to use Mockito for its simplicity and clearness.

Arquillian. It is an integration testing framework which empowers the developer to write integration tests for business objects that are executed inside a container or that interact with the container as a client. We mainly use it to manage the containers' integration with JavaBeans and to check that the interaction between a component and its surrounding execution environment is happening correctly (as far as the Java application server is involved).
As it combines a unit testing framework (JUnit), and one or more supported target containers (Java EE container, servlet container, etc), it provides a simple, flexible and pluggable integration testing environment.

Finally, as stated at the beginning of this chapter, some of the planned testing activities will also require a significant amount of manual operations, especially to devise the appropriate set of testing data.
The main goal of manual testing is to make sure that the application under test is defect free and software application is working properly accordingly to the requirement specification document (RASD). In this type, tester takes over the role of end user and test the Software to identify any unexpected behavior or bug before the actual release.

5 Program Stubs and Test Data Required

As mentioned in Section 2.3, the integration testing strategy that will be adopted is the so-called bottom-up approach.
Following this criteria and without having developed the entire system first, we need to define and use drivers that will take place of the software components that still do not exist, so to have a complete environment and actually perform the necessary method invocations.
Here follows a list of all the drivers, stubs and test data structures that will be developed as part of the integration testing phase, together with their specific role.

Test Database: in order to perform an operational test, it is quite important that the database is configured properly. The target environment must include a working and configured DBMS, in which test data and tables must reflect the entities and the relations described in the Class Diagram showed in the RASD document for the PowerEnJoy system [Section 3.5.2]. To test the mapping of JEBs over actual database entities, a mock DB should contain random valid and invalid data about Registration, User, Car, Fare, Reservation, Ride, etc for JavaEE, to allow exaustive tests on the data layer.

Drivers for the Java Entity Beans: they will be used to test the proper behaviour of the Java Entity Beans while the Application Server is not fully implemented. They call the relevant methods offered by the single JEBs to test the correctness of the queries.

Test Application Server: in order to properly host the Application Server, a working Glassfish test server is needed.

Drivers for the Session Beans: all the Session Beans used to implement the application logic need a dedicated driver in order to test their functionalities in an appropriate way. Each driver will invoke its methods and will be adopted to test its interaction with the subsiding subcomponents (all methods are grouped for each subcomponent in section 2.1). These drivers will be gradually replaced by other subcomponents, as specified in Section 2.4.1.

API client: in order to test the REST API of the Application Server Tier without the actual client application, it is necessary to emulate a simple client application with an API-client which interacts with the Application Server via HTTP requests. This driver needs to be scriptable in order for the tests to be automated.

Drivers for the Mobile Application APIs: in order to integrate the Mobile Client Tier with the Application Server Tier we will need drivers to simulate appropriate RESTful API clients. This is necessary in order to properly test the communication over the defined APIs that connect the client layers to the business logic one.

Mock e-mail sender and receiver: in order to test and automate the email confirmation process when a guest signs up for the service, before the actual deployment and release of the application. Both valid and invalid requests and e-mail must be processable by the service, that in turn must reply in a coherent way to ensure a proper behaviour of the system logic.

Drivers and Stubs for the interaction with External Systems: The interaction with the PaymentPermission and LicenseValidation must be tested using a stub, in order to verify that all the calls performed over the provided APIs are well crafted or rejected if that is not the case.
In order to test the interaction with the Maintenance Management System, both a driver and a stub will be needed. The stub will simulate the behaviour of the external system in case the system needs to signal an out-of-service car. The driver will test interactions triggered by the external system, like the requests of car status modification after intervention.

Appendix A: Used Tools

A.1 Microsoft Word 2013

To redact and format this document.

A.2 *g* it

To submit this document in the online repository.

A.3 Dropbox

Used as version control system in order to lead development.

A.4 Draw.io

To create and design the provided diagrams.

Appendix B: Hours of work

This is the time spent by each group member in order to redact this document:

- Pozzati Edoardo: ~15 hours
- Stefanetti Nadia: ~15 hours
- Total work time: ~30 hours

Appendix C: Changelog

These sections will be eventually redacted during future post-release updates in order to approach the ITPD modifiability providing a comfortable and higly effective way to trace changes.

Definitions

• Subcomponent: each of the low level functional unit realizing the functionalities of a subsystem.

• Subsystem (or Component): a high-level functional unit of the system.

Acronyms

- API: Application Programming Interface.
- AS: Application server.
- DB: Data Base.
- DBMS: Data Base Management System.
- DD: Design Document.
- GPS: Global Positioning System.
- ITPD: Integration Test Plan Document.
- JEB: Java Entity Beans.
- JEE: Java Enterprise Edition.
- JML: Java Modeling Language.
- OS: Operating System.
- RASD: Requirements Analysis and Specification Document.
- REST: REpresentational State Transfer.

It is an architectural style and an approach to communications often used in the development of Web services.

- RESTful: a REST-compliant system.
- SB: Session Beans.

Bibliography

The indications provided in this document are based on the ones stated in the previous deliverables for the project:

[1] A.Y. 2016/2017 Software Engineering 2 - *Requirements Analysis and Specification Document* - Pozzati Edoardo, Stefanetti Nadia.

[2] A.Y. 2016/2017 Software Engineering 2 - *Design Document* - Pozzati Edoardo, Stefanetti Nadia.

Moreover it is based on the specifications concerning the RASD assignment [3]

[3] Luca Mottola and Elisabetta Di Nitto, *Software Engineering 2: Project goal, schedule and rules*, 2016.

and on the provided ITPD examples [4] and [5]:

[4] A.Y. 2015/2016 Software Engineering 2 Project – *MyTaxiService Integration Test Plan Document* – Casati Fabrizio, Castelli Valerio.

[5] SpinGrid Project - *Integration Test Plan.*