Content

1. Introduction

1. Purpose and Scope

In our project, the Integration Testing Plan Document is an important part, which can guarantee all the subcomponents be integrated consistently.

The main purpose of this document is to make up the system when the subcomponents need to be revised according to the user's requirements or other unexpected reasons. The Integration Testing Plan Document can give us a clear and simple way to organize all the testing of subcomponents. We will provide these sections as follows: - To introduce the integration testing subsystems and their subcomponents involved in the integration activity. - The Elements to be Integrated and their entry criteria. - A description of the Integration Testing Strategy. - The Sequence of Components needs to be integrated, including Software Integration Sequence and Subsystem Integration Sequence. - Individual steps and test description and their input data and the expected output. - A description of performance analysis. - All the tools used in testing, and a description of the operating environment that perform all the tests.

2. Definitions and Abbreviations

1. Definitions

- Subsystem: a single, pre-defined work environment and a high-level functional unit of the system.
- Subcomponent: Implementation of subsystem functions

2. Abbreviations

- RASD: Requirements Analysis and Specification Document
- DD: Design Document
- ITPD: Integration Test Plan Document
- API: Application Programming Interface
- GUI: Graphical User Interface
- DBMS: Database Management System
- GPS: Global Positioning System
- SDK: Software Development Kit
- Req: Requirement
- App: Application

3. Reference Documents

- Requirements Analysis and Specification Document
- Design Document

- Assignments AA 2016-2017
- Integration testing example document
- IEEE standard on requirement engineering

2. Integration Strategy

1. Entry Criteria

For doing this part we should ensure that all units of project work can commence to achieve the corresponding unit test quality objectives and output the corresponding test report.

Besides, the following documents should have been completed, reviewed, approved to do unit testing phase, for instance, the Requirements Analysis and Specification Document and the Design Document. W.R.T integration testing phase, we assume that the project has already being code-complete. And there are no missing features or media elements. The product satisfies the performance and memory requirements specified by the Functional Spec. All priority bugs have been fixed and closed. Internal documentation has been updated to reflect the current state of the product.

Also, we should as far as possible to keep the percentage of completion of every component with respect to its functionalities as:

- 100% for the Data Persistence component
- At least 90% for the **Car Management** subsystem
- At least 90% for the **Reservation and Billing Management** subsystem
- At least 80% for the **System Administration and Account Management** subsystems
- At least 70% for the **Client Application**

2. Elements to be Integrated

As we presented before in the design document our system is composed by many components and units. So in this section, we will list all the main components which will be integrated in this phase. And we concern the integration phase as two levels of abstraction: - High-level components integration testing - Lower-level components integration testing

For high-level integration testing, as we introduced 3-tier structure to build our system(PowerEnjoy) in Design Document so we will follow the structure to process the testing, 1. presentation tier (mobile client, web client, web server), 2. logic tier (application server), 3. persistence tier (DB server).

At lower-level integration testing, we decided to integrate those components which are highly depending on one another to offer the higher level functionalities of PowerEnjoy. In this case, these components will be involved : Ride

controller, Bill controller, Reservation controller, Economic controller. And we assume that the car controller and charge station and the interaction between car and charge station will be well integration tested by third part.

3. Integration Testing Strategy

The goal of integration testing is to uncover errors that may arise when two or more components are integrated together. As stated before, unit tests are performed and validated before conducting the integration tests. The latter provides assurance that bugs arise from the integrations and not from the components (units) themselves.

We opted for a bottom-up strategy because of many reasons. First, we are using a lot of already-developed or commercially available components that interact directly with our low level components. Since those components are already tested, they represent reliable building blocks to integrate from. Second, using a bottom-up approach removes the need to develop stubs that are needed in a top-down approach. In addition to that, our main datasources (database and car parameters) are already tested which removes the need for stubs. Furthermore, the low-level modules are the most critical in our application (model, localization and car communication) and almost all other components depend on them. So the bottom-up approach ensures that these components are integrated first making sure that they are bug-free earlier. In case errors are uncovered, they can be fixed early in the process.

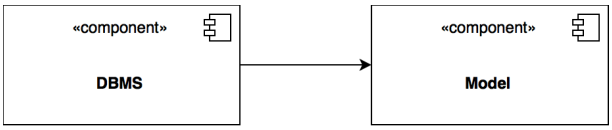
4. Sequence of Component/Function Integration

In the section below, we present the sequence of component integrations following the bottom-up approach described above. The arrows describe the dependencies; an arrow going from component A to B means that component A depends on component B.

The model is present in all the integrations because it is an important component as all the other components use it to persist data.

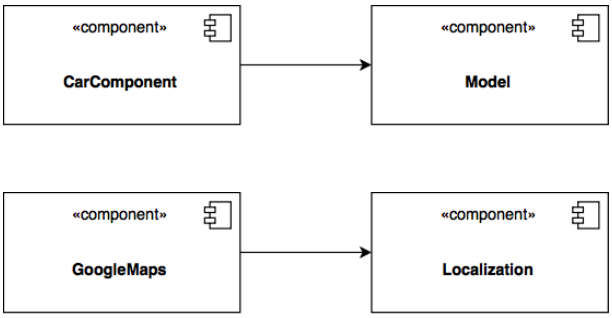
Model

Since the model is the most depended-on component of the system, we will start the integration test with it. In this test, we have to make sure that all operations (Create, Read, Update and Delete) are successfully transfered to the DBMS and applied in the database. This test checks also the data integrity.



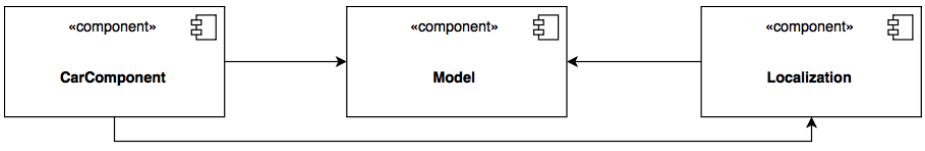
Google Maps & Car Component

In the process of integrating external components, the next step is integrating the Car Component and the Google Maps Component with the model and Localization Component respectively. The goal is to have these integrations before starting the integration of components we developed. The Car Component is the component that is made available by the car maker to communicate with the car and get all the variables. The Google Maps component is the component used to the translation of GPS Coordinates into places and also other localization concerns.



Car management system

The Car Management system takes care of gathering data about cars and persist it in the database. It is the system that makes sure of the constant communication with cars. It is composed of three elements: Car Component, Localization and Model. The Car Component and Localization should be integrated with the model to ensure data persistence. The Car Component use the Localization one to calculate distances and localizations.



3. Individual Steps and Test Description

1. External System

1. DBMS, Model

2. Google Maps Component, Localization

3. CarComponents, Model

2. Car Management System

1. Localization, Model

getLocalization(map)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	Return localization

getCarLocalization(carID, localization)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A inexistent carID	Throw InvalidArgumentValueException
A valid parameter	Return the current localization information of the car

2. Car Component, Model

getCarStatus(carID)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	Return the car's status and details

getReservatedCarStatus(carID, userID, reservationID)	
Input	*Effect*
A null parameter	Throw NullArgumentException
carID and userID and reservationID are invalid	Throw InvalidArgumentValueException
carID != reservation(info).carID	Throw InvalidArgumentValueException
Valid parameters	Return details and current status of the car to the user

isAvailableCar(carID)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	Return true or false

3. Car Component, Localization Component

updateCarLocalization(carID, localization)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A inexistent carID	Throw InvalidArgumentValueException
A valid parameter	Return the current localization infromation of the car

isCarInSafeArea(carID, localiaztion)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	Return true or false

3. Operations Management System

1. Reservation Component, Model

- insertReservation

insertReservation(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
By means of Model, A reservation with an id already existent in the database	An InvalidArgumentValueException is raised.
Formally valid arguments	By means of Model, an entry containing the reservation details inserted into the database.

- delete reservation

deleteReservation(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
By means of Model, A reservation with an id already existent in the database.	An InvalidArgumentValueException is raised.
Formally valid arguments	By means of Model, an entry containing the reservation data is deleted from the database.

- update reservation

updateReservationList(reservationList)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty array	A NullArgumentException is raised.
An array containing somenull values	An InvalidArgumentValueException is raised.
An array of non-null, but inexistent reservations	An InvalidArgumentValueException is raised.
An array of valid and existing reservations	By means of Model, the corresponding entries in the database are updated to set the reservation as completed.

2. Bill Component, Model

- update bill queues

updateBillQueues(billQueue)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing somenull values	A NullArgumentException is raised.
An array of non-nullqueues, but containing null values	An InvalidArgumentValueException is raised.
A non-empty array of valid bill queues	By means of Model, The content of the queues is updated in the database.

- get bill information

getBillInfo(UserId, ride)	
Input	*Effect*
A null user id	A NullArgumentException is raised.
An invalid user id	An InvalidArgumentValueException is raised.
A non-existing ride	A NullArgumentException is raised.
A valid user id and ride	By means of Model, Th bill information can be get from database.

3. Ride Component, Model

- startRide

startRide(carId, currentLocation)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A non-existing location	An InvalidArgumentValueException is raised.
A location far from the current location	An InvalidArgumentValueException is raised.
A current location	By means of Model, the corresponding location in the database are selected to start rides.

- updateRideInfo

updateRideInfo(CarId,RideInfoAvailable)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A non-existing CarId	An InvalidArgumentValueException is raised.
A set of valid parameters	By means of Model, the corresponding ride information in the database are updated

- getRideInfo

getRideInfo(CarId,RideInfoAvailable)	
Input	*Effect*
A non-existing CarId	An InvalidArgumentValueException is raised.
A set of valid parameters	By means of Model, Returns the stored ride information in the database

- endRide

endRide(CarId, currentLocation)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A non-existing CarId	An InvalidArgumentValueException israised.
A valid CarId and current-Location, the Car is on a ride and current Location is inside city	By means of Model, the ride is considered closed and is,nal-ized in the database.

4. Reservation Component, Car Component

- ReservationCar

ReservationCar(passengerId, passengerLocation, destination)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A passengerId not correctly formatted	An InvalidArgumentValueException is raised.
A passenger Location whose coordinates are invalid	An InvalidArgumentValueException is raised.
A destination whose coordinates are invalid	An InvalidArgumentValueException is raised.
A passengerLocation outside the city	An InvalidArgumentValueException is raised.
A valid set of parameters	A new reservation is created and handled

- existsAvailableCar

existsAvailableCar(reservation, location)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An inexistent location	An InvalidArgumentValueException is raised.
A zone with invalid elds	An InvalidArgumentValueException is raised.
A valid set of parameters	Returns true if a Car driver is available to serve the reservation, false otherwise.

5. Reservation Component, Bill Component

- checkBillInfo

checkBillInfo(reservation, ride)	
Input	*Effect*
A null parameters	A NullArgumentException is raised.
An invalid parameters	An InvalidArgumentValueException is raised.
A non-existing ride or reservation	A NullArgumentException is raised.
A valid reservation and ride	From the reservation and ride, the corresponding bill can be check by user

- ConfirmBillInfo

ConfirmBillInfo(reservation, ride)	
Input	*Effect*
A null parameters	A NullArgumentException is raised.
An invalid parameters	An InvalidArgumentValueException is raised.
A non-existing ride or reservation	A NullArgumentException is raised.
A valid reservation and ride	From the reservation and ride, the corresponding bill needs to be con rmed when the user check-out.

6. Reservation Component, Ride Component

- ride of reservation

RideOfReservation(CarDriver, reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A passenger location whose coordinates are invalid	An InvalidLocationException is raised.
An inexistent car driver	An InvalidArgumentValueException is raised.
A valid car driver and a reservation	Returns the reservation information and ride information of passengers location, destination and passengers number and whether share car or not.

- passenger Changed Destination

PassengerChangedDestination(CarDriver, reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A change with an invalid destination	An InvalidLocationException is raised.
A change with a valid destination	Returns the reservation information and ride information of new destination

- PassengerInterruptRide

PassengerInterruptRide(CarDriver, reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
Passengers don't want to go to the destination and asked interrupt ride immediately	Returns the current ride information
A change with a valid destination	Returns the reservation information and ride information of new destination

7. Bill Component, Ride Component

- CalculateBill

CalculateBill(ride)	
Input	*Effect*
A null parameters	A NullArgumentException is raised.
An invalid parameters	An InvalidArgumentValueException is raised.
A non-existing ride	A NullArgumentException is raised.
A valid ride	According to the formulas we write on DD the bill can be calculated

- GetStoredBillInfo

GetStoredBillInfo(UserId)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A location whose coordinates are invalid	A InvalidArgumentException is raised.
A non-existing car id	A NullArgumentException is raised.
UserId is valid	User can get the bill information which stored in the database

8. Ride Component, Localization Component

- MonitorCurrentLocation

MonitorCurrentLocation(CarId, location)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A location whose coordinates are invalid	An InvalidArgumentValueException is raised.
A non-existing car id	A NullArgumentException is raised.
CarId is valid, location is inside city	By GPS ,its status is set to available and its location is written in the database and monitor on the user application of his current ride.

- GetStoredRideRoute

GetStoredRideRoute(CarId,location)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A location whose coordinates are invalid	A InvalidArgumentException is raised.
A non-existing car id	A NullArgumentException is raised.
CarId is valid, location is stored in the database	From the location user can get the ride route exists in the database

9. Economic Component, Localization Component

- RemindEconomicInfo

RemindEconomicInfo(UserId)	
Input	*Effect*
A non-existing User id	A NullArgumentException is raised.
UserId is valid	System will remind the economic information when the user make a reservation and on his location who wants to share cars

- CalculateEconomicRide

CalculateEconomicRide(UserId, location)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A location whose coordinates are invalid	A InvalidArgumentException is raised.
A non-existing User id	A NullArgumentException is raised.
UserId is valid	The ride can be paid economically because of sharing cars, the percentageof discount can be calculate by the formulas we given on the DD

- MonitorEconomicRide

MonitorEconomicRide(UserId, location)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A location whose coordi - nates are invalid	A InvalidArgumentException is raised.
A non-existing User id	A NullArgumentException is raised.
UserId is valid	The economic ride information can be monitor on the userâ s application,

4. User Management System

1. User Component, Model

createUser(user)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	Insert the user's information into the DB

updateUserInfo(user)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	The user's information is updated in the DB

feedbackRequest(feedback)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	The user's feedback information is inserted into the DB

udateUserPassword (userID, password, newPassword)	
Input	*Effect*
A null parameter	Throw NullArgumentException
UserID dosen't match password	Throw InvalidArgumentValueException
A valid parameter	Update the user's password in DB

2. User Component, Car Component

unlockCar (userID, reservationInfo)	
Input	*Effect*
A null parameter	Throw NullArgumentException
UserID doesn't match the reservationInfo	Throw InvalidArgumentValueException
Valid parameter	Unlock the car

getAvailableCarList (userID, localization)	
Input	*Effect*
A null parameter	Throw NullArgumentException
An invalid localization	Throw InvalidArgumentValueException
A valid parameter	Return the availabe car list to the user

3. User Component, Bill Component

getBill(userID, ride)	
Input	*Effect*
A null parameter	Throw NullArgumentException
UserID dosen't match the ride information	Throw InvalidArgumentValueException
A valid parameter	Return the bill details to the user

checkHistoryBill(userID, date)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	Return the bill details to the user for those dates

4. User Component, Reservation Component

reservationRequest(userID, reservationInfo)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A non-null parameter	Return true

cancelReservation(userID, reservationID)	
Input	*Effect*
A null parameter	Throw NullArgumentException
A reservationID that doesn't exsit	Throw InvalidArgumentValueException which means the user didn't make a reservation
Valid userID and reservation ID	Remove the reservation from DB and add this operation into the log

4. Tools and Test Equipment Required

4.1 Tools

There are a ton of tools out there for automated testing, and just about any tool can be used for just about any testing methodology.

Testing the system is not how much on the technology we use, but on the difference between functional testing and system testing and on how we use tools in functional testing. Functional or feature testing is the process of testing newly written code to ensure it functions as designed. System or integration testing is the process of ensuring individual functional areas integrate (play nicely together). For what concerns the business logic components running in the Java Enterprise Edition runtime environment, we are going to take advantage of few tools to implement different functions in testing.

Mockito: Usually, Mockito helps us to cut all dependencies in unit tests, but it can be useful also in the integration tests where programmers often want to have all real applications layers.

In our case, we will use Mockito to do this testing on the main server and applications. To do so we want to have our integration tests, but without a real call to database. Because our dao layer is using some native queries to get data from database, and some embedded database doesn't support some query syntax.

Arquillian: A new testing framework developed at JBoss.org, 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 will apply it to execute tests against a Java container in order to check that the interaction between a component and its surrounding execution environment is happening correctly.

Rest-Assured: As we used RESTful architecture pattern in our system and it(REST-API) as one of our system's core methods to make sure the interaction between system and server goes well.

We will use Rest-Assured to test the implementation of REST-API instead of using JUnit which is usually used to test it by JAVA programmers. From some point of views the latter is a white box test, programmers can clearly know which class is being tested, which method is being used but all of this are not from users' view to test which API is being used. REST Assured is a simple Java library for testing of REST services, it can write code directly and initiate the HTTP request to the server and verify the returned result.

4.2 Equipment required

4.2.1 Server side

In order to ensure the stability and fluency of the system, we are going to use **B/S** model to develop our server terminal. From economy, scalable, extensible, easy to develop and easy to maintain points we propose to buy a subscription to **Amazon EC2** rather than a physical server.

4.2.2 Client side

For our system (PowerEnjoy) it will be applied on both mobile-phone terminals and PC terminals (browsers). In order to increase the user experience, we will as far as possible to realize that the system can be used in different operation systems and vary devices. So our system will support Android (2.0 later), IOS(7.0 later), WindowsPhone(8.0 later) operation systems. And it will be ensure that the system can be run on at least 80% of different sizes of display screens. As for browsers, we request Mozilla Firefox, v36 above, Microsoft Edge, Internet Explorer for Windows, v9.0 above, Google Chrome, v55.0, Opera, v40 above, and Safari, v5 above.

5. Program Stubs and Test Data Required

1 Program Stubs and components

As we said before, we are going to adopt a bottom-up approach to component integration and testing. In this part, the purpose of these stubs is to write on a log that they have correctly received the messages.

User component - This testing module will invoke the methods exposed by the user subcomponent, in order to test its interaction with the Model, the car component, the reservation component and bill component.

Car component - This testing module will invoke the methods exposed by the car subcomponent, in order to test its interaction with the Model, the reservation component, the localization component and the ride component and the Mapping component.

bill component

- this testing module will invoke the methods exposed by the bill subcomponent, in order to test its interaction with the reservation component, the user component and the economic component.

Ride component

- this testing module will invoke the methods exposed by the ride subcomponent, in order to test its interaction with the reservation component, the bill component and the economic component.

Economic component - this testing module will invoke the methods exposed by the ride subcomponent, in order to test its interaction with the the bill component and the localization component.

Model - this testing module will invoke the methods exposed by the Model, in order to test its interaction with all the other components and database.

Localization component - this testing module will invoke the methods exposed by the localization subcomponent, in order to test its interaction with the car component and model.

Reservation component - this testing module will invoke the methods exposed by the reservation subcomponent, in order to test its interaction with the the user component and car component.

2. Test Data

To test User component

In order to test the User component, we assume both valid and invalid users to exhibite the following problems: - Null username - Invalid username - Null password - Invaild password

To test Car component In order to test the Car component, we assume both valid and invalid users to exhibite the following problems: - A null parameter - A Invalid parameter - Driving license not compliant with the legal format

To test bill component - A null user id - A Invalid parameter - A non-existing ride or reservation

To test Ride component - An invalid user id - A non-existing car id - A null location - A non-existing location - UserID dosen't match the ride information

To test Economic component - A non-existing User id - A ride which is unfinished

To test Localization component - A null parameter - A non-existing location - A location far from the current location - A passenger location whose coordinates are invalid

To test Reservation component - A null parameter - An array containing somenull values - An array of non-null, but inexistent reservations

6. Effort Spent

Reda Aissaoui

- 08/01/2017 2h
- 11/01/2017 3h

Lidong Zhang

- 08/01/2017 2h
- 09/01/2017 3h
- 10/01/2017 3h
- 13/01/2017 4h
- 14/01/2017 3h