Software Engineering 2: PowerEnJoy

Integration Test Plan Document

Nardo Loris, Osio Alberto

Politecnico di Milano

1 Introduction

1.1 Revision history

Version	Author	Description
1.0	Nardo Loris - Osio Alberto	Initial release

1.2 Purpose and scope

This document describes the integration test plan for the PowerEnJoy car sharing management system. It aims at providing a detailed and exhaustive description of the steps involved in integration testing, along with entry condition and a description of test data.

Integration test is a very important activity as it aims at checking the communication between components, and in particular with components not actually developed by our company, as to verify that all the services implemented inside the various components actually fulfill the functional and non functional requirements.

This document is intended to be read mainly by developers and quality assurance team. In general, this document is to be read in the context described by the *Requirements Analysis and Specification Document* and by the *Design Document* written for this project.

1.3 Document structure

The content of this document is mainly organized in four sections:

- **Integration strategy** chapter provides detailed informations about the starting conditions required to begin integration testing, the approach chosen for integration, the components to be integrated and the integration sequence
- **Individual steps and test description** chapter provides informations about the tests to be performed in the different integration steps
- **Tools and test equipment** chapter deals with the tools that is planned to use in order to implement and run tests over the (partially) built system
- **Program stubs and test data required** chapter deals with the structure of the scaffolding to be developed in order to build a meaningful test environment

1.4 Acronyms, abbreviations and definitions

- **System**: the system to be developed for PowerEnJoy
- **User**: a generic person interacting with the system
- **Employee**: a person who works for PowerEnJoy
- **Actor**: can refer to both users and employees
- **Car**: an electric vehicle owned by the company
- **Bill**: an amount of money a user has to pay. It is related to only a single ride
- **Pending bill**: a bill that the user has not paid yet
- **Paid bill**: a bill that the user has already paid for
- **Area**: a space delimited by a polygonal line whose vertices are a set of geographical coordinates
- **Geographical coordinates**: a tuple of latitute and longitude describing a location on Earth
- **Geographical region**: an area where a user can reserve at most one car. They do not overlap
- **Safe area**: an area where it is possible to park a car and optionally to recharge it, in order to make it available for another user
- **Safe parking area**: a special safe area where it is not possible to recharge the car
- **Recharging station area**: a special safe area where it is possible to plug the car for recharging its battery
- **Registered user**: a user who has completed the sign up process
- **Logged user**: a user who has completed the log in process and has not yet started the log out process
- **Banned user**: a registered user who cannot reserve a car until all his pending bills are estinguished
- **Reservor user**: the user who has made a reservation for the specific car. A user is considered the reservor user of a car until the reservation expires or the user is charged with the bill
- **Available car**: a locked car for which no reservation exists

- **Reserved car**: a locked car for which it exists a user who has reserved it
- **Becoming available car**: an unlocked car is said to be "becoming available" as soon as all the passengers and the driver of this car exits the car, the doors of the car are closed and it is parked in a safe area
- **In maintenance car**: a locked car is said to be "in maintenance" as soon as its battery level is below 20%, the car cannot be reserved
- **In use car**: an unlocked car which is not becoming available
- **GPS**: A system capable of providing the location of a receiver device with a good precision (5 meters)
- **Overlapping areas**: Two areas are said to be overlapping if there exists at least one geographical coordinates which is contained inside the two areas
- **Expiration of a car reservation**: when a reservation expires, the car becomes available again, the reservor user loses his reservation and he is charged a fee of 1€
- **Percentage delta**: a discount or a raise based on percentage
- **Applying a raise** or **a discount**: The operation of increasing or reducing the amount of a bill for a specific reason. The amount is computed just before the system charges the user of a bill, and then all those amounts (each one related to a specific reason) are algebraically added to the same bill.
- **RASD**: Requirements Analysis and Specification Document
- **DBMS**: DataBase Management System
- **COTS**: Commercial Off The Shelf

1.5 Reference documents

- The project description document
- Requirement Analysis and Specification Document for PowerEnJoy car sharing system
- Design Document for PowerEnJoy car sharing system
- The Integration Test Plan Example document
- Testing tools
    - Mock server: http://www.mock-server.com
    - JUnit: http://junit.org/junit4
    - Arquillian: http://arquillian.org

2 Integration strategy

2.1 Entry criteria

This paragraph provides the criteria that have to be met in order to start the integration testing.

- Both the *Requirements Analysis and Specification Document* and the *Design Document* must be fully written and validated by the quality assurance team.
- The single components that go under integration testing must have been developed at least for the 90% of the functionalities they have to provide, and the developed part must be fully unit tested (a coverage of 95% is required)
- Drivers and stubs as of chapter 5.1 of this document have been fully developed
- Integration testing can start only when the estimated percentage of completion of every component with respect to its functionalities is:
    - 50% for *User Application* and *Employee Application*
    - 100% *Monitoring Controller*, *User Controller* and *Bill Controller*
    - 50% for *Router*
    - 80% for all other components

  Note that some components are required to be fully developed before integration testing starts because they are very basic components and are used by the majority of the other components. This reflects also the integration order defined by the strategy (see later).

  The choice of requiring different completion percentages is taken in order to anticipate as much as possibile the begin of the integration testing, as to speed up the overall system development.

2.2 Elements to be integrated

This paragraph provides a list of the components that need to be integrated together.

As specified in the *Design Document*, the system is made up of five main components, that are:

- Server
- User Application
- Employee Application
- DBMS
- Car monitoring system

These are the final, highest level components to be integrated together.

Before carrying out the full system integration, we must focus on lower level components. In particular, we have to focus on *server*, *user application* and *employee application*. The *DBMS* component is available as a COTS component, and thus it is a very well-known and well-tested component. The *car monitoring system* is provided by the manufacturer of the cars, and so the development and the testing of that component has not to be carried out by our company (we must test the integration at system level).

The most complex component is for sure the server component, which is made up of the integration of the following components:

- Router
- Monitoring controller
- Car controller
- Reservation controller
- Ride controller
- User controller
- Geographical areas controller
- Safe areas controller

- Bill controller

In the context of the *server* component, we define 3 main subsystems:

- **Core Service Subsystem**
  - Monitoring controller
  - Car controller
  - Reservation controller
  - Ride controller
- **Service Tuning Subsystem**
  - Safe areas controller
  - Geographical areas controller
- **End user subsystem**
  - User controller
  - Bill controller

The *router* cannot be included in any of those subsystem as it is unrelated with the functionalities those subsystems provide. It can only constitute a singleton subsystem.

The *User application* and the *Employee application* are not as complex as the server components as they represent only a user friendly interface that invokes the server APIs. In this context, integration testing is very limited because of the fact there is (almost) no interaction between subcomponents related to different APIs.

2.3 Integration testing strategy

The approach used in integration testing is a bottom-up approach.

Bottom-up approach is chosen for these reasons:

- It reflects the development process and avoids developing stubs of very simple components as the effort spent in developing the stub would not be so minor than the effort spent in developing the actual component to get a real advantage.
- Components like the *DBMS* or the *Car monitoring system*, which are bought from external companies, are immediatly available in a bottom-up approach without any explicit dependency
- It exercises real components instead of stubs, allowing the development team to have more precise feedback also on non functional requirements already during the integration testing, and so to detect and correct issues earlier (if a non functional requirement is violated during the integration testing phase then it is extremely unlikely it won't be violated also during the system testing)

2.4 Sequence of components / software integration

This section provides detailed specification of the order in which components must be integrated.

2.4.1 Software integration

This section describes the order of integration of subsystem to build a highest-level component. Because of already explained reasons we focus on the *server* component testing.

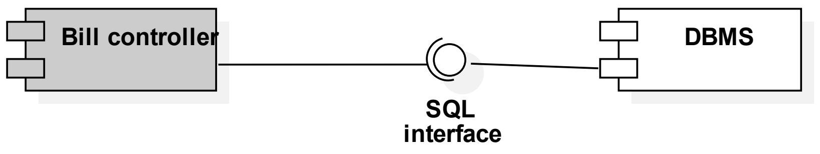
Stilistic notes:

- As we'll focus on one component at a time, that component will be highlighted in light grey.
- External systems will be highlighted in yellow

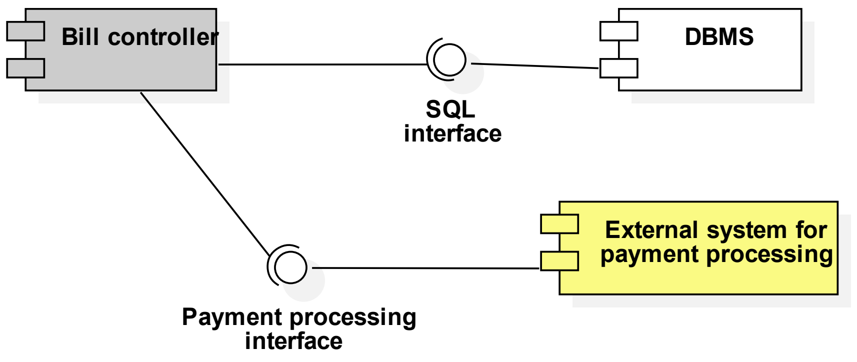
2.4.1.1 End user subsystem

According to the bottom-up integration approach, the first subsystem to be integrated is the *end user subsystem*. It only depends on the DBMS as an external component.

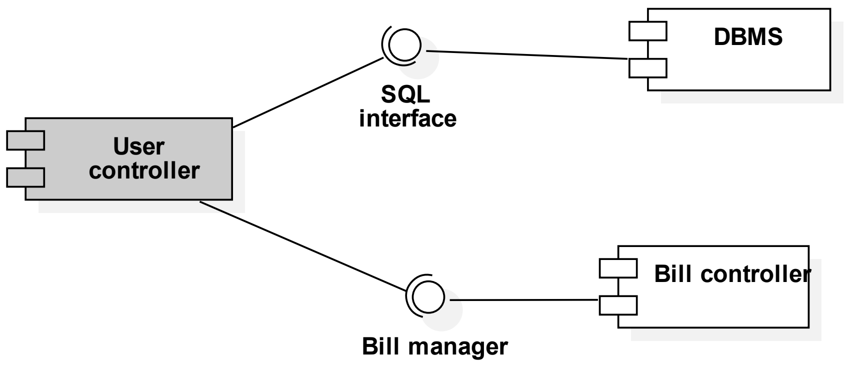
First of all, we test the integration between the *DBMS* and the *bill controller*.



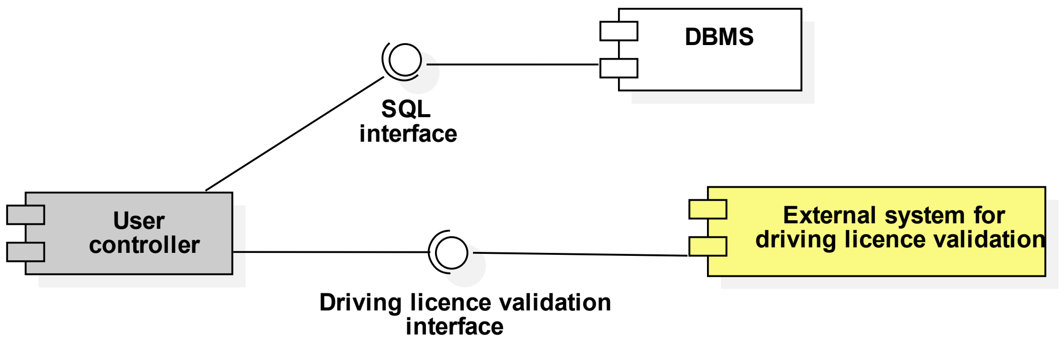
Then, we must test the interaction with the external system for payment processing


Bill controller
DBMS
SQL interface
External system for payment processing
Payment processing interface

Then we can add the *user controller*, which depends primarily on the *DBMS* and on the *bill controller*



Now we can focus on the interaction with the external components for driving licence validation

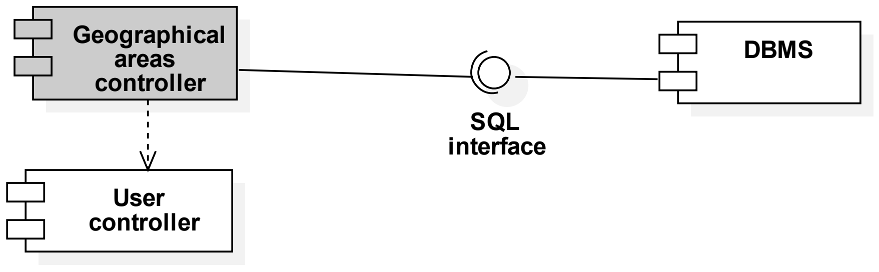

DBMS
SQL interface
User controller
External system for driving licence validation
Driving licence validation interface

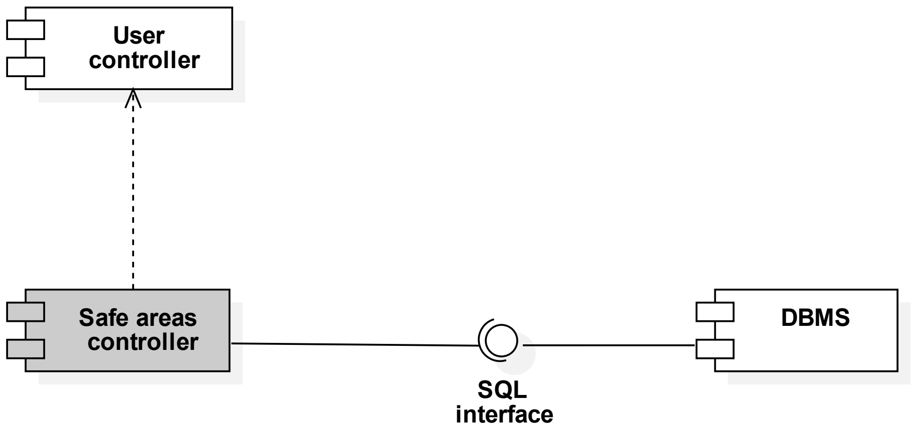
This completes the customer subsystem, which, according to the DD, provides the basic functionalities for user authentication, user management and billing.

2.4.1.2 Service management subsystem

Now we can perform integration testing on *service management subsystem*, that depends both on *DBMS* and *end user subsystem* (for user authentication and authorization).

The components of this subsystem are indipendent one from the other, so we can test integration of the two components with their dependencies separately.



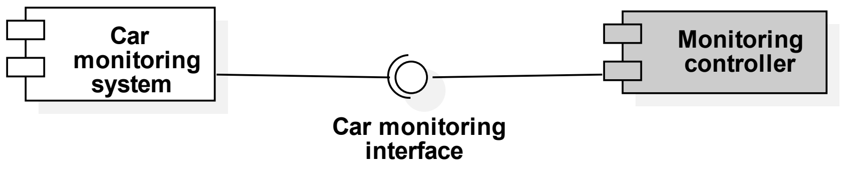


This subsystem provides functionalities related to the tuning of the service, such as geographical regions and safe areas management.

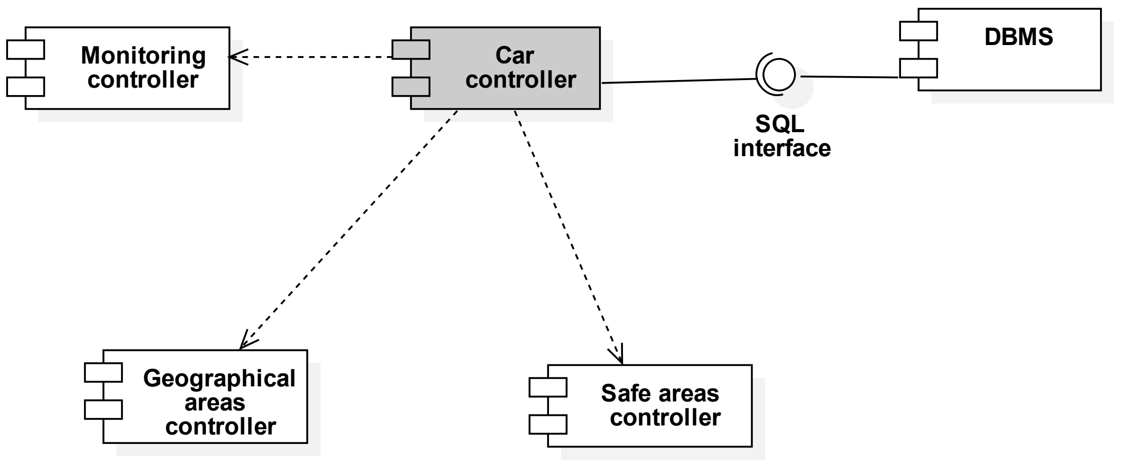
2.4.1.3 Core service subsystem

This is the most complex subsystem, and it is the last in the integration order as it depends on both the privious subsystems, plus the *DBMS*

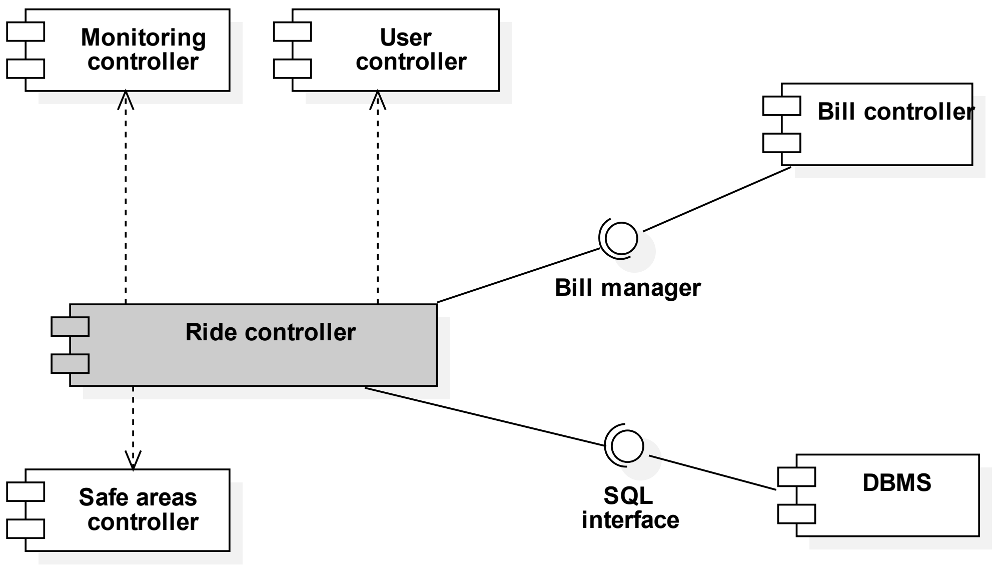
First of all, we exercise the integration with the external system that monitors cars onboard. We setup the message-driven architecture described in the DD integrating the broker



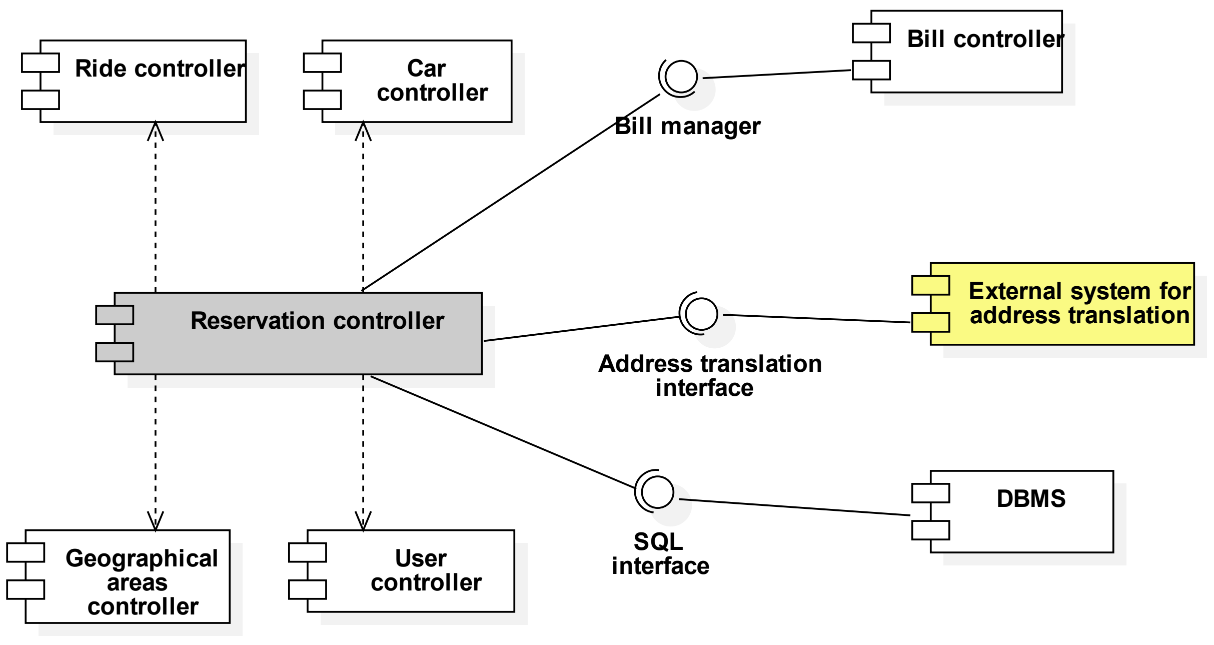
and then adding the *car controller* which provides informations about cars to all other components, interacting on one side in the message-driven architecture and on the other side with the DMBS in order to persist collected information, the safe areas controller in order to detect whether a car is parked in a safe area or not, and the geographical region controller, in order to associate each car with the geographical region it is parked in.



Now we can test integration of the two main business components. Due to the bottom-up approach and the dependency of reservations on rides, we must test first the integration of the *ride controller*

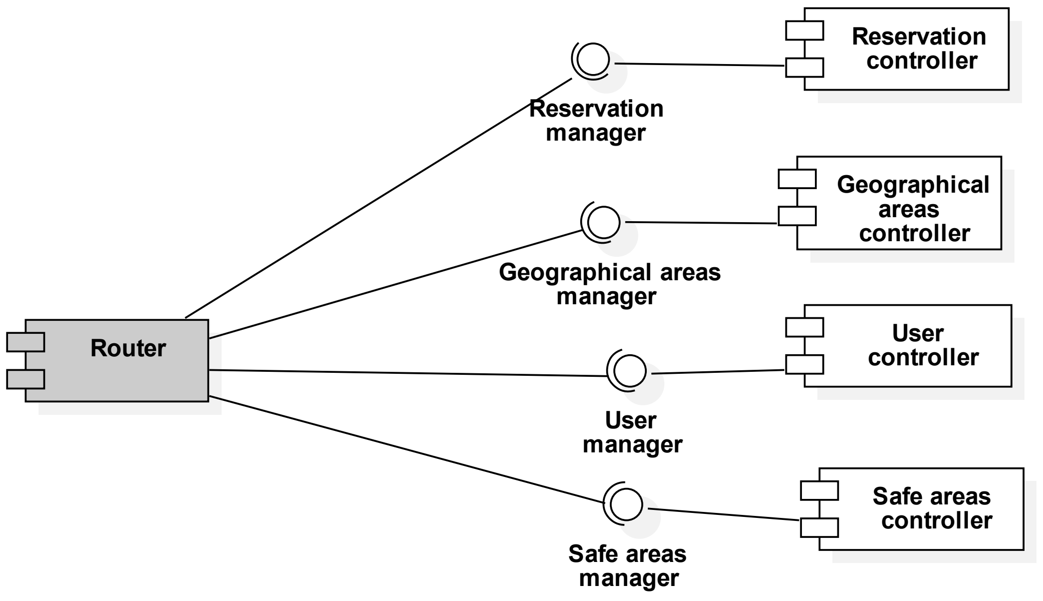


Then the last component of this subsystem is the *reservation controller*



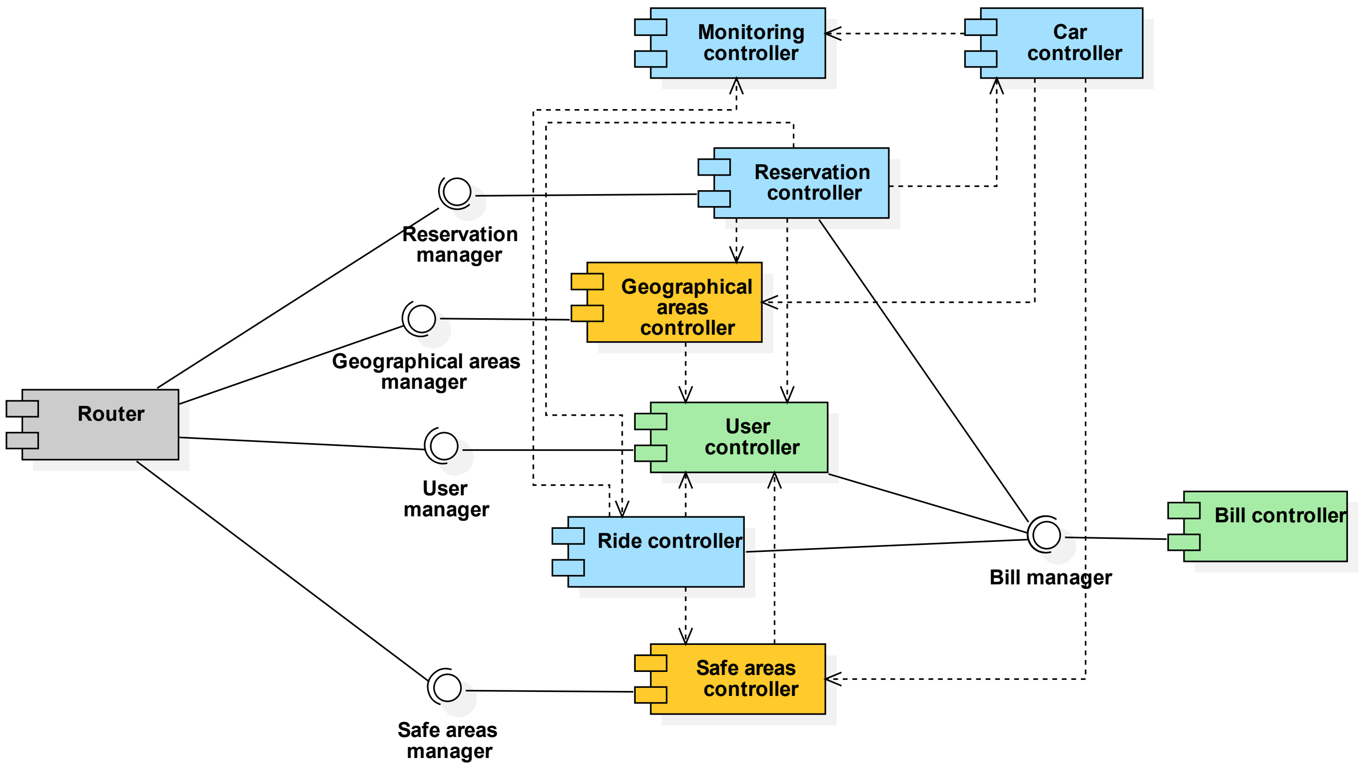
2.4.1.4 Router

The very last component to integrate is the *router*, which works on top of all the other subsystems and so can only be added as the very last component of the *server*. Components from different subsystems have different colors.



2.4.1.5 Fully integrated *server* component

Now we are ready to integrate all subcomponents to form the *server* high-level component (external dependencies are not included).



2.4.2 Components integration

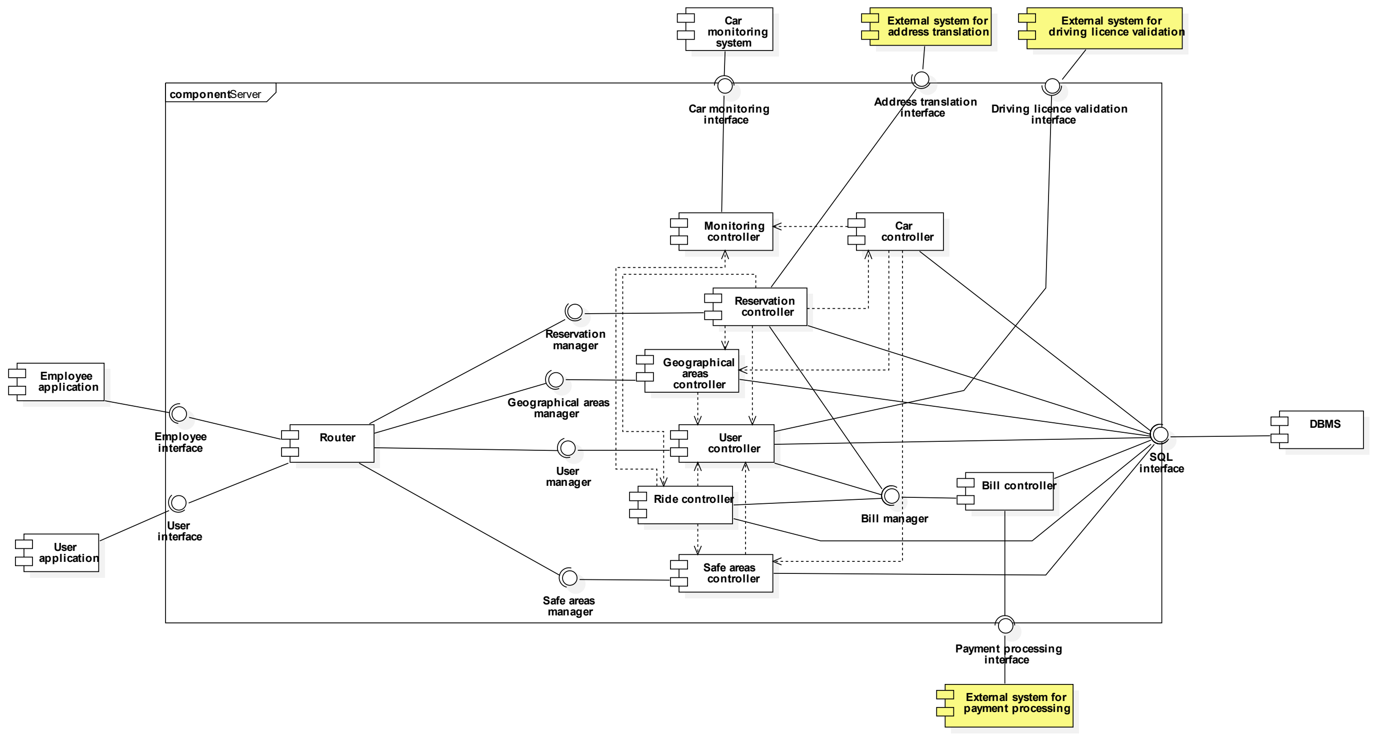
This paragraph provides informations about high-level component integration testing.

Due to the bottom-up approach, the first components that we are going to integrate are *server* and *DBMS*. The interaction between these two components has already been tested during the previous phase, when we verified all the interaction between the *DBMS* and a subcomponent of the *server*

Then we add the *car monitoring system*, whose interaction with the server has already been verified, just like what happened with the *DBMS*.

We can now test in parallel the integration of *user application* with *server* and of *employee application* with *server*, has this two components have no interaction one with the other. This can also save time, as we can carry out the analysis in parallel.

Here we have a picture of the fully integrated system.



Car monitoring system
External system for address translation
External system for driving licence validation
componentServer
Car monitoring interface
Address translation interface
Driving licence validation interface
Monitoring controller
Car controller
Reservation controller
Reservation manager
Geographical areas controller
Geographical areas manager
Employee application
Employee interface
Router
User controller
User manager
SQL interface
DBMS
Ride controller
Bill manager
Bill controller
User application
User interface
Safe areas controller
Safe areas manager
Payment processing interface
External system for payment processing

3 Individual steps and test description

3.1 Integration test case I1

Test case identifier	I1T1
Test items	Bill Controller → DBMS
Test description	This test ensures that a bill record will be created in the database.
Input specification	The amount and the date of a valid bill.
Output specification	The database table must have a bill record.

Test case identifier	I1T2
Test items	Bill Controller → DBMS
Test description	This test ensures that only valid bills will be stored into the database.
Input specification	A non valid bill (past date, negative amount…).
Output specification	It must return an error, no record must be stored in the database.

Test case identifier	I1T3
Test items	Bill Controller → DBMS
Test description	This test ensures that bills can be retrived for a specific user.
Input specification	A user.
Output specification	It must return all the bills for the user and no other bills.

3.2 Integration test case I2

Test case identifier	I2T1
Test items	External payment processing system → Bill Controller, DBMS
Test description	This test ensures that the bill controller communicates with the external payment processing system.
Input specification	Non valid credit card data.
Output specification	An error must be returned.

Test case identifier	I2T2
Test items	External payment processing system → Bill Controller, DBMS
Test description	This test ensures that the bill controller communicates with the external payment processing system.
Input specification	Valid credit card data.
Output specification	The test completes without errors.

Test case identifier	I2T3
Test items	External payment processing system → Bill Controller, DBMS
Test description	This test ensures that the bill controller communicates with the external payment processing system.
Input specification	Valid credit card data, bill.
Output specification	The payment must be processed.
Test dependencies	I2T1

Test case identifier	I2T4
Test items	External payment processing system → Bill Controller, DBMS
Test description	This test ensures that the bill controller communicates with the external payment processing system.
Input specification	Non valid credit card data (CVV non correct, non existing credit card number), valid bill.
Output specification	The payment must be rejected, and an error must be returned.
Test dependencies	I2T1

3.3 Integration test case I3

Test case identifier	I3T1
Test items	User Controller → DBMS, Bill Controller
Test description	This test ensures that a user record is created for a user.
Input specification	A user.
Output specification	The database table must have a user record.

Test case identifier	I3T2
Test items	User Controller → DBMS, Bill Controller
Test description	This test ensures that a user record is not created for a non valid user or for a valid user whose username is already in the database.
Input specification	A non valid user.
Output specification	Must return an error, no modification to the database must be performed.

Test case identifier	I3T3
Test items	User Controller → DBMS, Bill Controller
Test description	This test ensures that a user or an employee can be authenticated.
Input specification	The username and the password for a valid account.
Output specification	It must return a valid authentication token.

Test case identifier	I3T4
Test items	User Controller → DBMS, Bill Controller
Test description	This test ensures that a user or an employee can be authenticated.
Input specification	The username and the password for a non valid account.
Output specification	It must return an error, the error must not contain information regarding which field is not valid.

Test case identifier	I3T5
Test items	User Controller → DBMS, Bill Controller
Test description	This test ensures that a user having a pending bill is banned.
Input specification	A user, a bill for the user which is pending.
Output specification	The user must be banned.

3.4 Integration test case I4

Test case identifier	I4T1
Test items	External driving licence validation service → User Controller, DBMS, Bill Controller
Test description	This test ensures that ensures that this component is able to communicate with the external driving licence validation service.
Input specification	Some user data with valid and driving licence.
Output specification	User is inserted in the database

Test case identifier	I4T2
Test items	External driving licence validation service → User Controller, DBMS, Bill Controller
Test description	This test ensures that ensures that this component is able to communicate with the external driving licence validation service.
Input specification	Some user data with non valid driving licence (non existing driving licence number, outdated driving licence number).
Output specification	No operation is performed on the database

3.5 Integration test case I5

Test case identifier	I5T1
Test items	Geographical areas controller → DBMS
Test description	This test ensures that the merge operation behaves correctly.
Input specification	Two regions.
Output specification	The two region must be merged, the database must reflect this change, hence it must not contain the input region and must contain a new region which is the union of the two.

Test case identifier	I5T2
Test items	Geographical areas controller → DBMS
Test description	This test ensures that the split operation behaves correctly.
Input specification	A region and a path within the region.
Output specification	The region is split using the path and the database must reflect this change, hence it must not contain the input region and must contain two regions whose union corresponds to the input region, and that have as a part of their boundary the given path

Test case identifier	I5T3
Test items	Geographical areas controller → DBMS
Test description	This test ensures that the split operation behaves correctly.
Input specification	A region and a path outside the region.
Output specification	No operation must be performed, an error must be returned.

Test case identifier	I5T4
Test items	Geographical areas controller → DBMS
Test description	This test ensures that the search operation behaves correctly.
Input specification	A position and a radius.
Output specification	It must return all and only the regions whose polygon intersects a circle centered in the given position with the given radius.

3.6 Integration test case I6

Test case identifier	I6T1
Test items	Safe areas controller → DBMS
Test description	This test ensures that the remove operation behaves correctly.
Input specification	A safe area.
Output specification	The safe area must be removed from the database.

Test case identifier	I6T2
Test items	Safe areas controller → DBMS
Test description	This test ensures that the add operation behaves correctly.
Input specification	Data to create a valid safe area.
Output specification	A new safe area must be inserted in the database.

Test case identifier	I6T3
Test items	Safe areas controller → DBMS
Test description	This test ensures that the add operation behaves correctly.
Input specification	Data to create a safe area that overlaps with another, already existing, safe area.
Output specification	No operation must be performed, an error must be returned.

Test case identifier	I6T4
Test items	Safe areas controller → DBMS
Test description	This test ensures that the edit operation behaves correctly.
Input specification	Valid data to update a safe area.
Output specification	The old safe area is updated and the changes are stored in the database.
Test dependencies	I6T2

Test case identifier	I6T5
Test items	Safe areas controller → DBMS
Test description	This test ensures that the edit operation behaves correctly.
Input specification	Non valid data to update a safe area (open boundary, negative number of plugs for recharging stations, …).
Output specification	No operation must be performed, an error must be returned.
Test dependencies	I6T3

Test case identifier	I6T6
Test items	Safe areas controller → DBMS
Test description	This test ensures that the search operation behaves correctly.
Input specification	A position and a radius.
Output specification	It must return all the safe areas whose polygon intersects a circle centered in the given position with the given radius.

3.7 Integration test case I7

Test case identifier	I7T1
Test items	Car monitoring system → Monitoring controller
Test description	This test ensures that the StatusMessage is received.
Input specification	The StatusMessage from the car.
Output specification	The StatusMessage is received and dispatched.

Test case identifier	I7T2
Test items	Car monitoring system → Monitoring controller
Test description	This test ensures that the Unlock message is received by the car.
Input specification	The car to which the message is to be delivered.
Output specification	The car must perform all the action necessary to unlock itself.

Test case identifier	I7T3
Test items	Car monitoring system → Monitoring controller
Test description	This test ensures that the Lock message is received by the car.
Input specification	The car to which the message is to be delivered.
Output specification	The car must perform all the action necessary to lock itself.

3.8 Integration test case I8

Test case identifier	I8T1
Test items	Car controller → Monitoring controller, DBMS, Safe areas controller, Geographical areas controller
Test description	This test ensures that only available cars can be reserved.
Input specification	Some cars to be reserved (this test must be run several times, sometimes use an available car, some other times use a non available one).
Output specification	Only available cars must be reserved.

Test case identifier	I8T2
Test items	Car controller → Monitoring controller, DBMS, Safe areas controller, Geographical areas controller
Test description	This test ensures that low battery cars are put in maintenance.
Input specification	A StatusMessage reporting a low (< 20%) battery level and the car parked somewhere.
Output specification	The car sending the message must be put in maintenence.

Test case identifier	I8T3
Test items	Car controller → Monitoring controller, DBMS, Safe areas controller, Geographical areas controller
Test description	This test ensures that when a maintanance car has a battery level above 20% is put into the available cars.
Input specification	A StatusMessage reporting a battery level above 20%.
Output specification	The car must be put into the available cars.

Test case identifier	I8T4
Test items	Car controller → Monitoring controller, DBMS, Safe areas controller, Geographical areas controller
Test description	This test ensures that only reserved car can be put in use.
Input specification	Some MessageStatus reporting a InUse status and the cars sending them.
Output specification	Only reserved cars must be put in use.

Test case identifier	I8T5
Test items	Car controller → Monitoring controller, DBMS, Safe areas controller, Geographical areas controller
Test description	This test ensures that only in use car can be put in available status.
Input specification	Some MessageStatus reporting a InUse status, with engine turned off and the cars sending them.
Output specification	Only in use cars whose position is inside a safe area must be put in the available status, its geographical area must be updated accordingly.

3.9 Integration test case I9

Test case identifier	I9T1
Test items	Ride Controller → Monitoring controller, DBMS, Safe areas controller, Car controller
Test description	This test ensures that the ride controller terminates the ride after a car is parked in a safe area.
Input specification	A sequence of StatusMessage reporting: • a car in use. • the same car parked in a safe area.
Output specification	A bill must be processed and the ride removed.

Test case identifier	I9T2
Test items	Ride Controller → Monitoring controller, DBMS, Safe areas controller, Car controller
Test description	This test ensures that the ride controller computes a correct price for the ride.
Input specification	Some sequences of StatusMessage reporting a car in use, some StatusMessages that triggers either a single discount or a single raise and the StatusMessage reporting the car parked (if not already present).
Output specification	The bill amount must be compatible with the raise or the discount applied.
Test dependencies	I9T1

Test case identifier	I9T3
Test items	Ride Controller → Monitoring controller, DBMS, Safe areas controller, Car controller
Test description	This test ensures that the ride controller computes a correct price for the ride.
Input specification	A sequence of StatusMessage reporting: • a car in use by a single person. • the same car in use by 3 passengers. • the same car parked in a safe area.
Output specification	The bill amount must not include any discount based on the number of passengers.
Test dependencies	I9T2

Test case identifier	I9T4
Test items	Ride Controller → Monitoring controller, DBMS, Safe areas controller, Car controller
Test description	This test ensures that the ride controller computes a correct price for the ride.
Input specification	A sequence of StatusMessage reporting: • a car in use by 2 passengers. • the same car parked in a safe recharging area. • the same car in recharge.
Output specification	The bill amount must be compatible with multiple discount applied.
Test dependencies	I9T2

3.10 Integration test case I10

Test case identifier	I10T1
Test items	Reservations controller → DBMS, Car controller, Geographical areas controller
Test description	This test ensures that a reservation is created in response to a request.
Input specification	A valid car and a user who wants to serve the car.
Output specification	A new reservation record must be created, the car must be in reserved status.

Test case identifier	I10T2
Test items	Reservations controller → DBMS, Car controller, Geographical areas controller
Test description	This test ensures that a reservation is not made if there is already a reservation in the same geographical area.
Input specification	A car and a user who wants to reserve the car, the user has already a reservation in the same geographical area.
Output specification	An error must be returned, no operation must be performed.

Test case identifier	I10T3
Test items	Reservations controller → DBMS, Car controller, Geographical areas controller
Test description	This test ensures that only available cars can generate a reservation record.
Input specification	A car which is not available, and a user who wants to reserve that car.
Output specification	An error must be returned, no operation must be performed.

Test case identifier	I10T4
Test items	Reservations controller → DBMS, Car controller, Geographical areas controller
Test description	This test ensures that a reservation can trigger an expiration, for the purpose of this test the expiration may be set to a time interval different from the production one (1 hour)
Input specification	A car and a user who wants to reserve the car.
Output specification	After the defined time interval, the reservation must be removed, a bill must be processed for a 1€ fee.
Test dependencies	I10T1

Test case identifier	I10T5
Test items	Reservations controller → DBMS, Car controller, Geographical areas controller
Test description	This test ensures that the controller is able to get all the cars near a position.
Input specification	Some position, radius and the status of the car to search for.
Output specification	It must return all the cars with the given status at most radius meter distant from the given position.

Test case identifier	I10T6
Test items	Car controller → Monitoring controller, DBMS, Safe areas controller, Geographical areas controller
Test description	This test ensures that the geocoding of an address is done correctly.
Input specification	Some location, radius and the status of the car to search for.
Output specification	It must return all the cars with the given status at most radius meter distant from the given location.
Test dependencies	I10T5

3.11 Integration test case I11

Test case identifier	I11T1
Test items	Router → all the components of the server
Test description	This test ensures that requests are routed correctly.
Input specification	Some request for different services.
Output specification	The corresponding controller is actived with the data received in the request.

Test case identifier	I11T2
Test items	Router → all the components of the server
Test description	This test ensures that a request to a non existent controller method fails gracefully.
Input specification	A request for which no controller method is associated.
Output specification	An error must be returned.

Test case identifier	I11T3
Test items	Router → all the components of the server
Test description	This test ensures that requests for employee services are granted only to employees.
Input specification	A request issued by a user for an employee service.
Output specification	An error must be returned.
Test dependencies	I11T1

Test case identifier	I11T4
Test items	Router → all the components of the server
Test description	This test ensures that requests for user services are granted only to users.
Input specification	A request issued by an employee for a user service.
Output specification	An error must be returned.
Test dependencies	I11T1

3.12 Integration test case SI1

Test case identifier	SI1T1
Test items	Router → all the components of the server
Test description	This test ensures that requests can be processed fast enough to meet our performance requirements.
Input specification	Multiple concurrent request.
Output specification	Request must be processed within the time specified in the performance requirements.

3.13 Integration test case SI2

Test case identifier	SI2T1
Test items	Web browser → Web server
Test description	This test ensures that web request are correctly processed.
Input specification	Some HTTPS request issued by the web browser.
Output specification	The correct page is displayed.

Test case identifier	SI2T2
Test items	Web browser → Web server
Test description	This test ensures that web requests can be processed fast enough to meet our performance requirements.
Input specification	Multiple concurrent HTTPS request.
Output specification	Request must be processed within the time specified in the performance requirements.
Test dependencies	SI2T1

4 Tools and Test equipment required

4.1 Tools

One of the main desirabe characteristic of a set of test cases is that it can be run automatically. This is also helpful after system deploying in order to verify that maintanance operations and modifications do not break interaction between components. In order to achieve this goal, two main tools are going to be uses: **JUnit framework** and **Arquillian testing framework**.

- **JUnit**
  Even if JUnit has been historically designed in order to automate Unit Testing, it can come very handy also during integration testing, as it can automate the execution of methods as of the developed drivers (that are to be integrated with JUnit), verify responses against a oracle and check that appropriate exceptions are raised in the expected cases (according to the specification). This tool provides a common base upon which drivers are built, so as to fully automate test lifecycle (except for test definition, which is stated in this document).

- **Arquillian testing framework**
  This framework comes very handy in executing container-level tasks in Java EE. In particular, these kind of test concerns mainly the interaction with the *DBMS* (because many low-level interactions with the DBMS are actually performed by the Java EE containers, and so we need to check they interact properly with business components) and the testing of dependency injection (which is widely used between components that need instances of other components). Moreover, this framework also helps in creating partial builds of the system, which is used in bottom up integration, when different components are included one at a time. This is accomplished by the *micro-deployment* created in a programmable way with Arquillian.

Not all tests can however be automated, for example GUI related components cannot be completely tested in a automatic way. Another tool is then required, and it is **manual testing**. This tool has the great disadvantage of not being automatically reproducible, but it is the only viable tool to test some features of the application. Moreover, the intire activity of designing the test suit and implement the proper drivers and oracles has to be carried out manually.

During the integration of the system it may happen that some interactions are tested and then used during other tests. When this interactions involve external services, they must be tested against the real service in the dedicated tests, but then external services should be mocked in order to avoid to query them uselessly. A tool that grants this possibility is **mock server**.

In order to test *user application* and *employee application* interaction with the server component, the **Chrome developer tools** may come very useful, as they provide functions to inspect both web requests and responses and interal components' state (other web browsers provide similar tools, but this is the most complete and flexible tool, according to developer's experience). Because of the fact that this needs to be a cross-browser application, it should be tested on all the major modern browsers, which include:

- Google Chrome
- Mozilla Firefox
- Safari
- Opera browser
- Micorsoft Edge
- Google Chrome for Android
- Mobile Safari
- Android Browser
- Android WebView
- IE Phone
- Opera Mobile
- Firefox Mobile

Mobile versions of major desktop browsers can also be tested through the emulator provided by the desktop version.

4.2 Test equipment

A complete test equiment is composed of devices on which the different components can be deloyed.

In order to deploy the *server* component, a complete J2EE server infrastructure is needed. A draft for this infrastructure is composed of:

- **GlassFish Java Application server**
- **Java Enterprise Edition** runtime
- **GlassFish Message Broker** (in order to setup the message-driven part of the overall architecture)

The *DBMS*, in order to mimic the real deployment scenario, should be constituted by an instance of **MySQL** deployed on a dedicated machine.

The *monitoring system* is deployed on board of the cars, thus at least one car of the PowerEnJoy company is needed (unless the car manufacturer provides a reliable stub of the real system for testing purposes)

The *user application* and the *employee application* are web application, and so they run in the context of a browser. The only hardware needed to test them is a normal notebook, plus a set of mobile devices or emulators in order to test system agaist mobile browsers. No particular screen resolution is required, and more different screen resolutions should be tested, in order to check the reactivity of the design, in particular with respect to very large or very small devices. Clearly, an instance of each of the browsers listed in the previous paragraph is needed.

4.3 Program stubs and drivers

In order to actually perform the integration testing, and then to run tests, a number of drivers and stubs is generally needed. Drivers and stubs, in order to invoke methods and emulate components, will work in the context of the **JUnit** framework.

4.3.1 Stubs

Due to the *bottom up* integration approach, no stub should be required by the testing activity, as system is built from the most simple components up. By the way, as components involved in client-server interaction are mutually coupled, a number of stubs will also be required in order to emulate the behaviour of a component receiving a response. In particular, only one stub component, with the very simple function of logging the response received from the server should be used until *user application* and *employee application* are fully developed and unit tested.

4.3.2 Drivers

Drivers are a very key point in *bottom up* integration testing, as they responsible of trigger actions (and then interactions) in components, and this actually drives the test.
Here is a list of all the drivers that will be developed in order to carry out the integration testing phase.

- **Bill driver**
  This driver calls the methods exposed by the *bill controller* component in order to test its integration with the *DBMS*

- **User driver**
  This driver calls the methods exposed by the *user controller* in order to test its integration with the *bill controller* and the *DBMS*.

- **Geographical areas driver**
  This driver calls the methods exposed by the *geographical areas controller*, in order to test the integration with *DBMS* and *user controller*. Due to the intrensic complexity of some algorithms used inside the *geographical areas controller*, particular attenction should be put to test that the communication with other modules works correctly.

- **Safe areas driver**
  This driver calls the methods exposed by the *safe areas controller* in rder to test the interaction with the *user controller* and the *DBMS*.

- **Monitoring controller driver**
  This driver calls the methods exposed by the *monitoring controller* in order to test its interaction with the external *car monitoring system* component. This driver is a very key point in integration testing, both because the driven components directly interacts with an external component, and because that external component is supposed to be deployed on cars, and so the interaction over a mobile network must be checked very carefully.

- **Car controller driver**
  This driver calls the methods exposed by the *car controller* component, in order to test the interaction with the *monitoring controller* and the *DBMS*

- **Ride controller driver**
  This driver calls the method exposed by the *ride controller* component, in order to test the integration with *monitoring controller*, *user controller*, *bill controller*, *DBMS* and *safe areas controller*

- **Reservation controller driver**
  This driver calls the method exposed by the *reservation controller* component, in order to test the integration with *ride controller*, *user controller*, *bill controller*, *DBMS* and *geographical areas controller*

- **Router driver**
  This driver calls the methods exposed by the *router* component, in order to test the external public API provided by the system, which is composed of functions exposed by the *user controller*, the *reservation controller*, the *safe areas controller* and the *geographical areas controller*

- **Client application driver**
  This driver emulates the requests of both *user application* and *employee application*. This is possible because both use the same communication technology (HTTPS protocol). In particular, it sends requests to the public server API, and works together with the *client stub* mentioned in the previous paragraph to check correctness of server responses.

5 Appendix

5.1 Effort spent

- Nardo Loris: 8 hours
- Osio Alberto: 11 hours