POLITECNICO DI MILANO

SOFTWARE ENGINEERING 2 PROJECT

A.Y. 2016-17

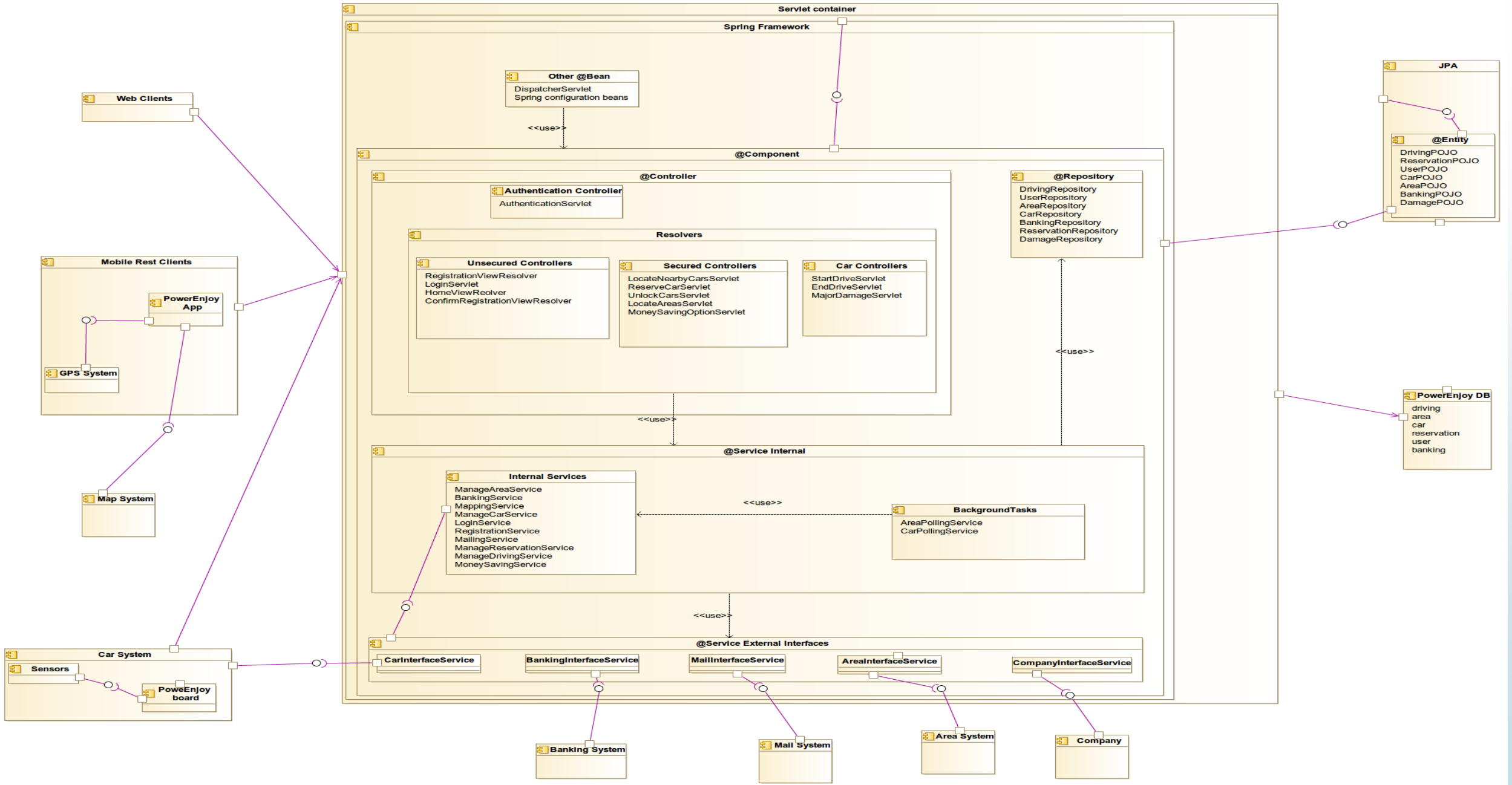
PowerEnJoy

Integration Test Plan Document

Enrico Migliorini, Alessandro Paglialonga, Simone Perriello

17/01/2017

The Component Diagram:



Entry Criteria

- Before this document is written, and for the steps written here to be meaningful, the **Requirements Analysis and Specification Document** and the **Design Document** of the *PowerEnJoy* project must have been fully written.
- The second required step before the integration test is performed is the Unit Testing of each component.
- Unit Testing verifies the correct functioning of each module and its methods according to their specification; it is used to test a specific unit (class) in the application testing all the methods in the class including all exception paths in the methods. Unit Tests only cover the testing of each module in the application, hence we can't test the functional requirements of the application or scenarios.
- The communication testing between **Services** has to be done in the Unit Testing while the testing between **Services** and **External Service Interfaces** is provided in this phase.
- Unit testing of the DBMS, Mapping System, Banking System has already been done by their respective software producers.
- As the exclusive job of the **External Service Interfaces** is to communicate with External Systems and so there's no real computation, we thought it would be more considerable to test the communication between these two in the Unit Testing.

Components to be integrated 1
Assumption

- The Framework Spring used for the implementation of this project, which has already been fully descripted in the DD, provides a set of native methods belonging to the Repositories that may be called and used by the Services. Obviously the use of these methods between Services and Repositories has no need to be tested as the framework natively implements it. As a result, the only Repositories methods which we're going to test in the Integration Testing are the ones implemented exclusively for the *PowerEnJoy* project.

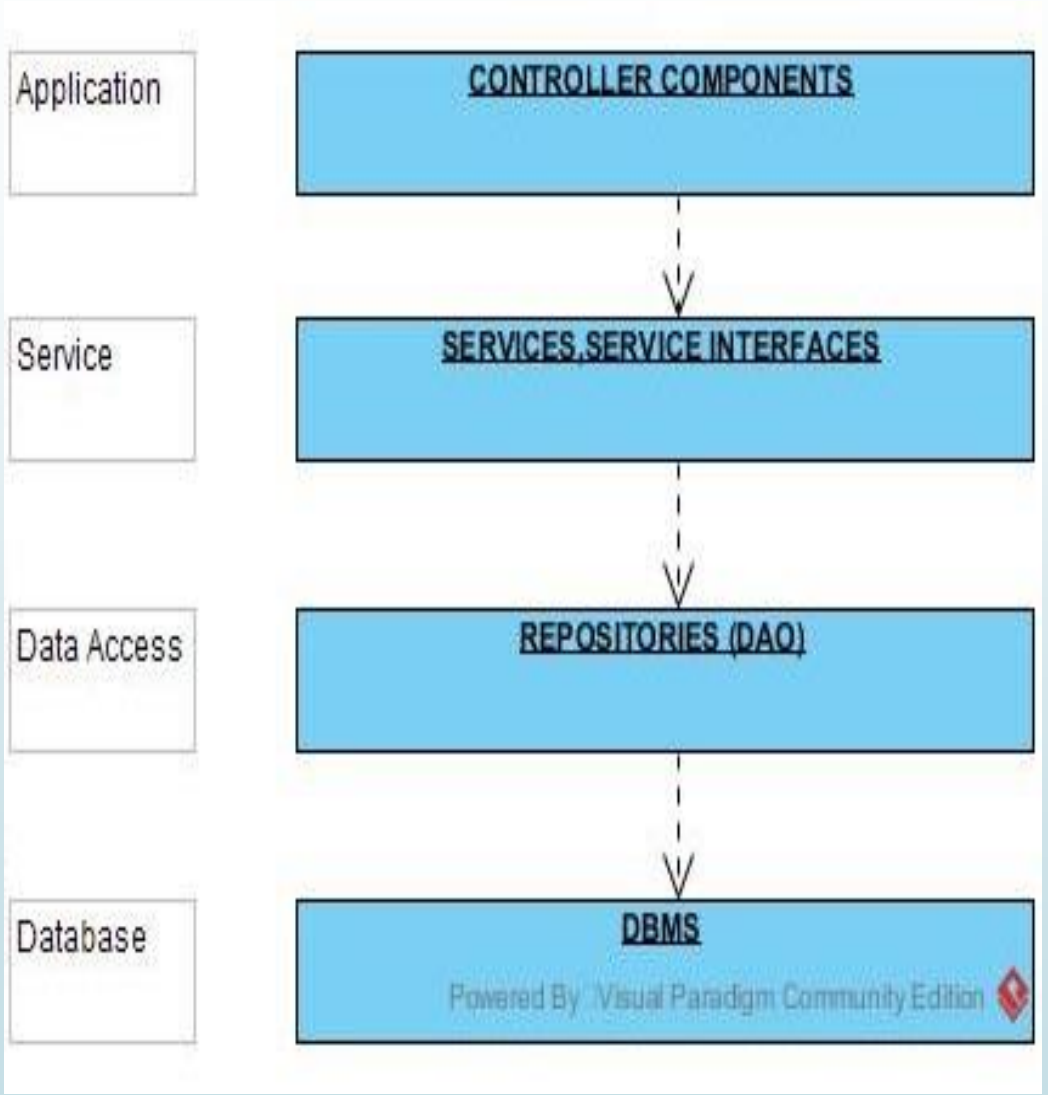
Components to be integrated 2
CRUDRepository Interface

Modifier and Type	Method and Description
long	count() Returns the number of entities available.
void	delete(ID id) Deletes the entity with the given id.
void	delete(Iterable<? extends T> entities) Deletes the given entities.
void	delete(T entity) Deletes a given entity.
void	deleteAll() Deletes all entities managed by the repository.
boolean	exists(ID id) Returns whether an entity with the given id exists.
Iterable<T>	findAll() Returns all instances of the type.
Iterable<T>	findAll(Iterable<ID> ids) Returns all instances of the type with the given IDs.
T	findOne(ID id) Retrieves an entity by its id.
<S extends T> Iterable<S>	save(Iterable<S> entities) Saves all given entities.
<S extends T> S	save(S entity) Saves a given entity.

Image from :
http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html?is-external=true

Components to be integrated 3
PowerEnjoy Server-Side Components

- Starting from the bottom layer of the architecture to go along with our Integration Testing Strategy (see next slides) we'll integrate Server-Side : **DBMS** with the **Repositories**, **Repositories** with **Services**, **Services** with **Service Interfaces**, **Services** with **Controllers**, **Controllers** and the different types of **Clients.**
- We don't need to configure the communication between the **Controllers** and the **Dispatcher** as the Spring framework used for the implementation of the *PowerEnJoy* project, by using annotations on Controllers, provides an automatic mapping between an URL and a specific Controller. In this way the Spring internal Dispatcher invokes for each URL a specific class and so the only check that has to be done is if the correct class is invoked for each URL, which is done in the Unit Testing.
- In the image there is the *PowerEnJoy* layered architecture diagram Server-Side which best shows the granularity of components to test.



Integration Testing Strategy

- During the integration testing strategy selection several popular choices were evaluated by the team. Different strategies exist for the Integration Testing which all have benefits and drawbacks depending on the project to implement, Top Down, Sandwich, Bottom Up, Critical modules first and Big Bang; there's no strategy which is clearly better than the others as each approach is perfectly suited for a specific type of architecture of components and communication between components. Consequently, the team has evaluated the different methodologies looking at the big picture of this project.

Integration Strategy Choice : Bottom Up

The choice which seemed to best fulfill our requirements was the **Bottom Up** approach.

Advantages

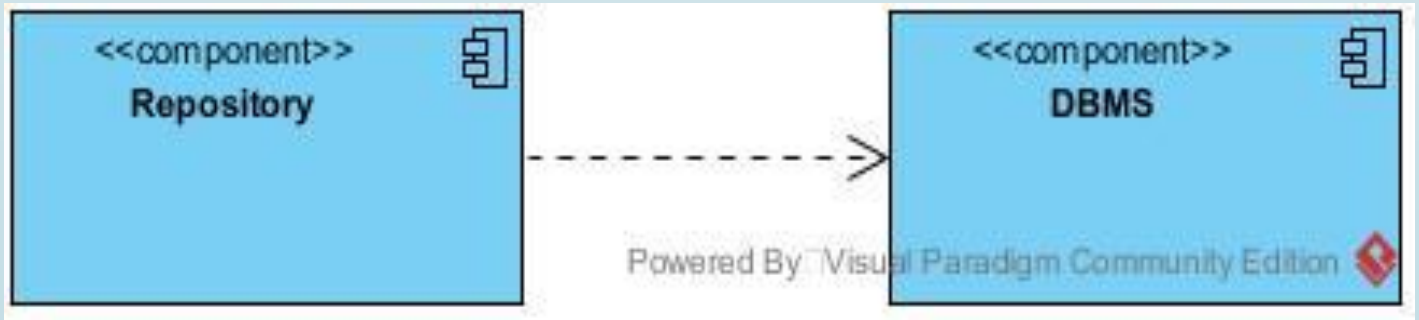
- During the evaluation some of the advantages which led our choice were :
- The absence of urgency for any knd of program stubs, which is needed in top-down approach instead, as we always sitart developing and testing with absolute modules
- Starting from the bottom of the hierarchy means that the critical modules are generally built and tested first and therefore any errors or mistakes in these forms of modules are find out early in the process.
- Fault localization is easier and no time is wasted waiting for all modules to be developed unlike Big-Bang approach.

Drawbacks

- During the evaluation some of the drawbacks which we accepted existing for our project were :
- We always have to form the test drivers which are the simulated environment in which the component being tested is to integrated. In addition, drivers have to be tested themselves and so more time and efforts are needed to accomplish this complicated task.
- The application as an entity does not exist until the last module is added.

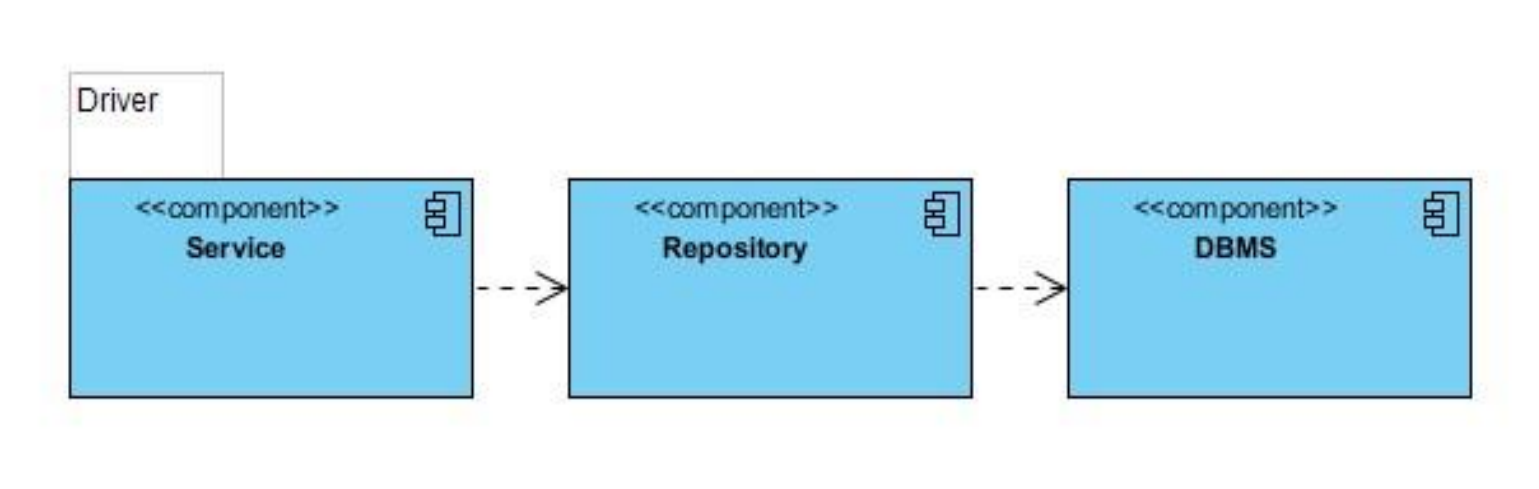
Software Integration Sequence 1
Repositories-DBMS

- We can now describe the order in which the components are integrated and tested. The arrow arrives in the component which needs to be implemented first.
- The first components to be integrated and tested are the DB and the Repositories. There's no need of drivers as the repositories are the first components to be implemented in this project and since the DB has already been implemented by its producer which is external to the team developing the *PowerEnJoy* project.
- There's no actual real communication to test between the DBMS and the Repositories as the only methods called and used by the repositories are the queries made available by the DBMS according to its implementation.
- For the sake of completion and to show every module from the very bottom of the project we're still putting this "fake integration" in this part. We assume that the configuration file of the communication between the DB and the Application server has already been verified in the Unit Testing.



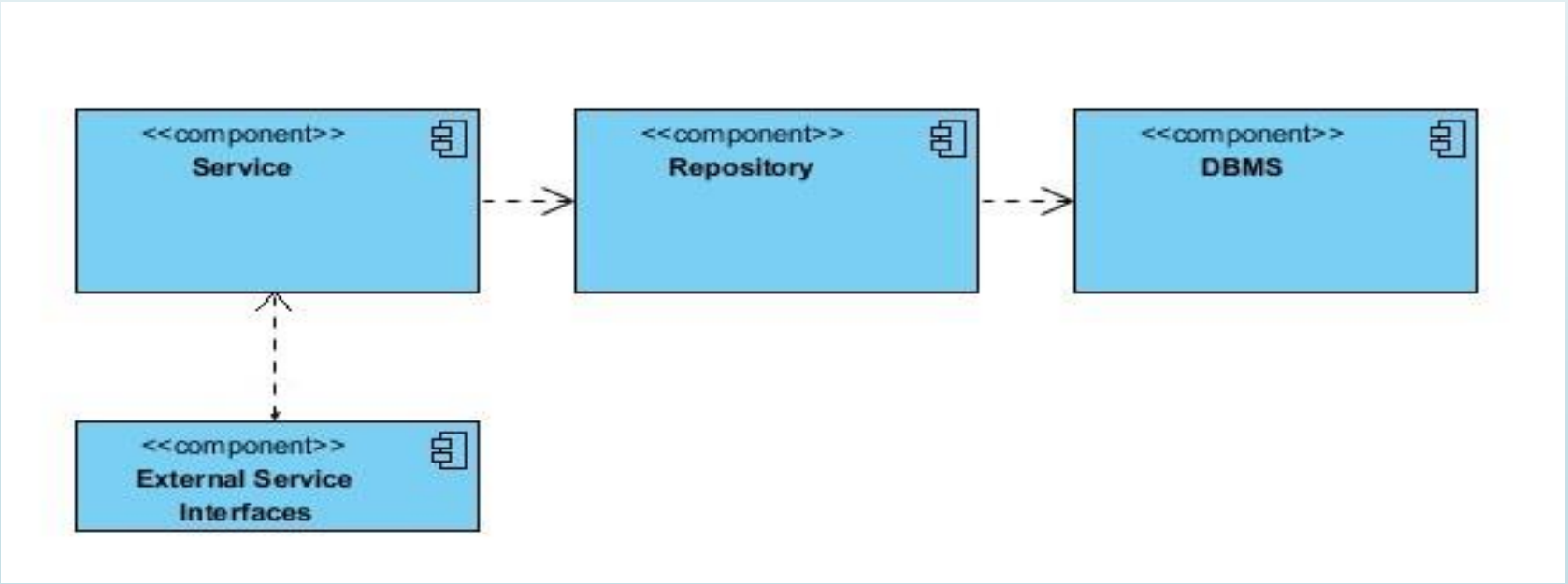
Software Integration Sequence 2

- We can now test the communication and integration between Repositories and Services which have access to them. Repositories are the first components to be implemented while the Services are not implemented yet, this is why we use drivers for Services.



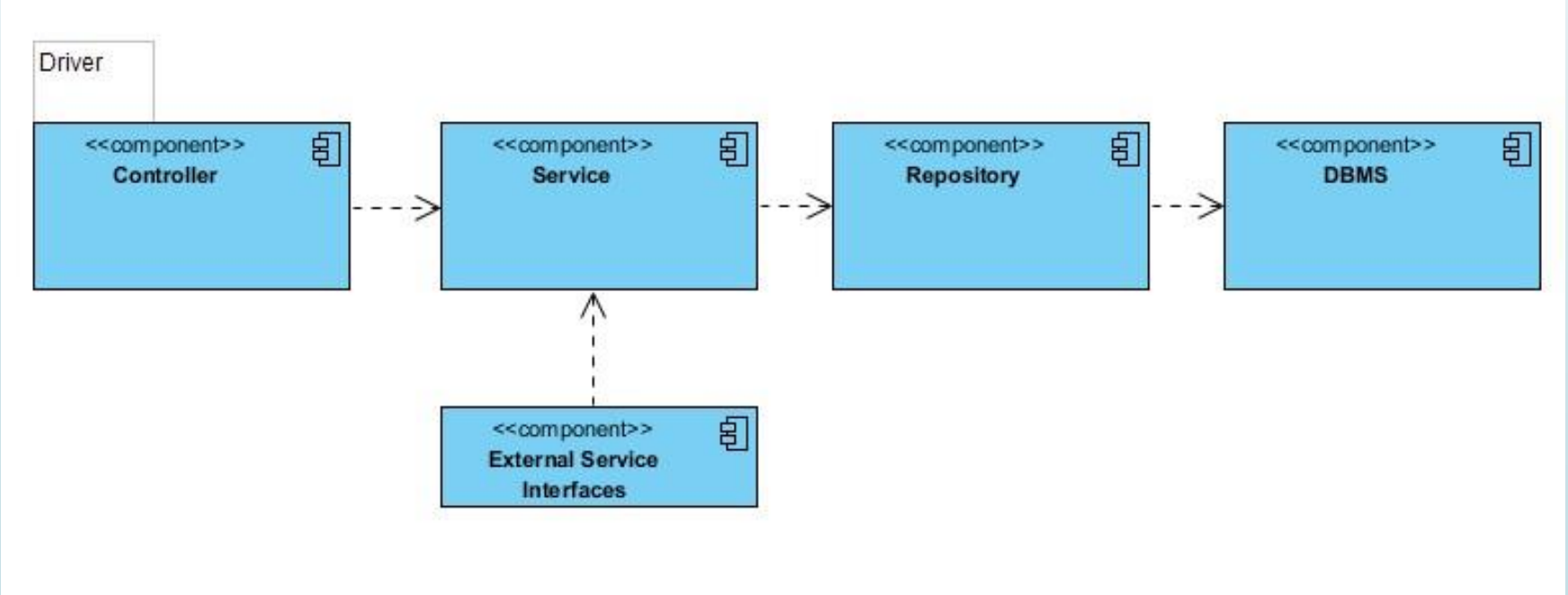
Software Integration Sequence 3

- Now we'll be able to test the integration between Services and External Service Interfaces. The latter are the components that allow the System to connect to External Systems such as the Banking and the Mailing ones. The implementation of External Service Interfaces can be done in parallel with the implementation of the Services, hence we can test the communication between the very components without using any driver.



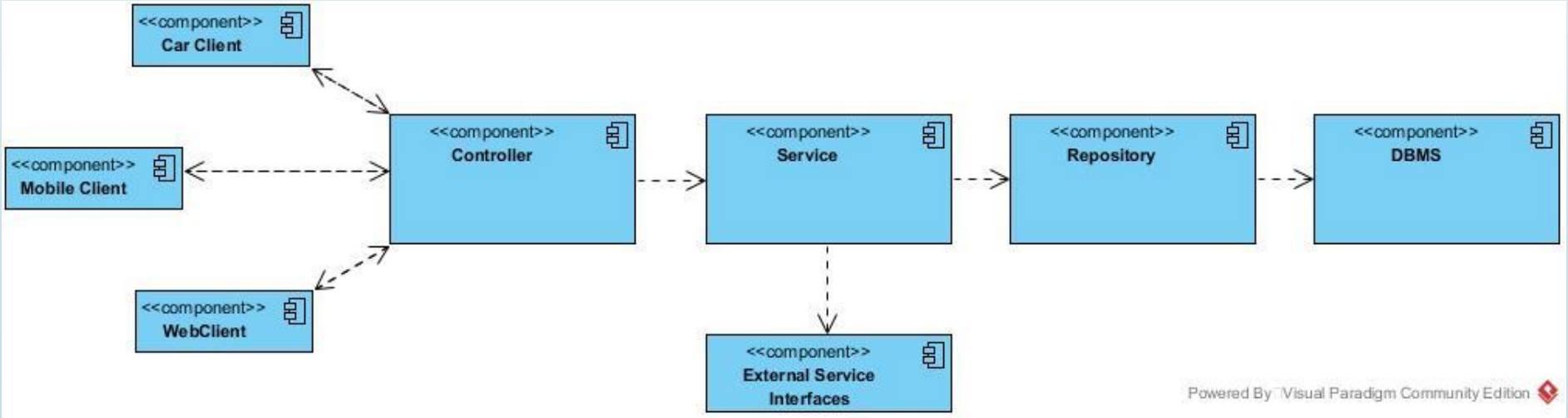
Software Integration Sequence 4

- Now including the Controllers we can test the all Business Logic. Controllers will be substituted by their drivers.



Software Integration Sequence 5

- Finally we integrate the System Server-Side with its Client-Side. So we integrate the Controllers with the different Clients: Mobile Client, WebClient and CarClient. We assume their development is already done and hence we don't need any driver. Web Clients only have access to Register and Confirm Registration methods via HTPP requests and their task is to simply receive a jsp/html page as response from Controllers. Hence, as there's no logic client-side we can think of including the testing of WebClients with Controllers in the Unit Testing.



Test Case Specification

- Here we're going to provide a detailed description of the tests to be performed on each pair of components that have to be integrated. Each subsection contains the name of the involved components, the first one is the caller component and the second one is the called component. For each method will be provided a different type of input and the corresponding expected effects on the system.
- Remember that native Repositories Spring methods have no need to be tested and consequently aren't listed in the below test cases.

"Caller Component", "Called Component"

Mapping Service, Area Repository

- Here's one of the methods of the Area Repository called by the Mapping Service. We presented this to explain that the "strange choice" of this very long method name is due to the fact that the Spring Framework automatically translates this method in a Database Query, so this is why we inserted each condition of the query in the method name.

FindByGpsLatitudeGreaterThanAndGpsLatitudeLessThanAndGpsLongitudeGreaterThanAndGpsLongitudeLessThanAndChargingSlotsGreaterThan(inout minLatitude: float, inout maxLatitude: float, inout minLongitude: float,inout maxLongitude: float, inout minSlots: integer)	
Input	Effect
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the minimum and maximum GPS coordinates which represent the vertices of a square and will perform a query to the DB to retrieve all the Areas whose GPS coordinates are inside the given built square. The Repository will then forward the list of the resulting Area POJO to the Service if there exists one.

Registration Service, User Repository

- There are no non-native Spring methods of the User Repository called by the Registration Service because, as already described in the DD, the checks on the unicity of SSN, Driving License Number and Email values for the insertion of an account is performed at Database level and so done by the DBMS. Hence the Service will just call the native Spring method “Save” to save the entity in the Database.

Manage Driving Service, Company Interface Service

sendLowBatteryNotification(inout internalID: CarPOJO)	
Input	Effect
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Company Interface Service receives the data about the Car with low battery. It will then communicate with the Company System the data and information about the Car and will receive by the Company System an acknowledgement. The company will then dispatch an employee which will perform the actions already described in the DD.

ReserveCars Controller, ManageReservation Service

reserveCar(inout user: UserPOJO, inout car: CarPOJO)	
Input	Effect
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments but the User has already another active Reservation or is actually driving another reserved Car	An InvalidOperationException is raised.
Valid arguments but the Car is already reserved or in use by another User	An invalidOperationException is raised.
Valid arguments, no active reservation or active drivings of the User and the Car.	The ReserveCars Controller sends the UserPOJO and the CarPOJO to the Service which will create the ReservationPOJO and forward it to the Controller. The Service calls its method "StartReservationTimerInBackground".

Mobile Client, Authentication Controller

- This method is actually performed every time the Mobile Client sends a request with a method reserved for logged Users. The dispatcher automatically dispatches the request to the Authentication Controller to check if the User is logged and then dispatches the request to the actual Controller needed to fulfill the original request.

"Send Token" : (*/secured/*,body(token=...))	
Input	Effect
Invalid arguments	A 401 Unauthorized HTTP Error Code is returned by the Controller.
Valid arguments	The Client is sending a correct login token. A 200 OK HTTP Code is returned by the controller. The request is forwarded to the Controller belonging to the original request.

Mobile Client, ReserveCar Controller

- The Mobile Client and Car Client (as well as WebClient) always communicate with Dispatcher first, but as we already described in paragraph 2.2 the Spring Internal Dispatcher just has the role to dispatch the Client request to the Controllers according to the bindings, so it's meaningful to directly test the communication between Clients and Controllers.

"Reserve Car" : PUT("/secured/car/reserve/{carid}",body("reservation=…,token…))	
Input	Effect
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.
Valid arguments	The Controller sends an answer with a 200 OK HTTP code. In the Body of the HTTP answer there's JSON data containing the ReservationPOJO.
Valid arguments but User has an another active reservation or is driving another reserved Car.	An answer with the 403 Forbidden HTTP error code is forwarded from the Controller to the client.
Valid arguments but the Car is reserved or in use by another User.	An answer with the 403 Forbidden HTTP error code is forwarded from the Controller to the client.

Car Client, Major Damage controller

"Report Damage" : POST("/unsecured-cars/{carId}/majordamage",body(timestamp=…,description=…")	
Input	Effect
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.
Valid arguments	The Controller sends an answer with a 200 OK HTTP code. No further data is attached to the answer.

Tools Used for Testing

- For what concerns the Unit Testing part the tools used are the **JUnit framework** and to adopt the testing technique called "Mock Objects" **Mockito** is the other tool.
- For the Integration Testing instead, we'll use the **Arquillian integration testing framework.** This tool runs tests against a Java container to check the correct behavior between a component and its surrounding execution environment. More specifically Arquillian will help checking if the right component is injected in a specified dependency injection.