

Institut für Rechnergestützte Ingenieursysteme  
Fakultät Informatik, Elektrotechnik und Informationstechnik

Universität Stuttgart  
Universitätsstraße 38  
D - 70569 Stuttgart

Master Thesis Nr. 3507

## **Conceptualization and implementation of a prototype for realistic simulation of vehicles**

John Alexander Velandia Vega

Studiengang:	INFOTECH
Prüfer:	Univ-Prof. Hon-Prof. Dr. Dieter Roller
Betreuer:	M.Sc. Leila Zehtaban
begonnen am:	24.04.2013
beendet am:	24.10.2013
CR-Klassifikation:	C.4, D.2.4, D.2.5, D.2.8, D.2.9, D.4.8, H.2.8, J6, K.6.3



## Abstract

Daimler FleetBoard offers telematic services by means of a special hardware installed in customers' vehicles to collect and send data to the FleetBoard's Service Centre (FBSC) platform. FBSC is in charge of receiving, processing and storing data generated by vehicles. The quality assurance and testing department guarantees that the telematic services meet their purpose, and no failures exist in the system. In that way, software to simulate vehicles' behaviour is required to test the functionalities of FBSC. However, the problem rises since this software uses simulated data instead of real data. In addition, the process of creating routes for simulations is manual. Based on the mentioned problems, the objective of this thesis is to design, implement and evaluate a prototype as mechanism of importing routes generated by real vehicles to the simulator's database, to emphasise on using real data for simulations. Additionally, the process of creating routes is optimized using Web Map Services to automate this process. Consequently, an evaluation of the prototypical implementation is considered to guarantee the proper operation of the prototype's layers: *WEB GUI* (supported by Java Server Faces), *business logic* and the *persistence* layer (fostered by Java Persistence API).



---

# Table of contents

---

<b>TABLE OF CONTENTS</b> .....	<b>V</b>
<b>LIST OF FIGURES</b> .....	<b>IX</b>
<b>LIST OF TABLES</b> .....	<b>XI</b>
<b>LIST OF LISTINGS</b> .....	<b>XIII</b>
<b>ABBREVIATIONS</b> .....	<b>XV</b>
<b>1 INTRODUCTION</b> .....	<b>1</b>
1.1 DAIMLER FLEETBOARD .....	1
1.1.1 Profile.....	1
1.1.2 Organisational structure .....	1
1.1.3 The FleetBoard's system architecture .....	1
1.1.4 Services.....	2
1.2 PROBLEM STATEMENT .....	3
1.3 GOALS OF THE MASTER THESIS .....	3
1.4 RESTRICTIONS .....	4
1.4.1 Financial aspect .....	4
1.5 THE THESIS STRUCTURE.....	4
<b>2 BACKGROUND</b> .....	<b>6</b>
2.1 TERMINOLOGIES .....	6
2.2 MAPPING .....	8
2.2.1 Geographical coordinates .....	8
2.2.2 Waypoints .....	8
2.2.3 Global Positioning System (GPS).....	8
2.2.4 GPS Exchange Format (GPX).....	9
2.2.5 Web Map Service (WMS).....	9
2.2.6 Geographic Information Systems databases (GIS).....	9
2.2.7 Tiles .....	9
2.2.8 Map Application Programming Interface (API).....	10
2.2.9 Open Geospatial Consortium (OGC) Architecture .....	10
2.2.10 GPX Visualizer .....	11
2.3 QUALITY MODEL ISO 25000 AND 9126 .....	11
<b>3 SYSTEMS AND COMMUNICATION ARCHITECTURE</b> .....	<b>13</b>
3.1 ARCHITECTURAL OVERVIEW.....	13
3.2 FBSC AND COMMUNICATION PLATFORM.....	14
3.3 INTEGRATION TEST SYSTEM (ITS).....	16
3.4 LIVEVEHICLESIM (LVS) .....	16
3.4.1 Features.....	16
3.4.2 Dataflow .....	17
3.4.3 The Model View and Controller pattern (MVC).....	19
<b>4 CONCEPT AND DESIGN</b> .....	<b>21</b>
4.1 CONCEPT TO INCORPORATE REAL ROUTES IN TOUR SIMULATIONS .....	21
4.1.1 Analysis .....	21
4.1.2 Approaches.....	21
4.1.3 Specification of the selected approach .....	23

4.2	CONCEPT TO INTEGRATE A WMS PROVIDER INTO LVS .....	28
4.2.1	<i>Analysis</i> .....	28
4.2.2	<i>Approaches</i> .....	30
4.2.3	<i>The selected approach</i> .....	33
4.2.4	<i>Specification of the selected approach</i> .....	33
4.3	CONCLUSION .....	34
<b>5</b>	<b>IMPLEMENTATION .....</b>	<b>35</b>
5.1	IMPLEMENTATION OVERVIEW .....	35
5.2	WEB GRAPHICAL USER INTERFACE (GUI).....	35
5.2.1	<i>Description of the current Web GUI</i> .....	35
5.2.2	<i>New CRUD operations using real routes</i> .....	36
5.2.3	<i>Integration of Nokia Maps into LVS</i> .....	37
5.3	THE BUSINESS LOGIC .....	40
5.3.1	<i>Creation of routes</i> .....	41
5.3.2	<i>Retrieving routes</i> .....	47
5.3.3	<i>Updating routes</i> .....	47
5.3.4	<i>Delete routes</i> .....	47
5.4	THE DATA MODEL .....	48
5.5	CLASS DIAGRAM .....	50
<b>6</b>	<b>TEST AND VALIDATION .....</b>	<b>51</b>
6.1	CONFIGURATION OF TESTS USING JMETER .....	51
6.2	TEST AND VALIDATION OF FUNCTIONAL REQUIREMENTS .....	51
6.2.1	<i>Test scripts</i> .....	51
6.2.2	<i>Tests results</i> .....	52
6.3	TEST AND VALIDATION OF NON-FUNCTIONAL TESTS .....	56
6.3.1	<i>Methodology</i> .....	56
6.3.2	<i>Response time analysis</i> .....	56
6.3.3	<i>Throughput analysis</i> .....	59
6.4	DATA QUALITY ANALYSIS AND EVALUATION .....	59
<b>7</b>	<b>ASSESSMENT .....</b>	<b>62</b>
7.1	EVALUATION OF REQUIREMENTS .....	62
7.1.1	<i>Functional and non-functional requirements</i> .....	62
7.1.2	<i>Evaluation of data quality</i> .....	64
7.2	CONCLUSION .....	65
<b>8</b>	<b>SUMMARY AND FUTURE WORK.....</b>	<b>66</b>
	<b>APPENDIX A .....</b>	<b>68</b>
	DATAFLOW DIAGRAM (DFD) .....	68
	<b>APPENDIX B.....</b>	<b>69</b>
	INDIVIDUAL EVALUATION OF WMS PROVIDERS .....	69
	<i>Google</i> .....	69
	<i>Microsoft</i> .....	70
	<i>OpenStreetMap</i> .....	70
	<i>Nokia</i> .....	71
	<b>APPENDIX C .....</b>	<b>72</b>
	DESCRIPTION OF TOOLS USED IN THIS THESIS .....	72
	<b>APPENDIX D .....</b>	<b>73</b>
	DESCRIPTION OF THE GRAPHICAL CONFIGURATION OF JMETER .....	73
	<b>APPENDIX E.....</b>	<b>77</b>

CRUD OPERATIONS RESULTS AND DESCRIPTIONS .....	77
<b>BIBLIOGRAPHY .....</b>	<b>81</b>





---

## List of figures

---

Figure 1.1 Overview of the communication platform, based on [DAI13].....	2
Figure 2.1: Company’s perspective: tour definition .....	7
Figure 2.2 Elements of geographical coordinates .....	8
Figure 2.3 Global schema of the GPX elements .....	9
Figure 2.4 Maps: system architecture .....	10
Figure 3.1 General system architecture overview .....	14
Figure 3.2 Communication and systems architecture .....	14
Figure 3.3 Systems and communication architecture for testing .....	16
Figure 3.4: Site map .....	17
Figure 3.5: LVS Dataflow Diagram.....	18
Figure 3.6 LVS and the MVC design pattern .....	20
Figure 4.1. The new model for importing routes .....	23
Figure 4.2 Use cases considered in this thesis .....	24
Figure 4.3: Current process to import routes .....	29
Figure 4.4 Approach 1: Import routes process removing <i>sub-process 1</i> . .....	30
Figure 4.5 Approach2: Import routes process, removing <i>sub-process 1 &amp; 2</i> .....	31
Figure 5.1 Current Web GUI .....	36
Figure 5.2 The implementation for importing routes from FBSC database .....	37
Figure 5.3 Interaction of Nokia WMS and the routes administration Web GUI.....	38
Figure 5.4 Example of the splitting operation.....	44
Figure 5.5 Class diagram.....	50
Figure 6.1 HTTP responses after running a pre-configured test case .....	52



---

## List of tables

---

Table 2.1 Criteria according to the Quality model ISO 9126 .....	12
Table 4.1 Description of use case <i>Create route</i> .....	25
Table 4.2 Description of use case <i>Update route</i> .....	25
Table 4.3 Description of use case <i>Delete routes</i> .....	26
Table 4.4 Description of use case <i>Display routes</i> .....	26
Table 4.5 Description of use case <i>Import routes</i> .....	27
Table 4.6 Description of use case <i>Store routes transaction</i> .....	27
Table 4.7 Description of use case <i>Update routes transaction</i> .....	28
Table 4.8 Description of use case <i>Retrieve routes</i> .....	28
Table 4.9 Evaluation and selection of the WMS Provider .....	32
Table 4.10 Use case to <i>configure Nokia API</i> .....	33
Table 4.11 Use case to create routes using Nokia API .....	33
Table 6.1 Results of the CRUD operations, using HTTP request.....	53
Table 6.2 Template to evaluate behaviour of the Web application using decision tables technique .....	54
Table 6.3 Description of test cases using decision table technique .....	55
Table 6.4 Real and misleading average response times .....	57
Table 6.5 Z confidence level intervals .....	58
Table 6.6 The well-known thresholds for response times using Web applications .....	58
Table 6.7 JMeter Summary report after running tests for CRUD operations over routes .....	58
Table 6.8 Data quality evaluation between FBSC and the new prototype.....	59
Table 7.1 Data quality evaluation between FBSC and the new prototype.....	65



---

## List of listings

---

Listing 5.1 Main part of the algorithm to retrieve waypoints from Nokia's servers.....	40
Listing 5.2 Save FBSC routes algorithm.....	42
Listing 5.3 Algorithm to import FBSC routes .....	43
Listing 5.4 Algorithm to obtain the right waypoints .....	45
Listing 5.5 Interface to access FBSC gpstracedata table. ....	46
Listing 5.6 Interface to access FBSC gpsdata table .....	47
Listing 5.7 Algorithm to delete routes .....	48
Listing 5.8 Interface to access data that belong to routes. ....	48
Listing 5.9 Implementation of methods to access data that belong to routes .....	49



---

## Abbreviations

---

API	Application Programming Interface
DAO	Data Access Object
DP	Datapackets
DFD	Data Flow Diagram
FBSC	FleetBoard Service Centre
GPS	Global Positioning System
GSM	Global System for Mobile
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ISO	International Organization for Standardization
ITS	Integration Test System
JMS	Java Message Service
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSF	Java Server Faces
LVS	Live Vehicle Simulator
MVC	Model View and Controller
OGC	Open Geospatial Consortium
ORM	Object Relation Mapping
OSM	OpenStreetMap
RDBMS	Relational Data Base Management System
SOAP	Simple Object Access Protocol
SQuaRE	Software product Quality Requirements and Evaluation
WAS	Web Application Server
WMS	Web Map Service





---

# 1 Introduction

---

The objective of this chapter is to introduce sufficient information to obtain a general idea about the topic of this thesis, in this regard: the company's profile, products, services and the telematic architecture are described. This chapter also considers the definition of the problem and the goals of this thesis.

## 1.1 Daimler FleetBoard

### 1.1.1 Profile

Daimler FleetBoard GmbH was established in 2003 and is a 100% subsidiary of Daimler AG. The company operates globally, and its headquarter is located in Stuttgart (Germany). Currently more than 140 employees are working for the company (as of December 2010) to develop sustainable solutions for more than 2,000 customers.

FleetBoard is offering telematics expertise, advisory services and support for the everyday business activities of transport companies and logistics firms. It is a DEKRA (German Vehicle Inspection Agency) certified company and has equipped in excess of 85,000 vehicles with the FleetBoard hardware solution.

The vehicles operate in Germany, UK, Austria, Switzerland, Belgium, Luxemburg, the Netherlands, Italy, Spain, France, Poland, the Czech Republic, Romania, the Middle East, South Africa and Brazil. FleetBoard has its own sales and distribution company in the UK - Daimler FleetBoard UK Ltd. In addition to its own dedicated sales representatives in the other countries [DAI13].

### 1.1.2 Organisational structure

The company uses a common vertical structure which consists of a *Chief Executive Officer* (CEO), who is the head of the organisation, *vice-presidencies* that develop the strategic plan of the company, and *departments* that perform specific processes to reach the FleetBoard's objectives. Additionally, SCRUM methodology is used to develop or enhance features in the FleetBoard's products periodically.

The quality assurance department, also part of the black team in SCRUM, has the tasks of testing and monitoring current and new features of products by means of tools to guarantee a high quality of products, other SCRUM teams are responsible for other issues.

### 1.1.3 The FleetBoard's system architecture

FleetBoard's products and services are offered to customers through a communication platform supported by telematic hardware installed in vehicles. This platform is composed of three fundamental parts for collecting, processing and sending data, where the first part constitutes all vehicles from customers, the second one is the FleetBoard Service Centre (FBSC) and the last

one is the customer software interface. Additionally, these three points contain a well-defined workflow that allows them to interchange data one to another. Figure 1.1 illustrates the communication platform.

Every vehicle has installed the TiiRec which is a hardware configured in the vehicle to establish the communication and send data to FleetBoard Service Centre. The TiiRec contains a GSM/GPRS modem for communication over mobile networks, and the Global Positioning System (GPS) receiver for the vehicle position tracking.

The FleetBoard Service Centre is responsible for receiving data from vehicles and providing data to customers. Communication is performed using a private protocol (see section 2.1). The data are stored in a distributed database which increases its availability and enhances the performance of responses to customers.

Customers access their services (see section 1.1.4) throughout internet using a Web GUI or Web services interfaces, e.g., customers see in real time where their vehicles are.

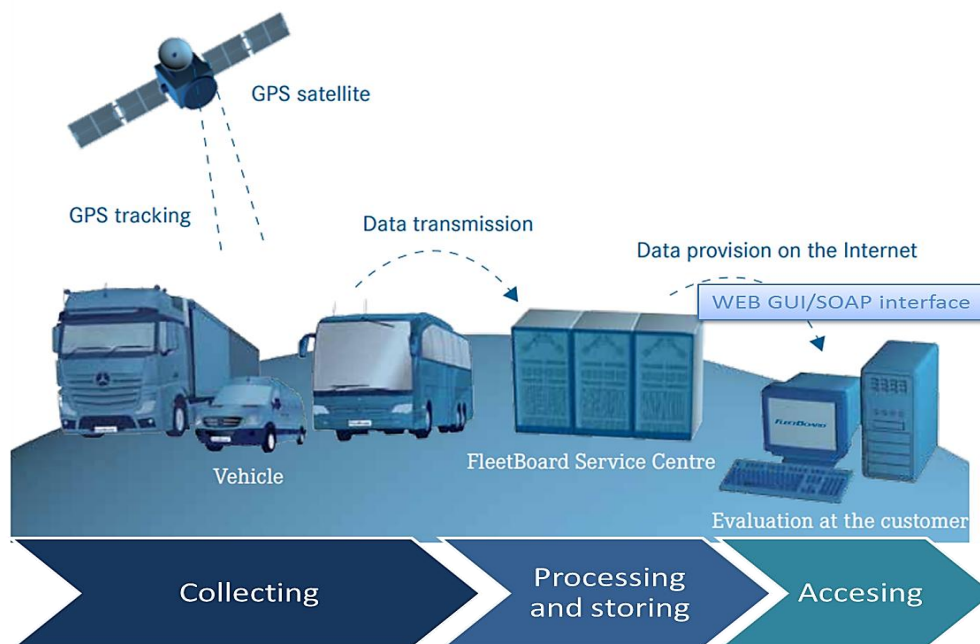


Figure 1.1 Overview of the communication platform, based on [DAI13].

### 1.1.4 Services

Daily messages that are sent from vehicles to FBSC are stored for further analysis and shown to customers throughout Daimler FleetBoard's services. The three main services offered are classified in Vehicle Management to obtain information regarding vehicles' behaviour, Logistics management to gather information related to goods (place of delivery, destination and some other data) and Time Management to acquire working hours, rest time and some additional

information that drivers could generate with their identification card. The communication platform is the means of giving these services (see Figure 1.1).

## **1.2 Problem statement**

The department for quality assurance needs to test the functionality of the entire platform based on software using existing or generated data. In the past, data for testing were generated or real customers data were copied from the production before the tests started. One problem with the existing approach is that the generated data are not close to customer behaviour and the second problem is that the real customers data are static, thus tests do not represent a running system with up to date data coming in continuously.

The first approach to solve these problems is the LiveVehicleSim (LVS). This is a software simulator that generates predefined tours for vehicles and so provides the live data needed for more realistic tests. The actual process of simulating a journey, also known as tour simulation, comprises a vehicle with its driver, thus, a virtual vehicle drives from one place to another, following a route and sending messages to central servers. In this way, data are processed and stored as it came from real vehicles, e.g., data such as speed of vehicle, fuel consumption, etc.

A truck driving all time just for testing purposes is not beneficial for the company; this represents additional costs and time. Thus, tours simulations are performed daily to test functionalities of the FleetBoard central servers. These simulations also help to minimize the impact of unexpected errors and ensure the correct running of the new and old functionalities offered to customers.

However, current simulations use routes that are not close to customer behaviour, because they are created using Google Maps. Furthermore, the creation of routes is a manual process. In detail, creation of routes comprises the setting of a route on Google Maps. The result from Google is transformed into GPS Exchange Format (GPX) to acquire a representable route with coordinates in between. Finally, coordinates are copied in the Web Graphic User Interface (GUI) and stored in the database of LVS.

As second approach, this thesis should provide new functionalities over LVS that incorporate real routes in tour simulations, and an automatic process to create routes using Web Maps, e.g., Nokia Maps, Google Maps, etc. In this way, simulations will be close to reality, and manual processes are eliminated.

## **1.3 Goals of the master thesis**

Considering the problem statement, section 1.2, the main purpose of this thesis is to incorporate a new process of importing and creating routes into LVS to obtain a simulation with more realism and mass data generation. In addition, the integration of Web Map Services (WMS) to make automatic the current process. To achieve the mentioned purpose the following objectives are defined:

- 
- ✓ Develop a refined concept and design based on the analysis of the data source for the simulation and the simulation itself in detail and suggest possible ways of implementation.
  - ✓ Implement a prototype for the concept. The prototype has to enable the simulation of all fleets at the same time using the previously developed concepts and algorithms.
  - ✓ Evaluate the quality of the data generated by the prototype between the actual customer data and previously generated data.

## 1.4 Restrictions

This section includes the conditions and restrictions given by Daimler FleetBoard to consider during the development of this thesis.

### 1.4.1 Financial aspect

The actual process of importing routes lets simulate vehicles driving where current customers have not been yet, which represents an advantage, because the behaviour of FBSC is tested with possible future customer data. Therefore, the concept should contemplate the alternative of optimizing the current process using WMS with minimal cost of money as well as low investment in maintenance, licensing and any other aspect that could cause additional costs

## 1.5 The thesis structure

The following paragraphs show a general overview of the chapters included in this thesis, including a brief explanation of them.

**Chapter 1 Introduction:** This chapter introduces the company with its services. Also, the main purpose and objectives of this thesis are defined based on the presentation of the problem. Finally, financial restrictions are stated according to the company's requirement.

**Chapter 2 Background:** This chapter contains all the basics regarding the telematic business, and also the technical fundamentals. Additionally, this comprises the technical architecture of LVS and the telematic architecture with its components.

**Chapter 3 Systems and communication architecture:** During this chapter all systems that are involved in this thesis are explained in details, including their interrelationships, technologies and patterns behind them.

**Chapter 4 Concept and design:** The development of the concept is contained in this chapter. The first part comes up with a solution and its description to import real routes. The second part is composed of a solution to create routes using WMS. Also, the specification to develop the prototype is defined.

**Chapter 5 Implementation:** The implemented prototype is detailed in this chapter. The tools used for building the prototype are described. The new features included in the prototype are described and the algorithms are explained.

**Chapter 6 Test and validation:** The functional requirements defined in *Chapter 4* are evaluated using testing tools, and the performance of the prototype is assessed using quality of service metrics. Additionally the quality of data is analysed.

**Chapter 7 Prototype assessment:** This chapter evaluates the functional and non-functional requirements analysed and tested in Chapter 6, considering the purpose of this thesis and the concepts defined in Chapter 4.

**Chapter 8 Conclusion and future work:** This chapter resumes the content of every chapter and provides conclusions of this thesis considering the objectives and the results achieved after implementing and assessing the prototype.

---

## 2 Background

---

In order to understand the problem stated in section 1.2 this chapter is composed of a *Terminology* section that defines the telematic business terms, and how they are related one to another. Consequently section 2.2 comprises the basis for Web maps. In addition, section 2.3 considers factors to evaluate software based on the ISO 25000: *Software product Quality Requirements and Evaluation as reference (SQuaRE)*.

### 2.1 Terminologies

There are some crucial definitions and concepts that are normally used in the telematic business and at FleetBoard. Thus, they are described in this section to clarify further concepts.

**Simulation:** It is defined as the imitation of real vehicles that use the communication platform (see Figure 1.1) for exchanging data. It encompasses vehicles behaviour with their variables such as GPS positions, number of stops, driving time, speed and some others, and the delivery of messages. The benefit of executing such simulations is to test the quality and performance of new features deployed on the telematic platform.

**Fleet:** A set of vehicles constitutes a *fleet*. At FleetBoard this term is used for commercial purpose to offer packages of products and services, but also for technical purposes. During the testing process an entire fleet is used for measuring the performance and the quality of the telematic platform.

**Vehicle:** While for customers a *vehicle* is just the actual truck, van or bus, at FleetBoard this comprises specialized hardware devices that collect and send messages to FleetBoard's central message queue servers. The communication is established using a private protocol in order to begin an effective interaction among vehicles and the central message queue servers. The term vehicle is also a generalized term for truck, van and bus.

**Route:** During a journey a real vehicle collects geographical positions (latitude and longitude). The ordered set of these positions from the beginning to the end of a real journey are considered a *route*, from which LVS uses them to follow a path. Hence, a route is a set of already generated geographical positions.

**Trip:** A *trip* contains several components: routes, vehicles that follow routes, and drivers. Concerning the simulation, the trip stands for the complete set of generated data that belongs to a particular simulated journey.

**Messages:** Data conveyed from the vehicle to FBSC comprise *messages* with special function one to another. Every message is equivalent to a *datapacket* (DP) identified with a number according to its functionality, e.g., DP 255 contains information regarding geographical positioning.

**Tour:** A *Tour* is defined from three different perspectives; customer, company and simulation perspective. Although only the last two perspectives are subject of this thesis, all three are

explained because they contain several similarities and offer a wider perspective of how this term is used in different scenarios.

- a) FleetBoard's customers define a *tour* as the plan of the day for a vehicle on a specific day to bring specific goods from particular customer A to customer B.
- b) From the FleetBoard's perspective, a *tour* is a broader context because it contains not just a route, but also components such as a vehicle that generates coordinates at the instant that this starts driving, a constant period to send encapsulated messages in form of DP to FBSC, and the driver. The Figure 2.1 depicts this definition.

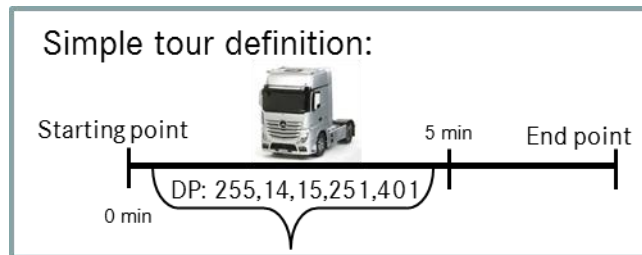


Figure 2.1: Company's perspective: tour definition

- c) From the simulation's perspective a *tour* is defined as a template for a set of trips; this contains the same information stated in the FleetBoard's perspective, however the purpose is different, because for every simulation a new trip is created based on a tour which acts as a template. i.e., if a simulation requires driving 100 vehicles, it is necessary to create 100 trips using just one tour, hence, an entire fleet can be tested with a similar behaviour. Additionally, during the trip creation DPs are generated and stored into LVS to send them during the simulation to FBSC.

**Trace:** A geographical position is collected and calculated by a GPS receiver (see section 2.2.3), and at the same time every position is stored in the vehicle's hardware in the form of latitude and longitude, thereafter, vehicles send this data every constant interval with the current date and time. Hence, what it is received by FBSC platform is a *trace* that contains a set of coordinates with certain date and time. In the literature the term *trace* is also known as *track*, which is an element of the GPX file format (see section 2.2.4) [Tu06].

**Protocol:** The communication between vehicles and FBSC is performed by using the private *FleetBoard Protocol* to increase the security level throughout an encryption mechanism, and minimize traffic by reducing the overhead of the messages to a minimum. The DPs' specification is also found in this protocol.

**Event types:** There are some *event types* that are considered during the transmission of DP from vehicles to the message queue server. Those *event types* represent the state of a vehicle or driver. They are important for this thesis, because they contribute to the delimitation of a single tour with regard to the starting and end point, e.g., drive start and drive end are *event types*.

## 2.2 Mapping

### 2.2.1 Geographical coordinates

Coordinates represent locations points. They are expressed using latitude and longitude, where latitude measures from south to north, and longitude measures from west to east. The representation is by means of the Cartesian plane. This is built based on the equator line, where the latitude is zero, and used as reference to define the positive and negative points. Likewise, the Greenwich England line is considered as longitude zero reference [Sven10]. The Figure 2.2 illustrates how the geographical coordinates are built it up.

### 2.2.2 Waypoints

They are coordinates that identify a physical space [Tu06], in that way, they represent intermediate points on a route, including starting and end points. For example, Figure 2.2 depicts one waypoint, which is in the centre of the coordinate system, (0, 0).



Figure 2.2 Elements of geographical coordinates

### 2.2.3 Global Positioning System (GPS)

It is a satellite network that provides three dimensional locations, latitude, longitude and altitude; one of the most important functionalities consists of offering geographical coordinates, also known as geo-position. GPS is operated by a high altitude satellite that broadcasts position and time to GPS receivers that determine their position and time based on the information received from the satellite [Tu06].

FleetBoard's vehicles contain a GPS receiver that allows vehicle tracking, these tracking information is stored in the special hardware installed in vehicles (see section 3.2), and send it to FBSC every constant period.



### 2.2.4 GPS Exchange Format (GPX)

It is a XML file format that contains coordinates. The advantage of this format is that guarantees an easy way to process and convert geographical data to other formats. Waypoints, tracks and routes, are essential components of the GPX format, including the metadata that represents the namespaces. Figure 2.3 shows the structure of the GPX format.

```
<gpx version="1.0">
  <name>Example gpx</name>
  <wpt></wpt><!-- Waypoints -->
  <rte></rte><!-- Routes -->
  <trk></trk><!-- Tracks -->
</gpx>
```

Figure 2.3 Global schema of the GPX elements

### 2.2.5 Web Map Service (WMS)

WMS is a geographical application based on the specifications of Open Geospatial Consortium (OGC) and ISO/TC211 to create and display maps that come from heterogeneous and distributed systems. These maps provide standard images such as JPEG, PNG, SVG, among others [LZ05].

The communication is supported by a protocol that provides operations to obtain the description, features and rendering of maps, it lets the WMS provider to expose accurate responses based on well-defined HTTP requests using standard Web browsers or Web services clients. The specification of this protocol is defined by OGC to maintain interoperability and integration among providers (Google, Nokia, OpenStreetMap and some others) as well as internet users.

### 2.2.6 Geographic Information Systems databases (GIS)

They are distributed data management systems to increase the performance and quality of the responses, they are able to support geometry data (points, lines, graphs and some additional data) and process huge amounts of data. There are some alternatives to store information in local servers for personal purposes; however it is necessary to provide an optimal infrastructure for processing it. This option is given by OpenStreetMap. In normal cases the storage is hosted by the main providers (Google Maps, Nokia, and OpenStreetMap).

### 2.2.7 Tiles

A map is made up of many images called tiles; they are displayed in a grid arrangement and closed each other so they appear as a single image. So every time an end user scrolls up or down new tiles are loaded, it means that every zoom level retrieve tiles from WMS [Tu06].

Consequently, a Map Tiles Server has the function of retrieving images from distributed databases (see section 2.2.6) and sending them as response based on requests containing

coordinates, addresses, and other sort of information associated to Web maps. Requests are allowed only using a map API (see section 2.2.8).

### 2.2.8 Map Application Programming Interface (API)

They are online libraries that gather information from different sources to manipulate and display information about maps. [SW12]. This map API contains functions that control the appearance of a map, including size, shape, position and some other features.

The map API offers an Asynchronous JavaScript and XML (AJAX) as engine to interact to Web map Servers asynchronously. Thus, every zoom in/out on Web maps is requested to the server since new tiles need to render, thereafter the map API conveys data to browsers to display a map using HTML and CSS formats. Moreover, API provides a number of utilities and services throughout the main object called map. This map API is purely object oriented, where any functionality is reached by instantiating objects [CHO08].

### 2.2.9 Open Geospatial Consortium (OGC) Architecture

According to the OGC [ISO 19142], the WMS and maps APIs providers should implement a common architecture for conveying and storing spatial data to guarantee interoperability and flexibility among different applications. Figure 2.4 portrays the standard architecture provided by OGC.

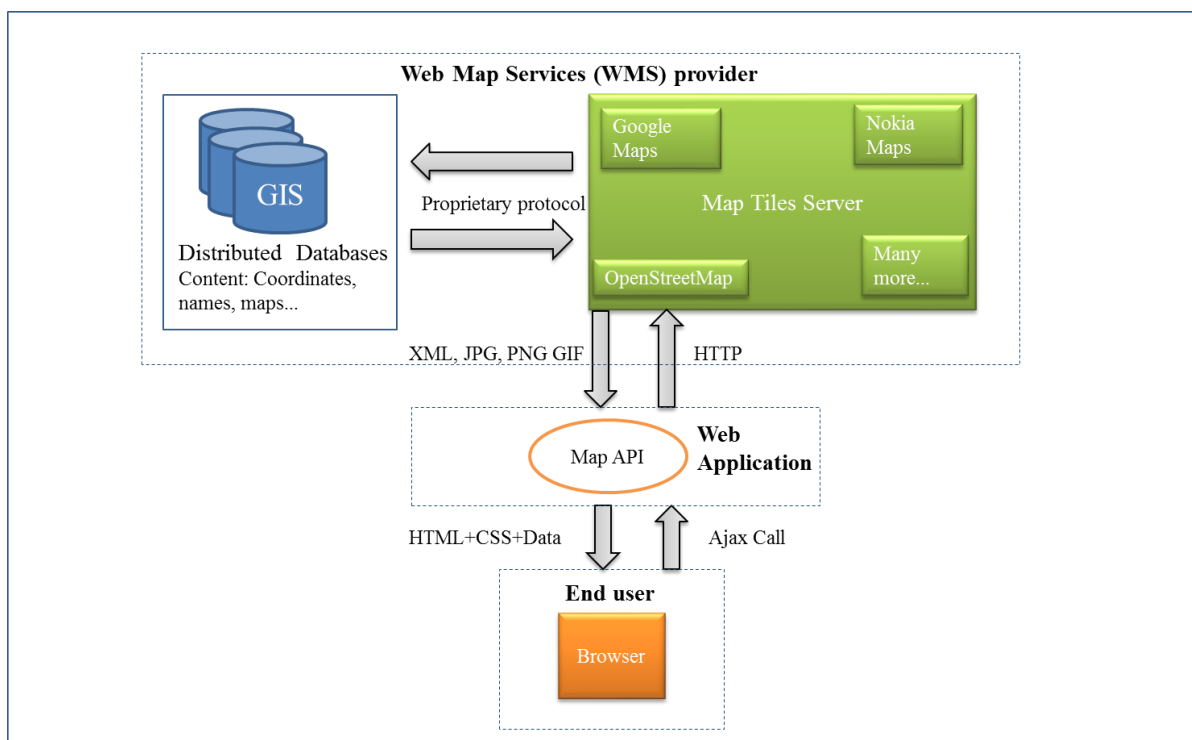


Figure 2.4 Maps: system architecture

This architecture comprises an *End User*, who uses a browser to make queries, a *WMS* to deliver and send responses, and a *Web Application* to host map API. The interaction among these three

fundamental components assumes a priori communication between browser and Web application that allows further asynchronous calls using a map API for sending HTTP requests to the WMS, upon delivery the request, the *Map Tiles Server* processes and forwards the request by means of a proprietary protocol that depends on the WMS provider, consequently *distributed GIS databases* retrieve the information and send it back to the *Map Tiles Server* to process and serve the response in the appropriate representation format, finally a map is rendered in the Web browser using HTML, CSS and information data (see Figure 2.4).

### 2.2.10 GPX Visualizer

It is a free and online Web application [SCH13] to create maps and profiles based on simple coordinates, waypoints, driving routes and results from Google WMS. It is suggested as solution instead of using the optimized responses generated by Google Maps (see section 4.2.2) [SCH13], e.g., Google only generates initial and end waypoints, while the GPX visualizer generates additionally waypoints in between.

## 2.3 Quality model ISO 25000 and 9126

The international standard ISO 25000: *Software product Quality Requirements and Evaluation* (SQuaRE) contains some reference models and standards for guiding the application of new standards into software [ISO25000]. These are used in this thesis to evaluate software offered by different WMS providers.

SQuaRE is based on the ISO 9126: *Quality Model*, from which this thesis extracts the criteria to evaluate functionalities of software. The *Quality Model* is composed of a frame of reference to describe the quality factors of any software; this comprises four parts:

- 1) ISO/IEC 9126-1: Quality model
- 2) ISO/IEC TR 9126-2: External metrics
- 3) ISO/IEC TR 9126-3: Internal metrics
- 4) ISO/IEC TR 9126-4: Quality in use metrics

The first part describes the framework and the relationships between the different approaches, the second describes the external quality to evaluate the execution quality. The third covers the internal quality to assess the quality of the code, and the last one describes the metrics used for combining the three previous parts considering the user point of view [ACK05]. This thesis considers only the following criteria: functionality, reliability, usability and efficiency, which are part of the quality model *ISO/IEC 9126-1*.

The quality criteria according to the ISO 9126 Quality Model [ISO9126] are defined in the following table:

---

<i>Functionality</i>	It focuses whether the required functionalities are available in the software or not.
<i>Reliability</i>	It focalises on the capability of software to maintain the performance under certain circumstances, even if failures occur.
<i>Usability</i>	It concentrates whether the functionalities are easy to use or not.
<i>Efficiency</i>	It focuses on how the performance is between the software and the resources.

Table 2.1 Criteria according to the Quality model ISO 9126

These criteria and their descriptions are considered in this thesis to evaluate WMS providers and select one of them based on the concept and design (see section 4). The implementation, section 5, takes the result of this evaluation to develop the prototype.

---

## 3 Systems and communication architecture

---

This chapter provides a basis for FBSC, Integration Test System (ITS), LVS, and the communication architecture that are used in daily operations at FleetBoard. A deep analysis is considered over LVS, because the prototype is based on it. Thus, features, dataflow and pattern of design are analysed to comprehend the complexity of this system and its interactions.

### 3.1 Architectural overview

The architecture comprises FBSC to receive messages from real vehicles and provide services to customers (see section 1.1.4), and Telematic Platform (TP) to collect data in vehicles and send it to FBSC. In addition, a parallel architecture exists with the purpose of testing and simulating the current and new features in FBSC. The benefits of having this parallel system encompasses reduction of unexpected behaviours or errors in the production environment, more quality in the services provided to customers, and real trucks' hardware are not used for testing purposes, that also minimize costs.

The parallel architecture contains Integration Test System (ITS) that substitutes FBSC, and LVS that replaces real vehicles. ITS is used for testing FBSC and LVS to simulate vehicles. Running both systems allows measuring of FBSC's performance and guarantees that FBSC's functionalities work properly with high quality. Figure 3.1 depicts the architectural overview.

The process of adding new features on FBSC considers first a deployment of these features in ITS, then old and new features are tested using several testing tools, from which LVS is also included. Thus, if results are successful the new functionalities are deployed in real environment and customers are willing to use them. In case of having errors or weird behaviours new functionalities are corrected and a new test process starts.

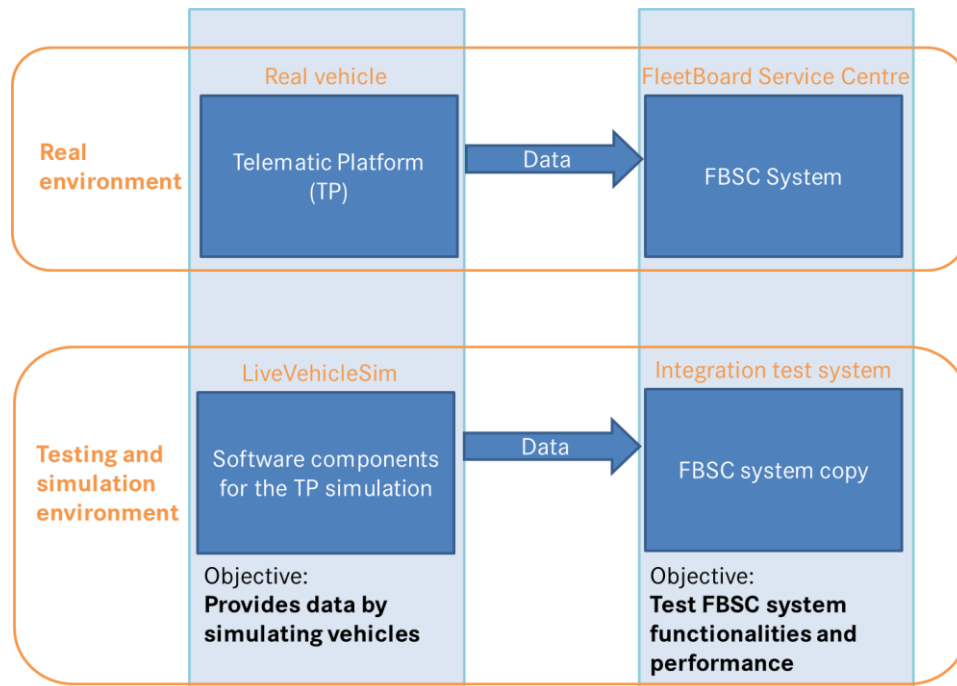


Figure 3.1 General system architecture overview

In this thesis, the development and implementation of the prototype consider the testing and simulation environment. In that way, LVS is used to reach more realism in the simulations, and ITS will be utilised to receive messages and to import historical data from real vehicles to LVS.

### 3.2 FBSC and communication platform

It is essential to describe the main components of the communication architecture to understand how the entire system works. The main components are illustrated in the Figure 3.2, and described in the following paragraphs.

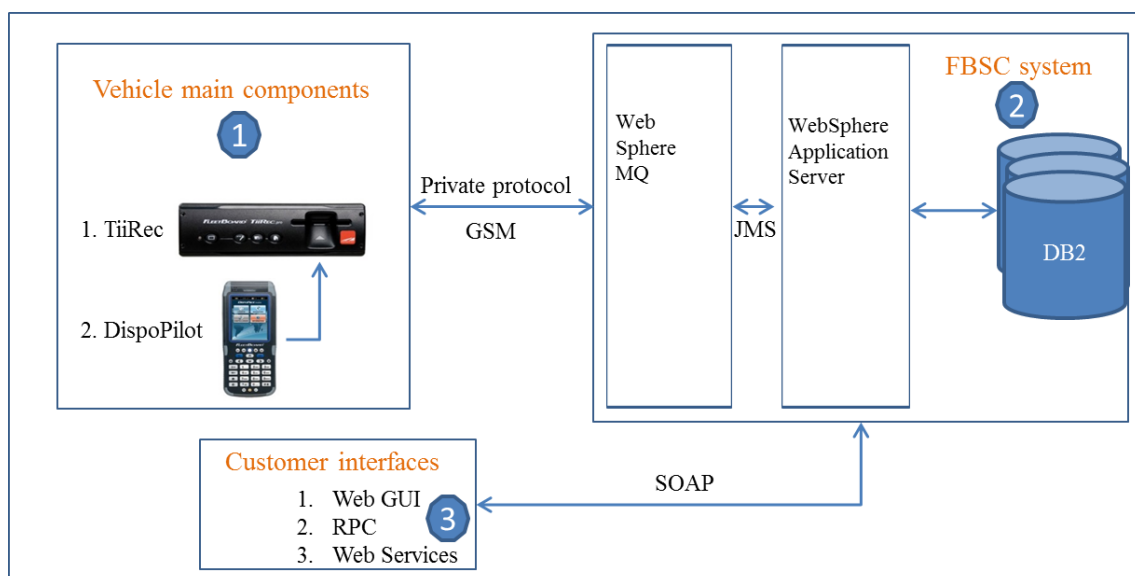


Figure 3.2 Communication and systems architecture

The communication platform refers to *vehicles' main components* (1), *FBSC* (2), and the *customer interfaces* (3), these three prominent parts define a peculiar interaction using FBSC as central repository of information, Figure 3.2 . The interaction is performed using the GSM network and the FleetBoard's protocol to establish communication between vehicles and FBSC. SOAP protocol is used between the customer interface and FBSC mostly to manage fleets through the services offered by Daimler FleetBoard, see section 1.1.4.

Vehicles comprise two main hardware sub-components (see Figure 3.2); one is the *TiiRec*, also called *TP*, and its main function is to collect data related to vehicles and time management services, see section 1.1.4 . The second component is the *DispoPilot*, its main task is to provide logistics management services, see section 1.1.4, and help drivers with the map navigation. Both the *TiiRec/TP* and the *DispoPilot* are installed in the driver's cabin.

Figure 3.2 depicts the interaction among the *DispoPilot*, the *TiiRec* and *FBSC*, when information is generated by the *DispoPilot*, the *TiiRec* receives and sends this information to *FBSC*. Therefore, exchanging information is only possible between the *TP* and *FBSC*.

According to Figure 3.2, the second main component of the architecture is *FBSC*, in which three sub-components are important to know. The first element is the WebSphere MQ Server; this has the function of receiving all messages that are sent by the *TP*, thereafter those messages are stored in a queue, and then forwarded to the WebSphere Application Server (*WAS*) using Java Message Service (*JMS*) as messaging standard.

The second component is the *WAS*, that is responsible for processing and storing the information into the database. Furthermore, the *WAS* is in charge of providing services to customers by means of SOAP interfaces which are requested using RCP application clients and Web services. The Web GUI is accessed using HTTP.

The third component is the *DB2* database, its main function is to store messages and private information related to customers. Technically this database is distributed in a cluster manner to optimize processing of data and minimize the response time for requests. *LVS* accesses this database throughout a JAVA Application Programming Interface (*API*), which is called Test *API* which is used for importing information related to fleets, vehicles and drivers.

Customers' interfaces are provided as Web GUI, RCP Clients or custom user interfaces connected to FleetBoard via SOAP to retrieve information from *FBSC*. The Web GUI is used to visualize information related to fleets, drivers and the available services (see section 1.1.4) offered by Daimler FleetBoard to customers. However, this Web GUI is being replaced to RCP, which are applications installed at customer side. Web services interfaces support RCP applications to retrieve information from *FBSC*, e.g., messages generated by vehicles in a certain range of dates can be visualized it. These mechanisms are listed in the *Customer interface* component in the Figure 3.2.

The relevant parts of the telematic architecture to be considered in this thesis are the *vehicle* and *FBSC* copy in the ITS. However, the *vehicle* and its main components are replaced by *LVS* for testing purposes; the section 3.4 explains this in details.

### 3.3 Integration Test System (ITS)

ITS is a copy of FBSC, which is a replica of the software from real environment, the hardware has some variations. This replica is used for testing new and current functionalities to avoid unexpected behaviours or results in the real environment. The performance analysis is also performed in ITS to measure the capacity of response and stability under a particular workload.

The technological architecture is the same as FBSC shown in the section 3.2, but the only difference is that both have different purposes. FBSC is for providing services to customers and ITS for testing and performance analysis.

### 3.4 LiveVehicleSim (LVS)

LVS is a software accessed via a Web GUI that simulates vehicles' behaviours. This covers the delivery of messages to FBSC and the generation of data regarding time and vehicle management (see section 1.1.4). The objective of this simulator is to support the tests on FBSC. When new features are added to FBSC, the failures that reach the customer may be dramatically reduced, because the simulator acts as real vehicle and the functionalities of FBSC can be tested before customers make use of the platform. This is considered a great benefit to the company.

The interaction between LVS and FBSC is performed in the local network instead of the GSM network, and the private protocol is still used. Every message received by the message queue server is processed as if it were real vehicle. Currently, more than 600 vehicles are simulated at the same time using LVS, in that way, functionalities, availability and performance of FBSC can be tested. In Figure 3.3 this interaction is visualized.

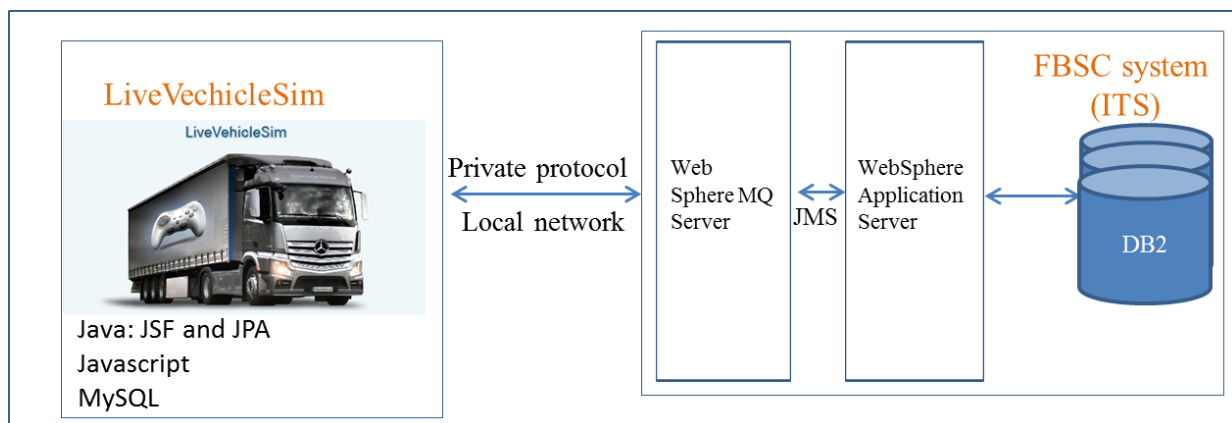


Figure 3.3 Systems and communication architecture for testing

#### 3.4.1 Features

The functionalities of LVS are found directly on the Web application; therefore they are described based on the site map, see Figure 3.4. Functionalities are accessible through the login



Web page (index.html) using the private network, and a valid user with his corresponding password. Currently this only supports Mozilla Firefox to navigate along the Web site.

The dashboard interface (dashboard.html) contains the three fundamental application modules, the tour administration (tourmanagement.html), the trip administration (trips.html) and the Messages administration (31.html).

The tour administration module is used for creation of tours; this feature lets tours to be created immediately or scheduled for its execution in the future. This is the most important feature, because the simulation is configured and planed in this module. Four additional sub-modules are found for the manipulation of vehicles (vehicles.html), drivers (drivers.html), fleets (fleets.html) and routes (routes.html). The Web GUI that the prototype uses to meet the objectives of this thesis is the routes administration, the fields that comprises this Web GUI are name, description, initial and end destination and GPX that contains a set of waypoints.

The feature of the trips module consists of displaying information in detail regarding trips that are in state running, finished and pending to run. In any of these states the information shown comprises the vehicle with his drive and the messages that will be sent throughout internet. This feature acts as reporter.

Within the messaging module the main feature consists of the manual creation of messages, which are supported in the user's selection of a datapacket in the Web GUI, thereafter a message is built up and sent it to the queue server. The information content of every datapacket in this module is statically predefined.

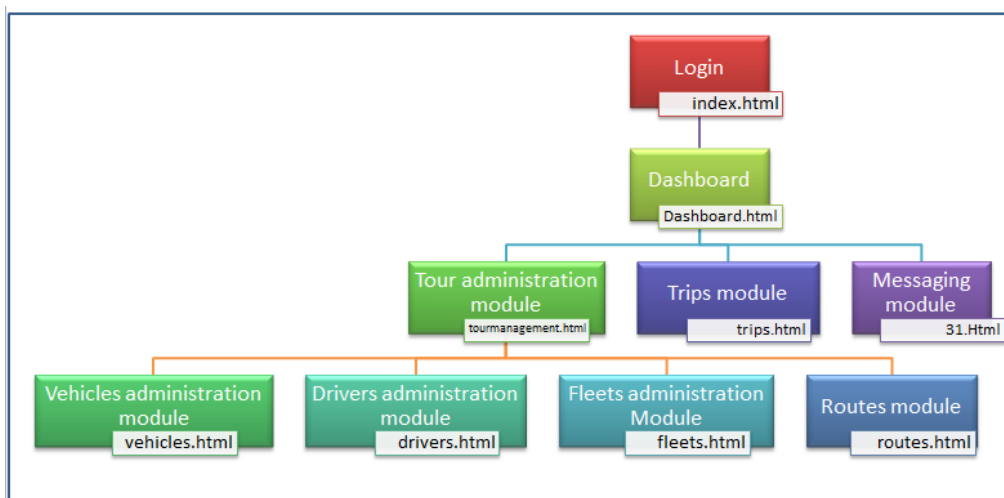


Figure 3.4: Site map

### 3.4.2 Dataflow

Data Flow Diagram (DFD) representation is applied in this section to describe how data flow among LVS, FBSC and Google Maps. In that way, inputs, outputs and processes that comprise LVS are understood.

DFD (see Appendix A) describes graphically a system from general to specific perspective; this description is based on the inputs, outputs and processes. Decomposition technique is essential

to start from a high level of description until a desired level of detail is obtained [AG92]; basically, this consists of defining the main functionality or process of the system as top level, from which other functionalities are defined in further sub-levels. Figure 3.5 illustrates two levels of description. Level 0: start tour process and level 1: import tour process with their corresponding inputs and outputs. The following paragraphs explain the DFD in detail.

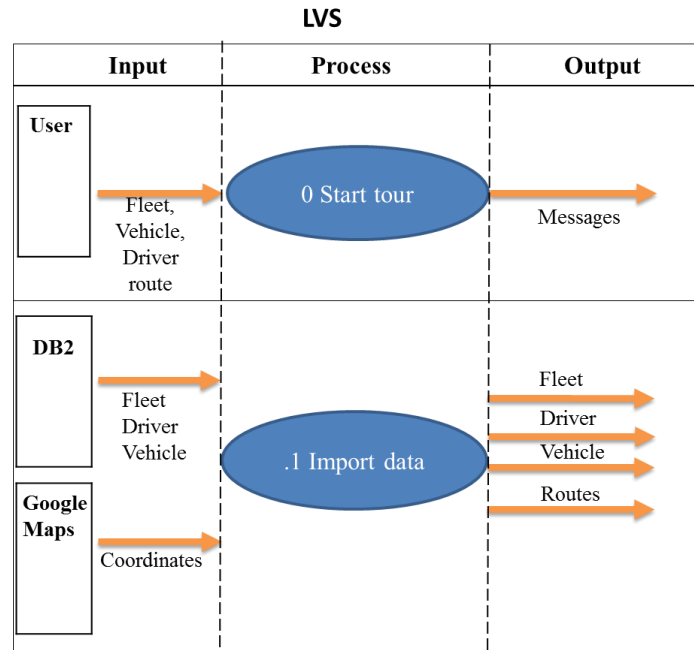


Figure 3.5: LVS Dataflow Diagram

The dataflow during a simulation in LVS consists of inserting data that is set when users select a specific fleet, vehicle, driver and date on the tour administration module. In this way, the tour simulation is performed with the purpose of letting a vehicle run, and a group of messages are generated as output of this process. A prerequisite to run the simulation of a tour comprises the importation of data regarding fleet, vehicles, drivers and coordinates from Google Maps.

The process of importing data considers as first source of information the DB2 database which provides data related to vehicles, drivers and fleets. The second source of information is Google Maps that provides waypoints to define routes. The output data of this process are fleets, vehicles, drivers and routes that are used in the tour simulation (see Figure 3.5).

Figure 3.5 depicts the dependency on the general start tour process and the importation of data processes as part of it. The tour administration module, see section 3.4.1, is used to operate these two processes.

For this thesis, the process of importing routes is analysed to create a new method to import routes close to the reality and optimize the actual process by integrating WMS into LVS. Hence, data generated by Google Maps are no longer needed, because current routes do not represent customers' behaviour.

### 3.4.3 The Model View and Controller pattern (MVC)

The MVC is an architectural model for software systems. The main objective is to isolate three different layers: *the model layer* for the data storage manipulation, *the view layer* for the data representation into user interface, and the *controller layer* in charge of the business logic or user interaction [LR01]. The benefit of applying this model leads to obtain a software application, easy to develop, modify and maintain [CL09]. LVS is supported on this model.

The interaction among these three layers starts when users send HTTP request to WAS using a Web browser, the controller acts as central unit of processing, this receives the request, performs the needed operations, retrieves data from the model if it is necessary, and finally sends the pertinent information to the right view for rendering the information in the Web browser [LR01]. The following sections explain in detail how LVS performs its operations using these three layers.

#### 3.4.3.1 Model

In this layer LVS employs Object Relation Mapping (ORM), which allows to apply analysis and design of oriented objects, improving the application performance when limits of memory in a relational database are reached [LZ10], and uses the Java Persistence API (JPA) as framework, to persist model objects to a relational database and retrieve them [BKS11].

The relational database management system (RDBMS) is MySQL, and is accessed through JPA using Java Persistence Query Language (JPQL) to interact with the relational database, access and manipulate data [Va08]. The controller performs the creation, update, retrieve and delete (CRUD) operations using Data Access Object (DAO), which are interfaces exposed by the model layer assuring a transparent access to MySQL [BKM04]. Figure 3.6 portrays the general overview of this layer used by LVS.

#### 3.4.3.2 View

This layer defines the Web GUI or Web pages; it is basically, what users see. Java Server Faces (JSF) is the technology used for rendering information into Web browsers; this is well-known as Web application standard Java Framework technology [LS11].

JSF uses a standard API for rendering Web applications user interfaces (UI), from which the UI components are defined, including their events and validations. This technology also comprises the MVC pattern, in which UI components are supported on the model layer to operate and implement the business logic, the view layer to exchange information among users, and the controller layer to control the Web applications from users' requests until a Web page is rendered using a special servlet named *FacesServlet* [LS11].

JSF defines a process with 6 phases for displaying the information in Web browsers, (1) *Restore view* is triggered when an end user clicks on a link or button, (2) *Apply request values* takes the values of every component in the Web application, and are updated in the application server, (3) *Process validation* evaluates the properties of every component based on the values of the attributes, if an error comes up, the process changes to the last phase, (4) *Update model values* bring values to the backing beans in the server, , (5) *invoke application* considers the application

events and passes the control to the next page, according to the navigation rules of the application, and (6) *Render the response* which displays the pertinent information to users [Hi05]. Figure 3.6 depicts these six phases.

### 3.4.3.3 Controller

Reading and validation operations are performed by this layer to determine the business logic needed by the end user, and retrieve information from the interfaces exposed by the model layer, if necessary. In addition, this layer selects the next view users should see, including the response. Figure 3.6 illustrates this intermediate layer and its interrelations.

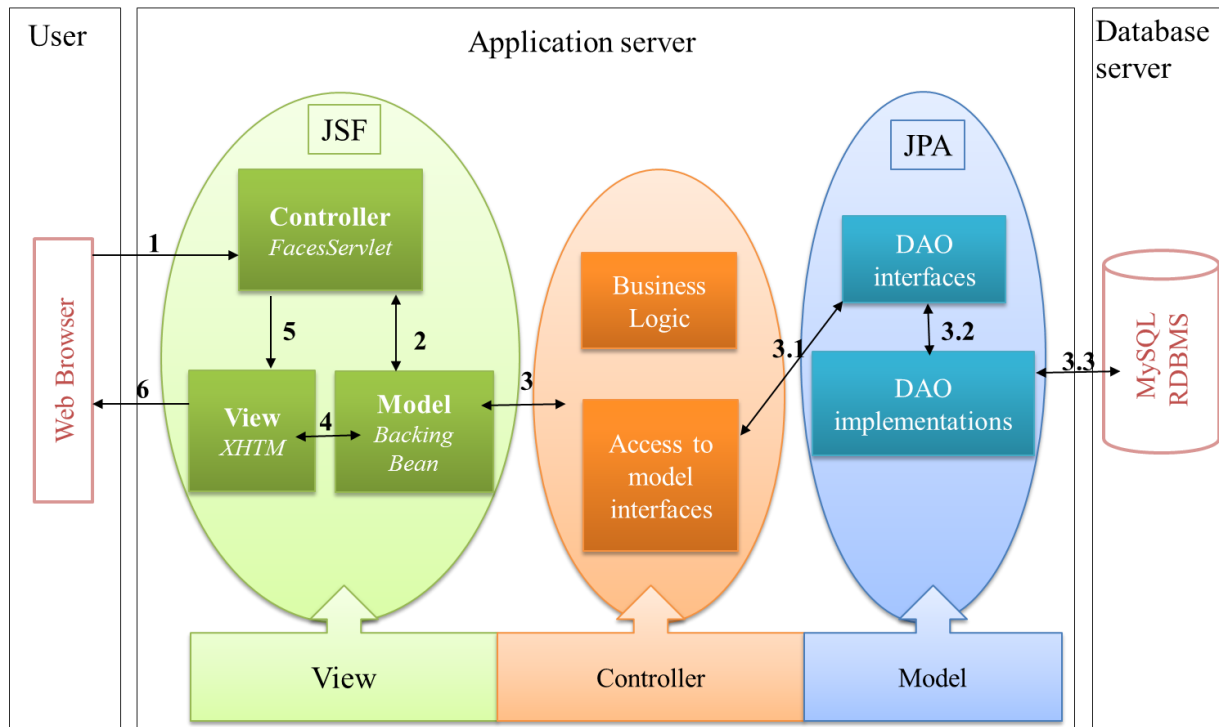


Figure 3.6 LVS and the MVC design pattern

### 3.4.3.4 Application and database server

Figure 3.6 shows the application server with LVS application, from which the MVC layers are clearly separated according to their functions. The database server contains MySQL that is in charge of storing physically data. Users also represent an important component in the architecture, because by means of a Web browser they send requests, and receive responses to and from the application server.

---

## 4 Concept and design

---

This chapter provides the analysis and two solutions to address the purpose of this thesis defined in section 1.3. The first solution encompasses a new process to create routes using data generated by real vehicles. The second solution provides a mechanism to create routes by setting a Web Map. Functional requirements are defined to support these two solutions.

### 4.1 Concept to incorporate real routes in tour simulations

This section defines a concept that comprises an analysis of the current process of creating routes. Subsequently, an algorithm describes the new process to create routes with more realism in tours simulations. Functional requirements are also defined to support the implementation of the prototype, these are based on the algorithm previously defined.

#### 4.1.1 Analysis

The process of running a tour simulation consists of generating data close to the reality for acquiring behaviour similar to real vehicles, with the objective of testing functionalities of FBSC. Currently during the execution of a tour simulation LVS follows routes manually created using Google Maps to simulate vehicles' movement. These routes do not represent any relation to the reality, i.e., none vehicles have driven over these routes, so that, actual simulations lack of realism, and the cause lies on using Google Maps as source to create and import routes.

In reality, every vehicle follows unique routes, even if a vehicle follows the same roads daily with same initial and end point, they differ in details, because a waypoint is not always collected at the same position and at the same time by the TiiRec, it varies from one to another. In that way, there must be more routes than vehicles in a simulation, which is not true for LVS. For this reason, it is essential to find a solution that provides the amount of data necessary to generate tours automatically with unique routes.

The following approaches encompass the problem statement defined in section 1.2, from which it is stated the lack of realism of tours simulations, and the non-existence of an automatic process to create amounts of routes that satisfies the number of available vehicles.

#### 4.1.2 Approaches

In this section, a solution tackles the problems stated in the analysis section by providing a new process and technical description to support the specification (see section 4.1.3) and a suitable implementation (see section 5) of the prototype. A unique approach is considered in this section, because the solution contemplates the closest datasource to the reality and FleetBoard's customers.

##### 4.1.2.1 The new process design

Considering that the messages generated by the real vehicles are sent to FBSC to be processed by the message queue server, and finally stored into FBSC's database, the appropriate source for

importing routes is FBSC's database, because this contains real routes that vehicles generate continuously. With this solution tours simulations use and generate data close to the reality.

Furthermore, the import of routes from FBSC's database and loading them into LVS' database constitutes a process of bringing large numbers of routes, which solves the manual execution and lack of generating large amounts of routes. Figure 4.1 represents the new routes flow from FBSC's database and LVS' database, from which exists an intermediate step for transforming the real routes into routes ready to use by LVS.

The transformation of routes is essential because FBSC's database stores waypoints continuously without indicating the initial and the end waypoint of a single route. This inconvenience is solved by establishing a clear limitation using the *event types records* (see section 2.1) generated by real vehicles to distinguish when a vehicle starts driving or stops.

Another inconvenience is that FBSC's database stores waypoints in different tables depending on the services that vehicles contract with Daimler FleetBoard, if *Track and Trace* service is part of the bundle of services, the table used is *gpstracedata*, otherwise *gpsdata*. Although, these two tables contain waypoints, it is important to differentiate them to retrieve the right information. The difference between them lies in that every entry of the *gpstracedata* table stores several waypoints collected every 30 seconds, meanwhile every entry of the *gpsdata* table stores only one waypoint, which is generated every 30 minutes. Figure 4.1 describes the interface in FBSC to access these tables.

Figure 4.1 depicts the summary of the new process model of importing routes, in which, FBSC data layer provides an interface to access *gpsdata* table that belongs to the Basic Service and *gpstracedata* table that is part of the *Track and Trace* Service. The improvement of this approach includes the following steps: (1) creation of an algorithm in the logic layer capable of interacting with FBSC's interface to import the waypoints, (2) transforming them into representable routes in a tour, and (3) loading them into LVS' *routes* table that is linked to the data layer from LVS. The order of execution of these three functions is shown in Figure 4.1, from which the three arrows represent the functions.

This new process model is executed by the end user in the view layer from LVS, and the routes are stored in LVS's database automatically for further simulations.

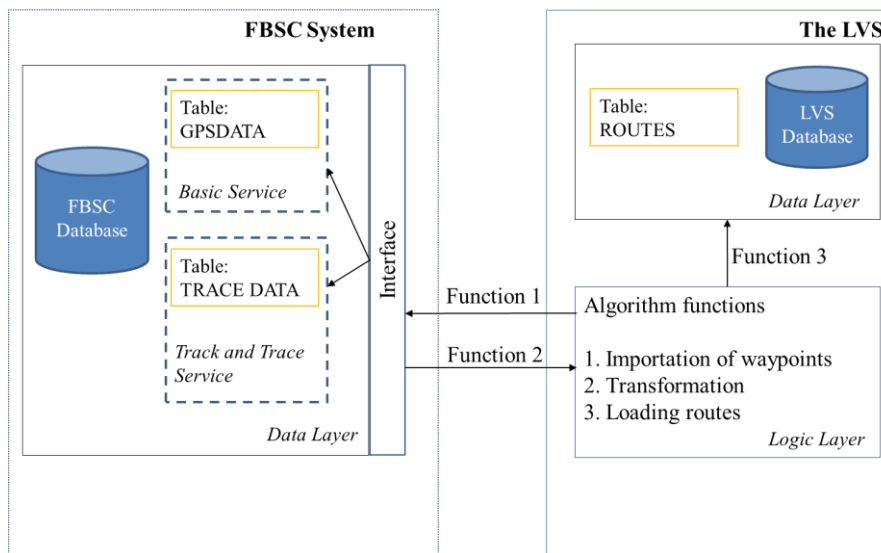


Figure 4.1. The new model for importing routes

### 4.1.3 Specification of the selected approach

Considering the MVC from section 3.4.3, the objectives exposed in section 1.3 and this approach, the functional requirements are described in this section.

The functional requirements are specified using *use case* diagrams from the Unified Modelling Language (UML). Figure 4.2 depicts the use cases to be implemented in this thesis. They are distributed according to the MVC pattern and include the user and FBSC as actors. The sequence of execution of every functionality is determined by the number assigned to the use cases. For example, if user creates a route, three use cases are included, create routes (1), import routes (2), and store routes transaction (3).

#### 4.1.3.1 Use cases

Use cases represented in Figure 4.2 are divided in three layers: *view* that represents the functionalities provided to users in the Web GUI, *controller* that relates to the business logic to support the *view* or *model* layer, and *model* that encompasses the operations to manipulate data using the database. Thus, the functional requirements are classified using these three layers and the UML notation to describe them.

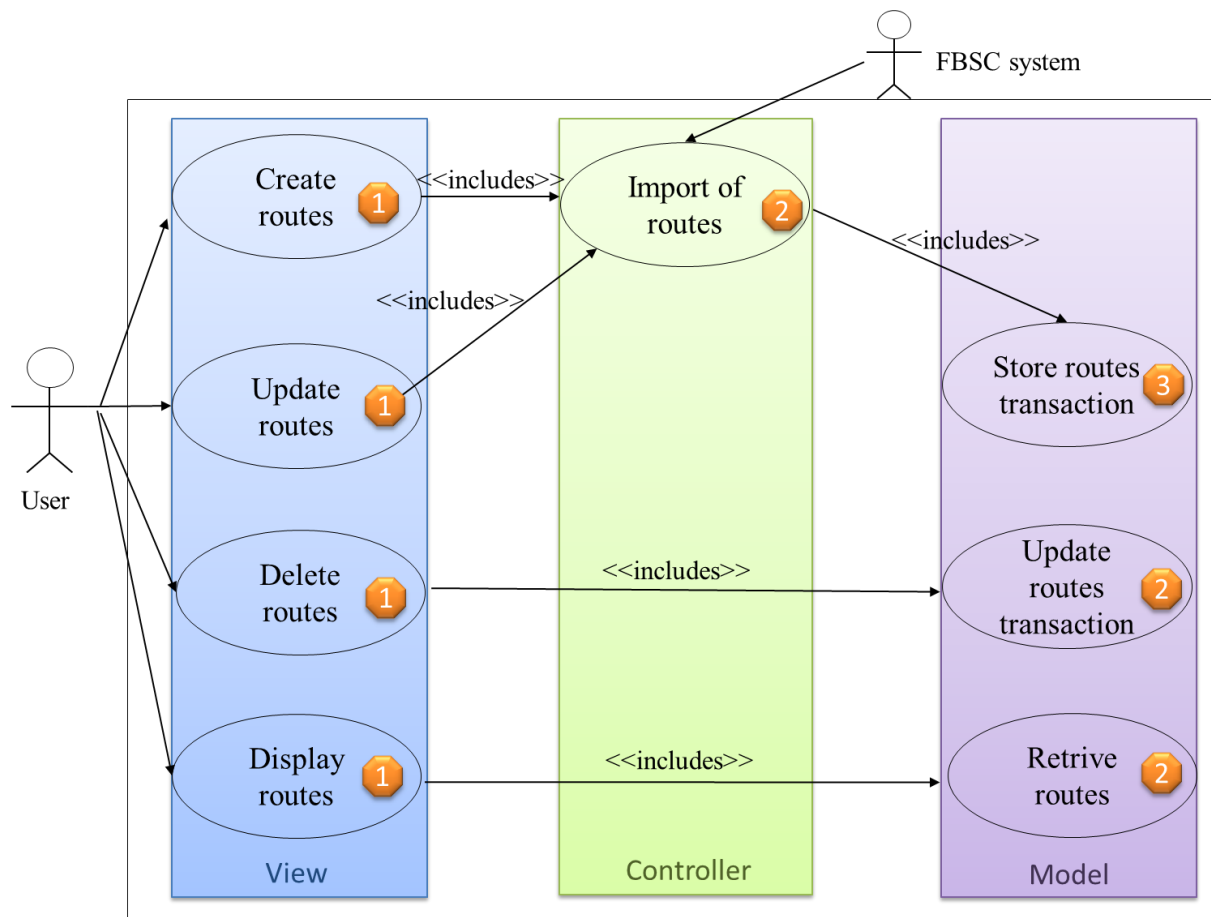


Figure 4.2 Use cases considered in this thesis

### View layer

The following tables (4.3 and 4.4) contain the *use cases* that represent the new functionalities on the Web GUI.



<b>Name</b>	Create routes
<b>Goal</b>	Users fill out the fields from the Web GUI to create a route object
<b>Actor</b>	Users
<b>Pre-Condition</b>	<ol style="list-style-type: none"> <li>1. The WAS is already running.</li> <li>2. Users are already logged in LVS</li> <li>3. Fleets are already created in LVS.</li> </ol>
<b>Post-Condition</b>	The controller layer receives the route created in the Web GUI
<b>Post-Condition in special case</b>	LVS displays a message with the exception
<b>Normal Case</b>	<ol style="list-style-type: none"> <li>1. Users add a new route.</li> <li>2. Route object is created.</li> <li>3. Users fill out the name, selects FBSC datasource and chooses the fleet.</li> <li>4. Save the route created.</li> </ol>
<b>Special cases</b>	<ol style="list-style-type: none"> <li>1. Missing mandatory fields. <ol style="list-style-type: none"> <li>a) LVS displays an error message and further operations are not performed.</li> </ol> </li> <li>2. No route is created</li> </ol>

Table 4.1 Description of use case *Create route*

<b>Name</b>	Update routes
<b>Goal</b>	Users fill out the fields from the Web GUI to update a route object
<b>Actor</b>	Users
<b>Pre-Condition</b>	<ol style="list-style-type: none"> <li>1. The WAS is already running</li> <li>2. Users are already logged in LVS.</li> <li>3. Fleets are already created in LVS.</li> <li>4. Route is already created in LVS.</li> </ol>
<b>Post-Condition</b>	The controller layer receives the route updated in the Web GUI
<b>Post-Condition in special case</b>	LVS displays a message with the exception
<b>Normal Case</b>	<ol style="list-style-type: none"> <li>1. Users select a route.</li> <li>2. Users update any field of the selected route.</li> <li>3. Save changes.</li> </ol>
<b>Special cases</b>	<ol style="list-style-type: none"> <li>1. Missing mandatory fields. <ol style="list-style-type: none"> <li>a) LVS displays an error message and further operations are not performed.</li> </ol> </li> <li>2. No route is updated.</li> </ol>

Table 4.2 Description of use case *Update route*

<b>Name</b>	Delete routes
<b>Goal</b>	Users select a route to delete
<b>Actor</b>	Users
<b>Pre-Condition</b>	1. The WAS is already running 2. Users are already logged in LVS 3. Route is already created in LVS.
<b>Post-Condition</b>	The controller layer receives the route selected in the Web GUI
<b>Post-Condition in special case</b>	LVS displays a message with the exception
<b>Normal Case</b>	1. Users select a route. 2. Users delete a route 3. LVS displays a confirmation message.
<b>Special cases</b>	1. No route is deleted.

Table 4.3 Description of use case *Delete routes*

<b>Name</b>	Display routes
<b>Goal</b>	Display the current active routes in the <i>route administration module</i> .
<b>Actor</b>	Users
<b>Pre-Condition</b>	1. The WAS is already running. 2. Users is already logged in LVS 3. Route is already created in LVS. 4. Users access the <i>route administration module</i> using the <i>tour administration module</i> .
<b>Post-Condition</b>	Through the controller layer routes are retrieved from the model layer.
<b>Post-Condition in special case</b>	LVS displays a message with the exception
<b>Normal Case</b>	1. The current active routes are displayed in the <i>route administration module</i> .
<b>Special cases</b>	1. No routes are displayed

Table 4.4 Description of use case *Display routes*

### *Controller layer*

This use case represents the new functionality on the business layer to import the routes from another datasource.

<b>Name</b>	Import routes
<b>Goal</b>	Import waypoints from FBSC's database, considering the parameters given by the upper layer.
<b>Actor</b>	FBSC
<b>Pre-Condition</b>	1. FBSC's database should be running. 2. The Test API should be correctly configured in LVS to access FBSC's database. 3. Available routes in FBSC's database
<b>Post-Condition</b>	The model layer receives the imported routes and uses them for storing in LVS' database
<b>Post-Condition in special case</b>	LVS displays the exception message
<b>Normal Case</b>	1. Controller layer is called from the upper layer and receives the parameter given by the user in the Web GUI. 2. Define the starting date and end date of the route. 3. Retrieve waypoints from FBSC's database that are in the interval defined in the previous step (2.) 4. Call the model layer and send the route with its waypoints.
<b>Special cases</b>	1. The vehicle has no waypoints associated to. 2. FBSC's database is off-line.

Table 4.5 Description of use case *Import routes*

### *Model layer*

These use cases represent the new functionalities on the data model layer.

<b>Name</b>	Store routes transaction
<b>Goal</b>	Store routes into LVS' database
<b>Pre-Condition</b>	1. LVS database should be running.
<b>Post-Condition</b>	Transaction committed
<b>Post-Condition in special case</b>	LVS displays the exception message
<b>Normal Case</b>	1. Model layer is called from the upper layer and receives a list with routes to store. 2. Perform a transaction for every route. a) Begin transaction. b) Save route c) Commit transaction.
<b>Special cases</b>	1. Transaction is not committed.

Table 4.6 Description of use case *Store routes transaction*

<b>Name</b>	Update routes transaction
<b>Goal</b>	Store routes into LVS' database
<b>Pre-Condition</b>	1. LVS database should be running.
<b>Post-Condition</b>	Transaction committed
<b>Post-Condition in special case</b>	LVS displays the exception message
<b>Normal Case</b>	1. Model layer is called from the upper layer and receives a list with the routes to store. 2. Perform a transaction for every route. a) Begin transaction. b) Save route c) Commit transaction.
<b>Special cases</b>	1. Transaction is not committed.

Table 4.7 Description of use case *Update routes transaction*

<b>Name</b>	Retrieve routes
<b>Goal</b>	Retrieve active routes from LVS database to display them into the Web GUI
<b>Pre-Condition</b>	1. LVS' database should be running. 2. Active routes are available in the database
<b>Post-Condition</b>	Convey a list of routes to the upper layer, consequently the routes are displayed in the Web GUI
<b>Post-Condition in special case</b>	LVS displays the exception message
<b>Normal Case</b>	1. A list of active routes is filled from LVS' database.
<b>Special cases</b>	1. Routes are not retrieved.

Table 4.8 Description of use case *Retrieve routes*

## 4.2 Concept to integrate a WMS provider into LVS

The main objective of this section is to find an optimized solution for the import of coordinates into LVS using a WMS. This solution contains an analysis section to be familiar with the current situation, the approaches section to evaluate the possible optimizations, and the decision section to decide on the best approach for the further implementation. Furthermore, the approaches section comprises an evaluation of the current WMS providers using the quality model ISO 25000 and 9126.

### 4.2.1 Analysis

According to the purpose of this thesis (see section 1.3), an optimization of the actual process of importing coordinates should be performed. The execution of this optimization requires the

manual operation of simple activities by means of different systems, which is a time-consuming process. In addition, routes generated from Google Maps are not close to real routes due to the optimization on routes performed by Google. Figure 4.3 illustrates this process with its sub-processes, activities and systems.

Figure 4.3 describes the current *importation and creation of routes* process, from which three sub-processes are divided according to the number of systems with activities associated to. Thus, the function of *sub-process 1* is to calculate the routes, this process is executed when the end user gives as input of information an initial and final place on Google Maps Web GUI. Thereafter, a request is sent it and processed it by Google. Finally a response is displayed and used for *sub-process 2*. The response from Google is a URL that contains the initial and final coordinates; this means that waypoints in between are not provided.

Since a real tour comprises not only of two coordinates, but several coordinates in between, *sub-process 2* converts the two waypoints given by Google Maps into waypoints in between, including the initial and end waypoints. The conversion is performed by using a bookmarklet, which is normally used to extend functionalities in Web browsers; in this case, bookmarklet stores a JavaScript command to include an external JavaScript library that allows transform the response from Google Maps into a set of waypoints. This set includes initial, end and intermediate waypoints.

The GPX Visualizer when users take the response from Google Maps and execute it on the GPX GUI; afterwards the result of this request is copied and used as input of *sub-process 3* (see Figure 4.3).

*Sub-process 3* operates the storing of waypoints into LVS database. This process starts when the end user pastes the waypoints given by the GPX visualizer, and executes the storing query. Thereafter it finishes with a confirmation response that is displayed on LVS GUI (see Figure 4.3).

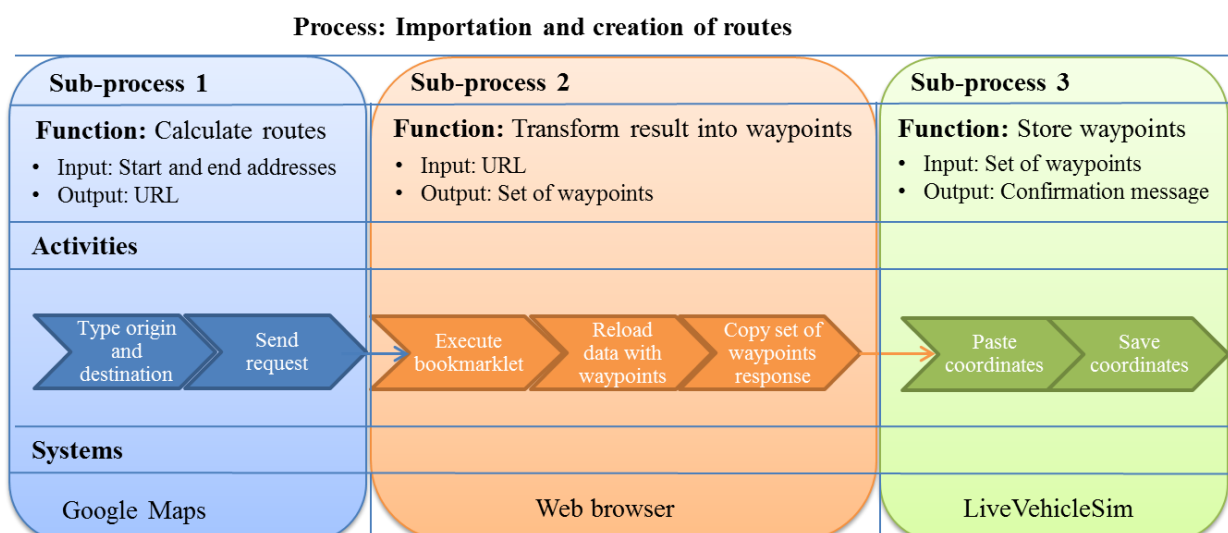


Figure 4.3: Current process to import routes

In general, this process consists of giving a name, description, starting point, end point and a set of coordinates that are used for tour simulations. After providing this information in the route administration module, the route is saved and ready for use on any tour.

## 4.2.2 Approaches

The following two approaches focus on the design of possible optimizations of this process. A subsequent evaluation is applied based on the following criteria: functionality, reliability, usability and maintainability (see section 2.3). Finally a decision is made to select one of the approaches.

### 4.2.2.1 Approach 1: merging calculation and conversion of routes

Based on the Sandbox feature for drawing a line on a Web map with the alternative to export it as waypoints offered by GPX Visualizer, this approach focuses on the optimization with removing *sub-process 1*, including the activities and systems that belong to it (see Figure 4.3). Since the calculation and conversion of routes are already integrated in Sandbox.

Figure 4.4 shows the optimization for importing routes. The input is the drawing of a route on a Web map, from which the calculation and conversion of routes is performed automatically by GPX Visualizer on its Web GUI. The output is a set of waypoints, which are obtained by calling the export function by sending http requests.

Storing coordinates on LVS database remains the same; hence, input, output, activities and LVS are not modified in this approach.

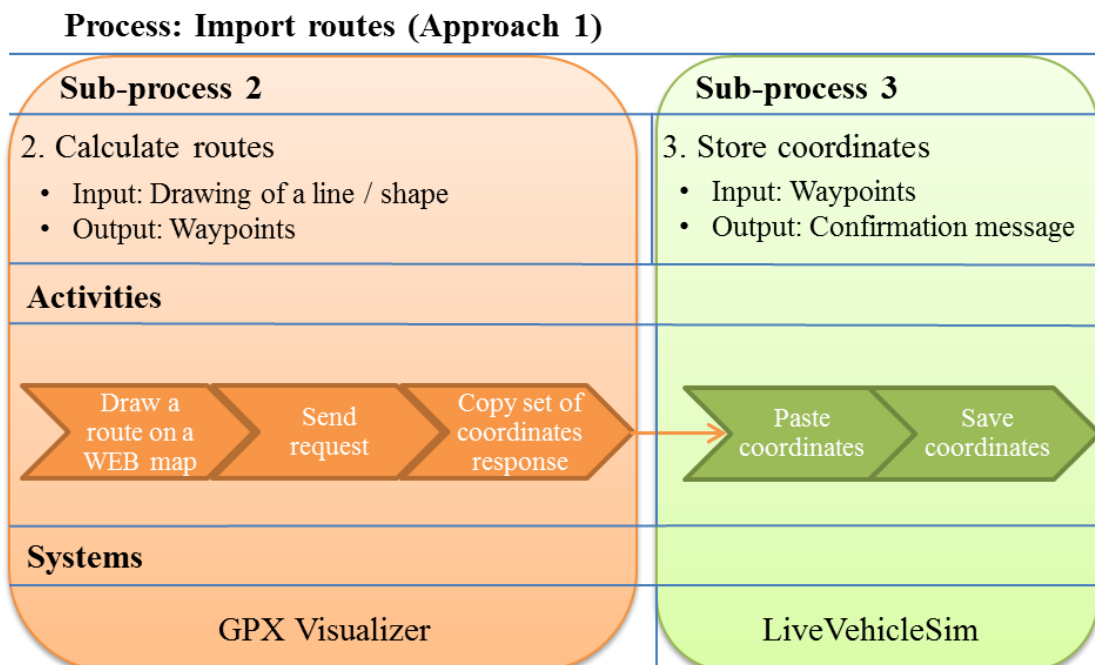


Figure 4.4 Approach 1: Import routes process removing *sub-process 1*.

Merging calculation and conversion of routes in one system is an advantage because a minimization of time is obtained by operating two systems instead of three. However, there is

still a drawback because manual tasks are needed to paste waypoints from one system to another. For this reason a second approach is presented to achieve a refined optimization, 4.2.2.2 Approach 2.

#### 4.2.2.2 Approach 2: merging calculation, conversion and storing

This approach focuses on integrating the calculation, conversion and generation of waypoints only into LVS using a Map API from WMS provider to make and manipulate data regarding routes; this means that Google Maps and GPX Visualizer are removed from the importing routes sub-process, as well as the activities associated with these two systems; in this way, all activities are executed in one single system.

Figure 4.5 describes the optimization of importing routes based on drawing a line as input of information using Web maps as first activity to define a route, this activity is performed in LVS. The final operation consists of sending a request to save the coordinates that were set in the previous activity, and then, an output is displayed as confirmation message on Web GUI to corroborate whether the operation was successful or not.

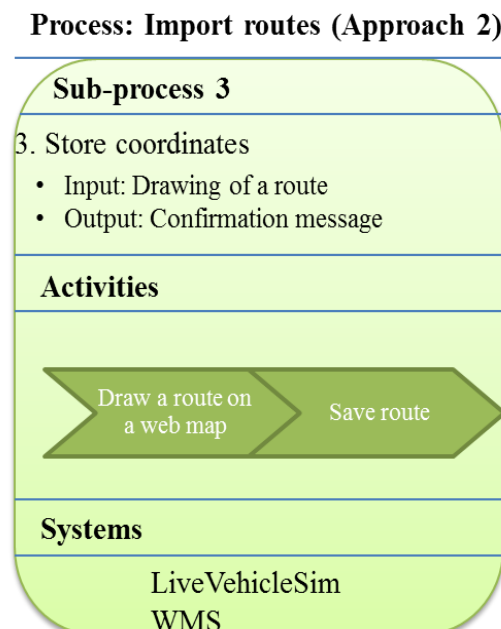


Figure 4.5 Approach2: Import routes process, removing *sub-process 1 & 2*

The advantage of this approach is that manual activities are removed and a considerable minimization of time is achieved when the execution of importing routes is performed; since only one system operates all the activities without the intervention of manual tasks. However, a further analysis and evaluation is needed to define the provider of the map API before this is integrated into LVS.

## Evaluation of different WMS providers

According to Schmidt and Weiser in *Online Maps with APIs and Web Services* [SW12] and Błażej [CJM+10], companies such Google, Microsoft, OpenStreetMap and Nokia are the most relevant in the market of WMS. This is due to the success of their services and the capacity to handle their services and support. Thus, these four main providers are part of the evaluation for integrating a map API with its WMS into LVS.

The evaluation of the WMS provider is based on the following criteria: *zero cost of investment* from the current business constraints defined in section 1.4, and *functionality, reliability usability* from the ISO 9126 (see section 2.3). These criteria are assessed considering the documentation from every provider, from which Appendix B contains the details regarding whether every criterion is satisfied or not.

Considering Table 4.9, those providers that hold the criteria are ticked, and those that do not, a blank space is left. After the evaluation *OpenStreetMap* obtained the lowest score, 16, because this is not reliable and implies costs of licences. Followed by *Google* and *Bing (Microsoft)* providers with a score of 24, they fulfil all criteria except cost of investment, since costs of licences are also associated to the integration of their map services. Finally, *Nokia Maps*, has a score of 34, this provider holds all criteria, including a zero cost of investment as FleetBoard's services include Nokia maps licences, which means that licensing is already paid. Therefore, Nokia maps service is used in this thesis. In addition, Appendix B contains justifications regarding this evaluation.

The selection is based on the provider with the largest *total* (see Table 4.9). This total contains the sum of all criteria that are ticked. The importance of every criterion is defined according to the FleetBoard's importance, from which 10 is the most important and 7 the least one.

Criteria	Weight	Company's Map service			
		Google	Bing	OpenStreetMap	Nokia
Zero cost of investment	10				✓
Functionality	9	✓	✓	✓	✓
Reliability	8	✓	✓		✓
Usability	7	✓	✓	✓	✓
	<b>Total</b>	24	24	16	34
	<b>Selection</b>				✓

Table 4.9 Evaluation and selection of the WMS Provider

Due to the results of Table 4.9, Nokia is the most suitable provider for a WMS and map API. This decision is based on the results from Table 4.9, from which Nokia reached 34 points out of 34 along different providers, despite all of them are supported by the same principle, which consists of using a central server called Map Tiles Server, a distributed database system and Web browser to make request using a map API (see section 2.2).



### 4.2.3 The selected approach

Considering the advantages exposed in the second approach, the implementation will be based on it. This approach includes integration of the calculation, the conversion and the storing of waypoints by means of LVS, Nokia WMS and Nokia's map API. As consequence manual tasks are eliminated and only one system controls the importing of routes process.

### 4.2.4 Specification of the selected approach

Supported by the decision of integrating Nokia Maps into LVS, functional requirements comprise: first, the configuration of the Nokia API to integrate the WMS into LVS. Second, modify the Web GUI to support the creation of routes using Nokia API. UML is used to describe these functional requirements through use cases notation. Considering the Figure 4.2, only the *View* is affected, because the business logic placed in the controller and the persistence operations in the model layer are the same. From Table 4.10 to Table 4.11 the description of the functional requirements is specified.

#### 4.2.4.1 Use cases for the approach 2

<b>Name</b>	Configure Nokia API
<b>Goal</b>	Integrate Nokia WMS into LVS
<b>Pre-Condition</b>	1. User and token provided by Nokia to access its map services must be active.
<b>Post-Condition</b>	1. Nokia's API available to receive request by LVS
<b>Post-condition in special case</b>	LVS displays the exception message
<b>Normal Case</b>	1. Nokia's API should behave as it is stipulated in the documentation.
<b>Special cases</b>	1. None function from Nokia's API is available.

Table 4.10 Use case to *configure Nokia API*

<b>Name</b>	Creation of routes
<b>Goal</b>	Using a map in the Web GUI, users select a route, and the waypoints from that route are imported, loaded and created in LVS.
<b>Pre-Condition</b>	1. User and token provided by Nokia are active 2. LVS' database is running. 3. WMS from Nokia is available
<b>Post-Condition</b>	1. A route is stored in LVS' database
<b>Post-Condition in special case</b>	LVS displays the exception message
<b>Normal Case</b>	1. A list of waypoints are retrieved using the Nokia API.
<b>Special cases</b>	1. No route is created in LVS

Table 4.11 Use case to create routes using Nokia API

### **4.3 Conclusion**

Two concepts are defined in this chapter, these follow the same methodology, analysis of the current problem, and the suggested solution including technical description and specification to support a suitable implementation of the prototype. The first concept defines a new process to import and create routes using FBSC's database, from which, the real routes are stored. The second concept includes an optimization of the process of importing routes from WMS provider. In addition, an evaluation is provided to select the best WMS provider based on criteria defined in the ISO 25000 and FleetBoard's restrictions (see section 1.4). As result of this evaluation Nokia WMS provider was selected.

---

## 5 Implementation

---

This chapter encompasses the development of the prototype considering the concept and design defined in Chapter 4. A glance over the current configuration environment and tools is defined in section 5.1, MVC design pattern (see section 3.4.3) is applied to develop the new functionalities. Section 5.2 contains the implementation performed in the Web GUI, and section 5.3 describes the business logic that support the prototype, followed by section 5.4 which comprises details about the persistence layer. Additionally, a class diagram in section 5.5 summarises the classes modified during the implementation.

### 5.1 Implementation overview

The new functionalities are developed in Java using Integrated Development Environment (IDE) Eclipse [Ec13]. For the Application Server and Servlet Container is Apache Tomcat [To13]. The RDBMS is MySQL [My13]. Maven is used as tool for Project Build Manager [Ma13] and SVN is used for version control on the file level [Su13]. Appendix C contains the detailed information regarding versions and libraries used during the implementation.

Considering the MVC pattern, the view is built it up using Java Server Faces(JSF), including a HTML tag library, for the user interface (UI) components, JSF core tag library to customize actions and RichFaces tag library for easily integrating Asynchronous JavaScript and XML (AJAX) features into the application. The model layer uses Apache Open JPA implementation for the Java Persistence API specification [Op13].

### 5.2 Web Graphical User Interface (GUI)

The implementation is performed over the route management and tour management views (see section 3.4) to satisfy the functional requirements defined in sections 4.1.3 and 4.2.4. Thus, this section comprises a description of the actual Web GUI, a new functionality using FBSC as source of real routes, and the integration of WMS into LVS.

#### 5.2.1 Description of the current Web GUI

The *route management view* is used by users for manipulating routes, thus CRUD (create, read, update and delete) operations that belong to routes are performed in this Web GUI. Every CRUD operation is composed of the following attributes: *name* to identify the route, *description* to detail a particular characteristic about the route, *origin* to specify where a route starts, *destination* defines the final place of a route, and the *GPX* data for the GPS coordinates which are used for setting waypoints of a route. After populating these fields in the Web GUI, a route is created and is ready to be used by tours (see section 3.4). The Figure 5.1 depicts this view.

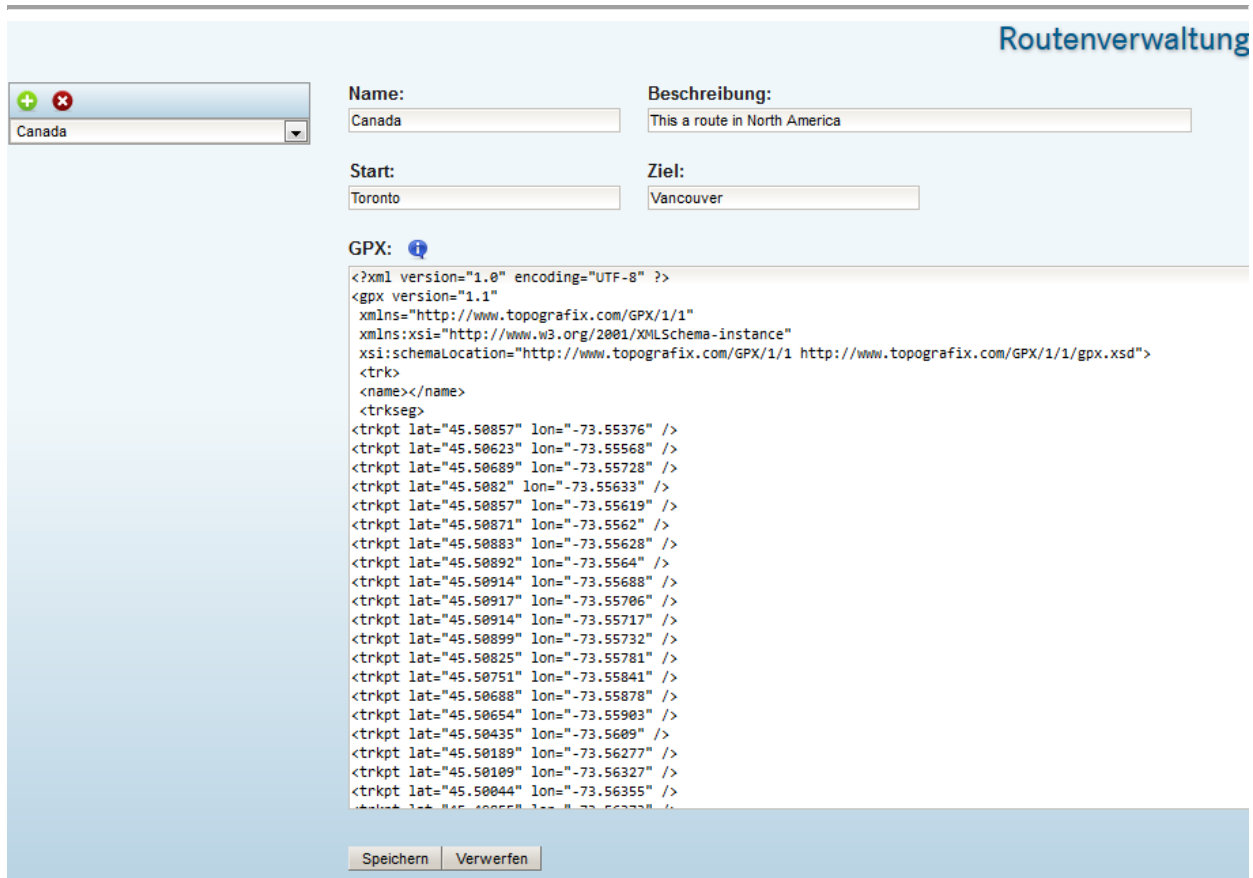


Figure 5.1 Current Web GUI

## 5.2.2 New CRUD operations using real routes

This section describes the new behaviour of CRUD operations in the *routes management* Web GUI using routes from FBSC. The description of every operation considers their objectives, data needed to execute them, results after executing them and their relationships to functional requirements defined in section 4.1.3. Additionally a screenshot (see Figure 5.2) illustrates how these operations look like in the Web GUI.

### 5.2.2.1 Creation of routes

The objective of this operation is to create real routes using the Web GUI. For every execution of this operation a set of routes is created in LVS, the size of the set depends on the number of vehicles that belongs to a fleet's selection in the Web GUI. For every execution users only see a confirmation message whether operation was successful or not, which is similar to the previous mechanism to create routes. However, they differ from input data and the algorithms.

Due to the automatic operation to create real routes the Web GUI only requires a *name* which identifies the set of routes, a checked *checkbox* to indicate FBSC as source of information, and a *fleet* from which vehicles are used to create routes, so that, for every vehicle a real route is created. This implementation covers the use case *creation of routes* (see Table 4.1).

After users send a request to create the real routes, only one route in the left panel is displayed, which is called parent route, because this represents the set of routes already created. Figure 5.2

illustrates the routes administration Web GUI with the new input data, and the results after create operation is executed.

### 5.2.2.2 Update of routes

Update operation consists of bringing up to date data regarding real routes. Thus, considering a selection of a parent route and any modification of its *name* and *fleet*, this operation removes the set of routes that belongs to the parent; afterwards, the same process of creating routes is applied. This operation is based on the definition of the use case *Update routes* (see Table 4.2).

### 5.2.2.3 Deletion of routes

The aim of this operation is to remove a set of routes that belongs to a parent route selection on the Web GUI (see Figure 5.2). After executing this operation a message is displayed on the Web GUI confirming whether the operation was successful or not. If operation is successful, the selected parent route from the left panel, is also removed, otherwise parent route remains in the Web GUI. This functionality is based on the use case *delete routes* in Table 4.3.

### 5.2.2.4 Retrieving routes

This operation retrieves routes from LVS' database every time the Web GUI is requested. The previous mechanism retrieved routes one by one. The new implementation keeps the previous mechanism, because this is used for routes that come from WMS provider (Google Maps). In addition, to retrieve routes that come from FBSC the new implementation displays only parent routes which represent sets of routes. Thus, the Web GUI displays all routes that are created using WMS provider, and parent routes (see Figure 5.2). This operation is supported by the use case *Display routes* in Table 4.4.

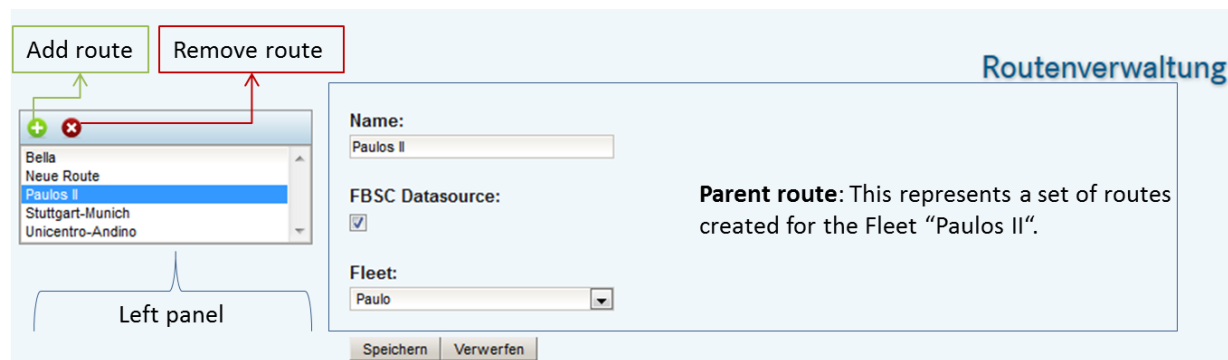


Figure 5.2 The implementation for importing routes from FBSC database

## 5.2.3 Integration of Nokia Maps into LVS

This section describes the new functionality that allows creating routes automatically from WMS provider. This functionality runs on the *routes administration* Web GUI. *Create* and *retrieve* operations are described, while *delete* and *update* operations are not treated, because they are not modified.

Technically this Web GUI integrates Nokia map API (see section 2.2.8) to interact with the Web GUI and WMS. Consequently, API's functions and a tailored algorithm are used to import and

load waypoints automatically. This functionality is based on the use case *configure Nokia API* (see Table 4.10) and *create routes using Nokia API* (see Table 4.11).

The screenshot displays a web interface for route management. At the top, there are input fields for 'Name' (containing 'puerto') and 'Beschreibung' (containing 'Description'). Below these are 'Start' (Stuttgart) and 'Ziel' (Munich) fields. A section labeled 'FBSC Datasource:' has an unchecked checkbox. A map of Europe shows a blue route from Stuttgart to Munich, with blue circles indicating waypoints. Below the map, there is a 'GPX:' section with a checked checkbox 'Füllen mit Nokia' and a text box stating 'This removes automatically the current waypoints, and brings values from Nokia'. At the bottom, an XML code block shows the resulting route data.

```
<?xml version="1.0" encoding="UTF-8" ?>
<gpx version="1.1"
  xmlns="http://www.topografix.com/GPX/1/1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/1/1/gpx.xsd">
  <trk>
    <name></name>
    <trkseg>
      <trkpt lat="48.7766609" lon="9.1789703"/>
      <trkpt lat="48.7765694" lon="9.1791281"/>
    </trkseg>
  </trk>
</gpx>
```

Figure 5.3 Interaction of Nokia WMS and the routes administration Web GUI

### 5.2.3.1 Description of the create operation

Two different perspectives are considered for describing routes creation. The first perspective is regarding the interaction between users and the Web GUI, and the second encompasses the interaction among Nokia's API, a JavaScript tailored algorithm and Nokia's WMS. These two perspectives are presented together, because they are part of the presentation layer, so that, they are executed in customers' side, instead of the server side.

Using Nokia JavaScript API users interact with a Web map in the Web GUI by setting the initial and final points of a route, thereafter an asynchronous request is sent it to Nokia's servers, and the response is formatted to a route which is display in the Web GUI (see Figure 5.3). For further details about mapping communication architecture refer to 2.2.9.

- a) *Interaction between users and Web GUI.* The objective of this Web GUI is to provide a means of importing and loading waypoints from Nokia Maps automatically. Thus, the new functionality considers the input fields from the previous Web GUI (see section 5.2.1), and adds one boolean checkbox to indicate Nokia Maps as source of information. Figure 5.3, illustrates how the Web GUI looks like when the checkbox (*Fullen mit Nokia*) is checked, and a route is imported and loaded into Web GUI.

---

Following Figure 5.3 the creation of routes using Nokia's WMS consists of four steps. *Step 1:* Users fill out the route description, which comprises *name*, *description*, *initial* and *final* place of a route. *Step 2:* Users check the checkbox (Füllen mit Nokia) that indicates Nokia as source of information; this selection displays in the Web GUI a Web map with a predefined route, which is modifiable. *Step 3:* If users set another route, the *GPX* text is loaded with waypoints coming from Nokia's servers, otherwise the waypoints from the predefined route are kept. *Step 4:* Users create the route in LVS. The implementation of this thesis covers steps 2, 3 and 4.

- b) *Interaction among Nokia's API, a tailored algorithm and Nokia's WMS.* Since the tailored algorithm not only contains functions to format and load waypoints, but also calls to functions of Nokia's API, a description of the algorithm is provided based on Listing 5.1.

The algorithm, written in JavaScript lets interact users and the Web map, and also the Web map with Nokia's WMS. Listing 5.1 describes the main part of the algorithm, from which the preconfigured route is created. Thus, if users active the checkbox to fulfil the input field, then, the local array *latLonArrayRoute* is loaded of waypoints that are contained in the *shape* object from Nokia WMS. This object contains all possible waypoints that define a route, and its parameters constrain the size of the array to return. Consequently, *coordinates*, which is a variable, creates a string of characters based on the *GPX* schema (see Figure 2.3) to manipulate geographical data, this operation lasts depending on the number of waypoints stored in the *shape* object. Finally, the route is set to *gpxString*, which is a text field in the Web GUI, and also the map is displayed.

```

// Create the new route polyline
route = obj.routes[0];

//Evaluates if checkbox is checked
if (document.getElementById("f:importFromNokia").checked) {
    var coordinates="";

    //This method retrieves latitudes and longitudes from strip
    elements, starting at the caller-specified index
    var latLonArrayRoute=route.shape.getLatLng
    (0,route.shape.getLength());

    //Setting variable with format <trkpt lat="X" lon="Y"/>
    for (var i = 0; i < latLonArrayRoute.length; i++){
        coordinates+="<trkpt lat=\""+latLonArrayRoute[i]+"\" lon=
        \""+latLonArrayRoute[++i]+"\"/>\n";
    }

    //Setting input text field in the WEB GUI
    var gpxString= "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n"
    + "<gpx version=\"1.1\" \n"+ " xmlns=
    \"http://www.topografix.com/GPX/1/1\" \n" + " xmlns:xsi=
    \"http://www.w3.org/2001/XMLSchema-instance\" \n"
    + " xsi:schemaLocation=\"http://www.topografix.com/GPX/1/1
    http://www.topografix.com/GPX/1/1/gpx.xsd\">\n"
    + " <trk>\n" + " <name></name>\n" + " <trkseg>
    \n"+coordinates+"</trkseg></trk></gpx>";

    document.getElementById("f:gpxString").value=gpxString;
    alert("Waypoints imported: "+latLonArrayRoute.length);
}
//Display the route in the map
routePolyline = new nokia.maps.map.Polyline(route.shape, {
    pen: {
        lineWidth: 3,
        lineJoin: 'round'
    }
});

```

Listing 5.1 The main part of the algorithm to retrieve waypoints from Nokia's servers

### 5.2.3.2 Description of the retrieve operation

The objective of this operation is to load a route on the Web map that is displayed on the Web GUI. The retrieve operation is executed when users select a route that contains Nokia Maps as source of information, subsequently initial and end waypoints are taken out of the route and sent to Nokia's API to display a route on a Web map.

## 5.3 The business logic

This section encompasses a description of algorithms that communicate the *routes administration* Web GUI to the database using the *data model* layer (see section 3.4.3.1), and also algorithms that are used for calculations and conversions. These algorithms are described depending on the CRUD operation that they belong to.



---

### 5.3.1 Creation of routes

During the *creation* of a route the Web GUI triggers the *saveFBSCRoute* method (see Listing 5.2), subsequently this method invokes the methods *importFBSCRoutes*(see section Listing 5.3) and *getGPSPositions* (see section Listing 5.4). These three methods belong to the class *RouteController* which sends routes to the *model layer* to store these routes in MySQL. In addition, these methods constitute part of the implementation of this thesis, and they are described in details.

- a) **saveFBSCRoute:** The objective of the *saveFBSCRoute* method is to save a route into the database as long as the *model layer* offers an active transaction to store routes. The process store routes consists of (1) *validating if an identical routes is already created*, if this condition holds, the previous route is overwritten, in that way, up to date data are kept in LVS. The next step (2) *brings vehicles* that belong to the fleet selected in the Web GUI. *Step 3*, every vehicle is iterated to import its route from FBSC invoking the method *importFBSCRoutes*. *Step 4*: if a route is found it, this route is set to the GPS positions for keeping a valid format, and then the route is stored in MySQL. This method is shown in Listing 5.2.

```

public void saveFBSCRoute(Route originalRoute, Route currentRoute) {
    //-----First step (1)-----
    //Delete current routes with the same name
    if ((this.model.getDaoFactory().getRouteDAO().countActiveRoutes
        (currentRoute.getName()) > 0) || (currentRoute.getId() != null)) {
        this.delete(originalRoute);
    }

    //-----Second step (2)-----
    //Bringing vehicles that belong to a specific fleet
    List<Vehicle> vehicles = vehicleController.getVehicles
        currentRoute.getFleet();

    for (Vehicle vehicle : vehicles) {

        //-----Third step (3)-----
        //Bringing GPS positions using the importFbscRoutes method.
        Route route = new Route(currentRoute, vehicle);
        List<GpsPosition> positions = importFbscRoutes(vehicle,route);

        //-----Third step (4)-----
        //Adding waypoints to the current route and storing
        if (positions != null) {
            route.setGpsPositions((ArrayList<GpsPosition>)positions);

            model.beginTransaction();
            model.save(route);
            model.commitTransaction();
        } else {
            //In case that start or event records are invalid
            LOGGER.info("Start/Event record not valid for vehicle(FBID):" +
                vehicle.getFbId()
                    + "LVS ID: " + vehicle.getId());
        }
    }
}

```

Listing 5.2 Save FBSC routes algorithm

b) **importFBSCRoutes**: Considering the use case *Import routes*, Table 4.5, this method (see Listing 5.3) returns GPS positions based on the vehicle received as parameter. The steps to accomplish this process comprise:

1. The *creation of vehicle and fleet* objects to access FBSC's database.
2. *Definition of a time interval*. Thus, considering a vehicle, the latest date of its stop defines the end of the interval, and 10 hours back, maximum hours allowed to drive [Ro06], set the initial date of the interval. The algorithm presented in Listing 5.3 bounds the time interval with the variables *driveStartDate* and *driveEndDate*, thus, waypoints within this interval are imported. In addition, the conditional statements *if* validate whether references are not null and the latest date of stop is greater than the date of starting to drive, if these conditions are not satisfied, routes are considered invalid.

3. *Invocation of getGpsPosition method.* The objective is to retrieve a set of waypoints from FBSC's database.

```

private List<GpsPosition> importFbscRoutes(Vehicle vehicle, Route route) {

    //-----Step 1-----
    //Instantiation of fleet and vehicles objects to import coordinates
    com.fleetboard.dto.Fleet fbscFleet =
    fbServerdataAccess.getFleetRepository().findOne(vehicle.getFleet
    ().getFbId());
    com.fleetboard.dto.Vehicle fbscVehicle =
    fbServerdataAccess.getVehicleRepository().findByChassisAndCurrentFleet
    (vehicle.getChassis(), fbscFleet);

    //-----Step 2-----
    //Setting the initial and final datetime to define a time interval.
    Waypoints that are within this interval are retrieved.

    //2.1 Bringing the latest stop event timestamp of a vehicle
    Timestamp driveEndDate =
    fbServerdataAccess.getTourEventRecordDataRepository().
    getLastTimeStamEvent Type (fbscVehicle, TourEventTypes.DriveEnd.getValue
    ());

    int seconds = 36000; //10 hours

    if (driveEndDate != null) {
        //2.2 Bringing the first start event timestamp considering a vehicle
        and the date of the latest stop
        Timestamp driveStartDate=
            fbServerdataAccess.getTourEventRecordDataRepository()
            .getFirstTimeStamEvent Type Greater Than Utcvehicle(fbscVehicle,
            TourEventTypes.DriveStart.getValue(),
            new Timestamp(driveEndDate.getTime() - (seconds * 1000L)));
    }
    //-----Step 3-----
    //Returning a set of waypoints that belong to a vehicle
    if ((driveStartDate != null) && (driveStartDate.getTime() <
        driveEndDate.getTime())) {

        return (getGpsPositions(fbscVehicle, driveStartDate, driveEndDate,
            route));

    }

}
return null;
}

```

Listing 5.3 Algorithm to import FBSC routes

- c) **getGPSPositions:** The task of this method, Listing 5.4, is to return an array of GPS positions. First, this method retrieve waypoints from *gpstracedata* table from FBSC's database, in case that waypoints are not found, a second request is sent to *gpsdata* table. However, if waypoints are not found, the value returned is null which means that a route is not created.

In details, this algorithm, Listing 5.4, imports waypoints from *gpstracedata*, and a splitting operation is performed using nested cycles, because sets of waypoints are stored in FBSC's database as traces (2.1) in one entry. Thus, the objective of splitting is to obtain individual waypoints. In addition, the execution of the splitting operation depends on the number of entries retrieved from the *gpstracedata*. Figure 5.4 describes this method.

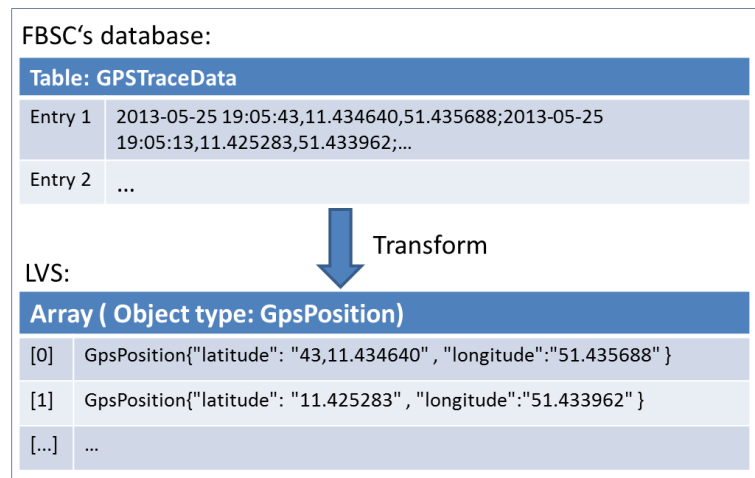


Figure 5.4 Example of the splitting operation

Different process applies when *gpsdata* table is used. As FBSC's database stores every waypoint as one entry. Thus, no splitting operation exists. In this case a set of waypoints is retrieve from FBSC's database, then this set is store in a list which is returned to *importFBSCRoutes* method.

```

private List<GpsPosition> getGpsPositions(com.fleetboard.dto.Vehicle
fbscVehicle, Timestamp driveStartDate, Timestamp driveEndDate, Route route) {

    List<com.fleetboard.dto.Gpstracedata> gpsTraceData = null;
    gpsTraceData = getFbServerdataAccess().getGpstracedataRepository
        ().findByVehicleByStamps(fbscVehicle, driveStartDate,
            DriveEndDate);

    ArrayList<GpsPosition> positions = new ArrayList<GpsPosition>();

    //-----Importing waypoints from GPSTRACE table-----
    if ((gpsTraceData != null) && (gpsTraceData.size() > 0)) {
        for (com.fleetboard.dto.Gpstracedata traces :gpsTraceData)
        {
            List<String> positionInformation = Arrays.asList
                (traces.getPositioninformation().split(";"));

            for (String entries : positionInformation) {
                String[] deltaPosition = entries.split(",");
                positions.add(new GpsPosition
                    (Double.parseDouble(deltaPosition[1]), Double
                        .parseDouble(deltaPosition[2]), route));
            }
        }
        return (positions);
    }
    //-----Importing waypoints from GPS table-----
    else {

        List<com.fleetboard.dto.Gpsdata> gpsData = null;
        gpsData = getFbServerdataAccess().getGpsdataRepository
            ().findByVehicleSinceStamps(fbscVehicle,
                driveStartDate, driveEndDate);
        if ((gpsData != null) && (gpsData.size() > 0)) {
            for (Gpsdata entries : gpsData) {
                positions.add(new GpsPosition
                    (entries.getLongitude().doubleValue(),
                        entries.getLatitude().doubleValue(), route));
            }
            return (positions);
        } else {
            return null;
        }
    }
}

```

Listing 5.4 Algorithm to obtain the right waypoints

### 5.3.1.1 New Interfaces to access FBSC' database

The objective of this section is to describe new interfaces for accessing FBSC's database. Although, interfaces to access this database already exist, they do not fit the purpose of retrieving waypoints considering a time interval. Thus, the interfaces implemented are *GpstracedataRepository* (see Listing 5.5) and *GpsdataRepository* (see Listing 5.6).

FBSC contains a test API to access its database; every interface of this API supports the access to every table using JPQL statements. In this case, *gpstracedatarepository* is associated to *gpstracedata* table, and *gpsdatarepository* to *gpsdata* table. Listing 5.4 contains invocations to these interfaces.

The process of implementing these statements consists of building it up locally, then they are placed in the Maven repository (see section 5.1), which evaluates whether statements are valid or not. In case they are valid, the new interfaces are updated to the test API that is in the production environment.

Listing 5.5 represents *GpstracedataRepository* to obtain waypoints from *Gpstracedata*. The set of the waypoints are bounded to a vehicle and the time interval defined in method *getGpsPositions*. This interface returns a list of *Gpstracedata* objects

```
package com.fleetboard.repository;

public interface GpstracedataRepository extends JpaRepository
<Gpstracedata, Long> {

    @Query("select g from Gpstracedata g where g.vehicle= :vehicle and
g.mingpstime >= :startdate and g.maxgpstime<=:enddate order by
g.vehicletimestamp")
    List<Gpstracedata> findByVehicleByStamps(@Param("vehicle") Vehicle
vehicle, @Param("startdate") Timestamp startdate, @Param("enddate")
Timestamp enddate);
}
```

Listing 5.5 Interface to access FBSC gpstracedata table.

Similarly, Listing 5.6 represents *GpsdataRepository* to access the *gpsdata* table. This interface returns a list of *gpsdata* objects. The statement is limited to : *vehicle*, *start* and *end* dates, which are sent by *getGpsPositions* method.

```

package com.fleetboard.repository;

public interface GpsdataRepository extends JpaRepository<Gpsdata, Long>
{
    @Query("select g from Gpsdata g where g.vehicle= :vehicle and
g.packetid=254 and g.utcvehicle between :startdate and :enddate order
by g.utcvehicle")
    List<Gpsdata> findByVehicleSinceStamps(@Param("vehicle") Vehicle
vehicle, @Param("startdate") Timestamp startdate, @Param("enddate")
Timestamp enddate);
}

```

Listing 5.6 Interface to access FBSC gpsdata table

### 5.3.2 Retrieving routes

This operation gathers routes from FBSC and Nokia Maps. *Routes administration* Web GUI invokes the method *getActiveRoutes* to obtain a list of active routes. In case routes were imported from FBSC, the model layer only returns parent routes, which represent sets of routes associated to a vehicle. The data model section (5.4) describes in details how the data are retrieved and grouped from the database.

### 5.3.3 Updating routes

Update operation uses the *saveFBSCRoute* method (see Listing 5.2), from which a set of routes are overwritten to obtain up to date data, the details of the algorithm are discussed in section 5.3.1.

### 5.3.4 Delete routes

Listing 5.7 describes the deletion of a route; this method uses routes as parameter to request actives routes from LVS' database. Routes' names act as filter to retrieve a set of active route. Consequently, the model layer returns a list of routes, which is iterated to set the *delete* field to true. *If* statement is utilized to avoid null values, if the value is null, a message is displayed to report that active routes were not found (see specifications 4.1.3).

```

public void delete(Route route) {
    List<Route> activeFbscRoutes = model.getDaoFactory().getRouteDAO
    ().findActiveRoutes(route.getName());

    if (activeFbscRoutes != null) {
        for (Route r : activeFbscRoutes) {
            model.beginTransaction();
            r.setDeleted(true);
            model.commitTransaction();
        }
    }
    else {
        LOGGER.info("There are not active Routes with the name: " +
            route.getName());
    }
}
}

```

Listing 5.7 Algorithm to delete routes

## 5.4 The data model

This section describes the interaction between the software application and the database, hosted by MySQL. The new methods in the interface *RouteDao* are described according to their objectives and relationships to the business logic layer (see section 5.3). These methods are shown in Listing 5.8 and their implementations in Listing 5.9.

The new Methods are:

- a) *getFbscActiveParentRoutes*: The objective of this method is to retrieve sets of parent routes from the database. Parent routes are built by grouping routes with the same name. This method returns a list of routes to *getActiveRoutes* method (see section 5.3.2).
- b) *getAllActiveExternalRoutes*: This method retrieves routes that were created using WMS provider. This method returns a list of routes to *getExternalActiveRoutes* method.
- c) *countActiveRoutes*: This method counts the number of parent routes. This method returns a variable to the *getExternalActiveRoutes* method.
- d) *findActiveRoutes*: This method finds active routes based on a route name parameter. This methods returns a list of routes to the *delete* method.
- e) *findVehicleRoute* returns a route that belongs to a vehicle received as parameter. This method is invoked by *getActiveRoutes* method.

```

public interface RouteDAO extends GenericDAO<Route> {
    List<Route> getFbscActiveParentRoutes();
    List<Route> getAllActiveExternalRoutes();
    long countActiveRoutes(String routeName);
    List<Route> findActiveRoutes(String routeName);
    List<Route> findVehicleRoute(Vehicle vehicle);
}

```

Listing 5.8 Interface to access data that belong to routes.



```

public class RouteDAOImpl extends GenericDAOImpl<Route> implements RouteDAO {

    private static final long serialVersionUID = -7083084841378184392L;
    protected RouteDAOImpl(EntityManager em) {
        super(em);}

    @Override
    public List<Route> getFbscActiveParentRoutes() {
        List<Route> routes = this.em.createQuery(
            "SELECT r FROM Route r where r.fbscdatasource=true
            and r.deleted=false group by r.name")
            .getResultList();
        return routes;
    }

    @Override
    public List<Route> getAllActiveExternalRoutes() {
        List<Route> route = this.em.createQuery(
            "SELECT r FROM Route r where r.fbscdatasource=false
            and r.deleted=false").getResultList();
        return route;
    }

    @SuppressWarnings("unchecked")
    @Override
    public List<Route> findActiveRoutes(String routeName) {
        List<Route> routes = this.em.createQuery("SELECT r FROM Route r
            where r.name=:routeName and r.deleted=false")
            .setParameter("routeName", routeName).getResultList();
        return routes;
    }

    @SuppressWarnings("unchecked")
    @Override
    public long countActiveRoutes(String routeName) {
        long counter = (Long) this.em
            .createQuery("SELECT count(r) FROM Route r where
            r.name=:routeName and r.deleted=false")
            .setParameter("routeName", routeName).getSingleResult();
        return counter;
    }

    @SuppressWarnings("unchecked")
    @Override
    public List<Route> findVehicleRoute(Vehicle vehicle) {
        List<Route> route = this.em.createQuery("select r FROM Route r
            where r.vehicle=:vehicle and r.deleted=false")
            .setParameter("vehicle", vehicle).setMaxResults(
            1).getResultList();
        return route;
    }
}

```

Listing 5.9 Implementation of methods to access data that belong to routes

## 5.5 Class diagram

The following class diagram represents the classes that were modified during the development of the prototype, including their relationships between model, view and controller layers. The modified attributes and methods are specified in sections 5.2, 5.3 and 5.4. Figure 5.5 depicts the class diagram.

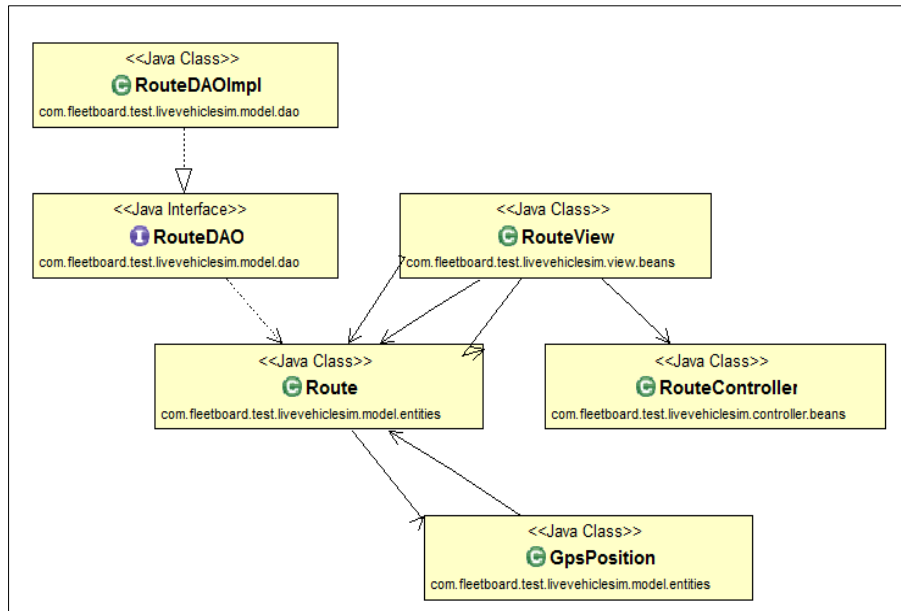


Figure 5.5 Class diagram

---

## 6 Test and validation

---

In this chapter the developed prototype is evaluated using testing methodologies and tools. Tests' results are validated based on: (1) functional requirements defined in sections 4.1.3 and 4.2.4 (2) Non-functional requirements according to the prototype's performance, and (3) Data quality of the prototype which is compared with the previous implementation. Subsequently, Chapter 7 evaluates the results obtained in this chapter.

Section 6.1 comprises a brief introduction to JMeter testing tool [Ap13] with its configuration elements. Section 6.2 evaluates functional requirements based on the output data of the prototype. Non-functional requirements are evaluated applying the performance analysis, section 6.2 that includes metrics and thresholds to ensure an adequate quality of service of the prototype. *Measurement methodology* is applied to assess the quality of data, see section 6.4.

Tests provided in this chapter are configured to interact with LVS' Web GUI, however the evaluation of these tests implies participation of the *model view and controller* layers, since they are related each other.

### 6.1 Configuration of tests using JMeter

The implemented prototype (see section 5) is tested using JMeter that tests functional behaviour and measures the performance [WW10] to corroborate the required functionality exposed in section 4.1.3. The configuration of this tool consists of a test plan for the test object describing the individual steps. Additionally, a thread group is created to visualize HTTP requests. The results after executing the test plan are stored in a result table (see Table 6.1) that contains several attributes to describe the user's request. In this case, the most relevant attribute is *status*, which determines if an effective request was sent to the server.

In addition, a HTTP proxy server configuration is created, because JMeter acts as intermediary for requests and responses. The setup includes name and port parameters to guarantee the availability of JMeter during the testing. To visualize the responses from the server *View Results Tree* reporter is added to the proxy server. Details has been explained in Appendix D and JMeter documentation [Jm13].

### 6.2 Test and validation of functional requirements

In this section JMeter is configured with scripts to perform CRUD operations in the administration of routes Web GUI (see section 3.4) while every operation is executed once and its response is evaluated. Additionally to the scripts, black box testing strategy is applied to assess the expected functionality of the web pages [DF05].

#### 6.2.1 Test scripts

Tests scripts are comprised of HTTP requests. Events and data required in the Web GUI are part of these requests. For example, to insert a route using waypoints from FBSC's database, the

required data are name, description and the fleet name, then a request event is executed to perform the operation in the WAS. The expected response in JMeter is *message response:OK*, which confirms that request and response were executed successfully.

After running tests, HTTP responses from the WAS are received by the proxy server to be analysed and stored in the configured reporters. Figure 6.1 shows in the right panel, the HTTP responses including the Web GUI that belongs to this, from which a first block of information states the metadata of the message, the following block contains the response headers, and the last block refers to HTTP fields. Thus, by reading every block of information the result of the test concludes that the CRUD operations were performed without inconveniences.

The left panel from Figure 6.1 contains the *test plan*, *reporters* and the *proxy server* configuration, which are the necessary elements to perform the complete test.

Figure 6.1 HTTP responses after running a pre-configured test case

## 6.2.2 Tests results

Results after running the test plan in JMeter are described in Table 6.1. The attributes of this table are: *thread name* which involves the request executed during the test plan, *operation* describes the operation to perform, and *label* is the relative path to reach the Web GUI, and *status* states whether the test is successful or not. Appendix E contains detailed information about these tests.

Tests regarding the importation of routes and CRUD operations that involve routes are running properly thus, they meet the functional requirements defined in section 4.1.3. Table 6.1

corroborate the success of tests by observing thread groups 1-2 *retrieving*, 2-2 *creation*, 3-2 *update* and 4-1 *delete*.

Successful tests are obtained after the integration of Nokia WMS into LVS, which is based on the concept defined in section 4.2 and its implementation described in section 5.2.3. Tests focus on sending requests to Nokia Maps server by means of the Nokia maps API, and the responses that contain huge sets of waypoints that define routes. Functional requirements defined in Table 4.10 and Table 4.11 are met according to the success of thread group 1-1, 2-1, and 3-1 from Table 6.1.

Thread Name	Operation	Label	Status
Group 1-1	Retrieve routes using Nokia WMS	/LiveVehicleSim/routes.htm	Success
Group 1-1	Retrieve routes using Nokia WMS	/routing/6.2/calculateroute.js	Success
Group 1-2	Retrieve routes using FBSC' database	/LiveVehicleSim/routes.htm	Success
Group 2-1	Creation of routes using Nokia's waypoints	/LiveVehicleSim/routes.htm	Success
Group 2-2	Creation of routes using FBSC's waypoints	/LiveVehicleSim/routes.htm	Success
Group 3-1	Update routes using Nokia WMS	/LiveVehicleSim/routes.htm	Success
Group 3-2	Update routes using FBSC's waypoints	/LiveVehicleSim/routes.htm	Success
Group 4-1	Delete operations using FBSC's waypoints	/LiveVehicleSim/routes.htm	Success

Table 6.1 Results of the CRUD operations, using HTTP request

Considering the successful results from requests and responses of the performed tests, the functional requirements from 4.1.3 and 4.2.4 are fulfilled. However, however a black box testing strategy is applied to these tests to strengthen the results from JMeter and to guarantee that the new developed functionalities behave as expected.

### 6.2.2.1 Black box testing strategy

The objective of applying this strategy is to evaluate the expected functionality of a Web page based on the analysis of its behaviour, which is addressed by decision tables technique [DF05]. This technique consists of *conditions* that represent the input values with events, and *actions* that comprise expected results, output values with events. Table 6.2 describes the template applied for this technique.

*Column conditions* from Table 6.2 represents any input value in the Web GUI, this column comprises four attributes: *Operation* which is the name of the functionality to test; *Action* that refers to the event that fires the test; *Variable* represents a field in the Web GUI; *Input value* which is the value associated to the variable. The second column describes the output after running the test. It contains four attributes: *Expected action* describes the expected action

performed by the server after running the test; *Action after testing* defines the action performed by the server after running the test; *Expected values* refers to the required values from the theory; *Values after testing* describes the obtained values after running the test. The last column refers to the decision which comprises a boolean result after evaluating the conditions and actions of every test case, e.g., result equals true means that the functionality passed the test, because expected values and obtained values are equal.

Input section (Conditions)				Output section (Actions)				Decision
<i>Operation</i>	<i>Action</i>	<i>Variable</i>	<i>Input value</i>	<i>Expected action</i>	<i>Action after testing</i>	<i>Expected values</i>	<i>Values after testing</i>	<i>Result</i>
...	...	...	...	...	...	...	...	...

Table 6.2 Template to evaluate behaviour of the Web application using decision tables technique

Considering the functional requirements defined in section 4.1.3 and 4.2.2, tests were run. The results obtained with data from customers represent the proper running of the prototype based on the specification. Every test consists of the data needed by the route administration module to perform any CRUD operation. The considered events are submission of the application and interactions with the Web map. For example, by clicking on the map, results are resumed in the *decision* column, which success if the expected behaviour of every test is met. Delete operation using Nokia Maps is not evaluated, since the logic uses the old mechanism, which has not changed.

Input section				Output section				Decision
Operation	Variables	Input value	Action	Expected values	Expected action	Values after testing	Action after testing	Result
Retrieve routes using Nokia WMS	Fill with nokia Maps (Checkbox)	Checked	Click on the map	Set of waypoints	GPX field updated with waypoints	List with 6985 waypoints	GPX field with a set of waypoints	TRUE
	Map	Initial and end point		Route in a map	Update map	Route in a map	Update map tile	
Retrieve routes using FBSC	Name (text field)	n.a	Load page	Buba (FBSC)	Retrieve data from DB and load values in the WEB GUI	Buba (FBSC)	Retrieve data from DB and load values in the WEB GUI	TRUE
	FBSC datasource ( Checkbox)	n.a		True ( checked)		True ( checked)		
	Fleet ( Selection list)	n.a		Buba		Buba		
Creation of routes using Nokia's waypoints	Name (text field)	Stuttgart->Munich	Submit form	Stuttgart->Munich	Creation of routes in the DB	Stuttgart->Munich	Retrieved and loaded values	TRUE
	Description (text field)	Route (Nokia)		Route (Nokia)		Route (Nokia)		
	Start (text field)	Stuttgart		Stuttgart		Stuttgart		
	End (text field)	Munich		Munich		Munich		
	Fill with nokia Maps (Checkbox)	Checked		Checked		Checked		
FBSC datasource ( Checkbox)	Not checked	Not checked	Not checked					
Creation of routes using FBSC's waypoints	Name (text field)	Buba FBSC	Submit form	Buba (FBSC)	Creation of routes in the DB	Buba (FBSC)	Retrieved and loaded values	TRUE
	FBSC datasource ( Checkbox)	Checked		True ( checked)		True ( checked)		
	Fleet ( Selection list)	Buba		Buba		Buba		
Update routes using Nokia WMS	Name (text field)	Stuttgart->Munich 2	Submit form	Stuttgart->Munich 2	Updated in the DB	Stuttgart->Munich 2	Retrieved and loaded values	TRUE
	Description (text field)	Route (Nokia) 2		Route (Nokia) 2		Route (Nokia) 2		
	Start (text field)	Stuttgart 2		Stuttgart 2		Stuttgart 2		
	End (text field)	Munich 2		Munich 2		Munich 2		
	Fill with nokia Maps (Checkbox)	Checked		Checked		Checked		
FBSC datasource ( Checkbox)	Not checked	Not checked	Not checked					
Update routes using FBSC's waypoints	Name (text field)	Buba FBSC 2	Submit form	Buba (FBSC) 2	Update of routes in the DB	Buba (FBSC) 2	Retrieved and loaded values	TRUE
	FBSC datasource ( Checkbox)	Checked		True ( checked)		True ( checked)		
	Fleet ( Selection list)	Buba		Buba		Buba		
Delete operations using FBSC's waypoints	Name (text field)	Buba FBSC 2	Submit form	No values	Removal of route	No values	Route removed from database	TRUE
	FBSC datasource ( Checkbox)	Checked		No values		No values		
	Fleet ( Selection list)	Buba		No values		No values		

Table 6.3 Description of test cases using decision table technique

## 6.3 Test and validation of non-functional tests

This section provides the evaluation of non-functional requirements to guarantee an adequate performance of the prototype in terms of responsiveness and stability. *Response time analysis time* (see section 6.3.2) is applied to validate the responsiveness, and the *throughput* to validate the stability (see section 6.3.3). A methodology (see section 6.3.1) is defined to evaluate the performance using JMeter.

### 6.3.1 Methodology

JMeter is used in order to evaluate the performance of the implementation. The performance metrics that are considered comprise *throughput* to measure requests per second to the server [FLZ10], and *the average response time*, which is the mean value of the elapsed time between a request to the server and the receipt of the response in the browser [FLZ10]. These metrics are useful, because the acceptability of a test is obtained from them. Consequently, if test is accepted, responsiveness and stability of the system based on the new functionalities are guaranteed.

The performance of an application comprises two criterion for acceptance, the *response time*, which is user concern, and the *throughput* as business concern [BBC+07]. In that way, the *response time* is evaluated using the *confidence interval analysis* with thresholds determined by human behaviour (see Table 6.6), and the *throughput* is directly compared to business constraints.

### 6.3.2 Response time analysis

This section evaluates the *response time* of CRUD operations. JMeter provides in its results *average response time* and *standard deviation*. Thus, after running JMeter's tests, these results are compared to thresholds that validate whether the prototype meet the minimal standards of responsiveness.

Because *average response time* is sometimes a misleading measure, an additional evaluation is performed using *confidence interval analysis* method to obtain a measure close to reality. This case is presented when response time values are far from the *average response*. An example is provided in Table 6.4, from which *Test 1* and *Test 2* show *average response time* and *standard deviation* that result after running every test. Clearly, *Test 1* provides a misleading average, because every sample data are far from the average, e.g.,  $x1=16s$  is far from 5s, which is called *variability*. On the other hand, *Test 2* provides an ideal scenario, since sample data is close to the mean value, e.g.,  $x4= 5s$  is close to 5s.



	Test 1 (Seconds)	Test2 (Seconds)
	x1= 16s	x1= 5s
	x2= 2s	x2= 4s
	x3= 1s	x3= 6s
	x4= 1s	x4= 5s
<b>Average resp. time</b>	5s	5s
<b>Standard deviation</b>	6,36	0,70

Table 6.4 Real and misleading average response times

*Confidence interval analysis* solves the problem of misleading *average response time*. This analysis estimates the *variability* of the sample data, and provides an *average response time* close to reality based on a predefined probability or interval. In addition, the *standard deviation* is used for the calculation of the *confidence interval*, and also for determining whether the *average response* is misleading, e.g., if *Standard deviation is low than the average response time, then average is accurate, otherwise misleading* [Jo04]. This thesis will adapt the response time analysis for all evaluations to obtain more accurate results, even if the standard deviation is lower than the average response.

The *response time analysis* is divided in three sections: (1) *Definition of the confidence interval* that provides details about the mode of estimating the confidence interval and conditions. (2) *The well-known thresholds for response times* that Web applications should consider. (3) *Validation of results*, from which tests results are validated by comparing them to the thresholds. These three sections are briefly described in the following paragraphs:

(1) *Definition of the confidence interval*

The acceptability of the *response time* metric is determined using the *confidence interval analysis* method, which is based on the *Central Limit Theorem*, that states: if a representative part from a group has an average distribution ( $\mu$ ) and standard deviation ( $\sigma$ ), and then, for at least 30 samples, the sampling distribution has an approximate normal distribution [Jo04]. In this case, JMeter provides the *average response time* that represents  $\mu$ , *standard deviation* that represents  $\sigma$ , and the number of samples considered is 31. For further details about the *central limit theorem* refer to the literature [Jo04].

$$\text{Confidence interval} = X \pm Z_{\frac{\alpha}{2}} \left( \frac{\sigma}{\sqrt{n}} \right) \quad (6.1)$$

Formula 6.1 is used to analyse the confidence interval. From which,  $X$  defines the average of the sample ( $\mu$ ),  $Z_{\frac{\alpha}{2}}$  is a fixed value that comes from Table 6.5 that represents the probability of samples included into the calculated confidence interval (average response time). In this thesis the defined probability is 95%.  $\sigma$  defines the standard deviation,  $n$  is the size of the sample, and the *confidence interval* represents the level of certainty of the sample.

Confidence interval level	Z
0.90	1.645
0.92	1.75
0.95	1.96
0.96	2.05
0.98	2.33
0.99	2.58

Table 6.5 Z confidence level intervals

(2) *Thresholds for response time*

Table 6.6 The well-known thresholds for response times, these are used for determining if the response time of operations is optimal. These thresholds are derived from research regarding human perceptual abilities and brain behaviour [Ni93]. This thesis applies these thresholds to give a criterion after calculating the confidence interval.

Threshold	Response time	Description
1	< 0.1 s	Users do not notice a delay.
2	0.1 - 1 s	Users will notice the delay but this won't interrupt their work flows.
3	1 s - 10 s	Users actively wait for a response and consciously consider this an interruption.
4	> 10 s	Users lose focus and start doing something else.

Table 6.6 The well-known thresholds for response times using Web applications

(3) *Validation of results*

Table 6.7 contains results generated by JMeter after running tests for CRUD operations, and also results after applying *confidence intervals*. The values for *retrieving* routes operation are replaced into the Formula 6.1, from which the confidence obtained is equal to 0.37s (see Formula 6.2). Thus, according to the *threshold 2* from Table 6.6, the response time is acceptable for the retrieve operation. Similarly, the confidence interval for the *creation* is 1.08 s, *update* is 1.76 s, and *delete* equals 0.13s, which means, that these results are acceptable considering *threshold 3* from Table 6.6. In general,

Operation	Average response time (sec)	Throughput (req/time unit)	Confidence interval (sec)	Standard deviation (sec)
Retrieve	0.347	4.9 req/sec	0.37	0.12798
Create	0.736	10,6 req/min	1.08	1.00397
Update	1.462	11,3 req/min	1.76	0.86025
Delete	0.604	6.9 req/min	0.13	0.38563

Table 6.7 JMeter Summary report after running tests for CRUD operations over routes

$$\text{Confidence interval} = 347 \pm 1.96 \left( \frac{127.98}{\sqrt{62}} \right) = 378,85\text{ms} = 0.37 \text{ sec} \quad (6.2)$$

### 6.3.3 Throughput analysis

*Throughput* metric is obtained dividing *number of request* by *total of time* (end time of the last sample – first sample time) [Jm13]. After running the test, the *throughput* for the *retrieve operation* is 4.9 requests per second (see Table 6.7). This is an acceptable result considering that in the worst case the maximum number of users performing this operation simultaneously does not exceed this result. These CRUD operations associated to routes are part of the configuration of LVS, so that, users do not use this functionality frequently. Similarly, the result for *creation* is 10.6 requests per minute, *update* is 11.3 requests per minute and *delete* equals 6.9 requests per minute.

In addition, *throughput* metric is also used for measuring where WAS reaches its overload point, however based on the Table 6.7, the behaviour of the WAS does not reach saturation in neither of CRUD operations.

## 6.4 Data quality analysis and evaluation

*Measurement methodology* is applied in this section to assess the quality of data by comparing data generated before and after the prototype was implemented. This methodology consists of activities that enhance the precision of the quality evaluation [BS06]. First activity involves the definition of relevant dimensions, also well-known as quality criteria, with their indicators. Second activity, concerns the measurement using *record matching technique* to compare data between different data sources based on dimension indicators, and the last activity comprises the *analysis of results*. Activities details are described as:

- 1) *Definition of relevant dimensions*: The dimensions to evaluate quality of data generated by the prototype comprise: *completeness* to assess whether required data are present or not. *Accuracy* encompasses the correct representation of real data, so that, if  $v = \text{“John”}$  and  $v' = \text{“john”}$ , then data are not accurate. *Consistency* is with regard to data values being the same along the systems. *Timeliness* defines data freshness. [BS06]. Table 6.8 describes the indicators associated to every dimension.

Dimensions	Indicator
Completeness	Missing records
	Extra records
Accuracy	Reflects real routes
Consistency	Data values are the same across different data sources
Timeliness	Data are up to date

Table 6.8 Data quality evaluation between FBSC and the new prototype

- 2) *Record matching technique*: This consists of generating a sample of records from the origin data source. In this case, FBSC's database, from which every record is searched using LVS' database, and the number of matches is stored and analysed to set the indicators. The same activity is performed between FBSC' database and the old mechanism for handling routes. The following procedure represent the way to obtain the results:
- i. Generate a report from LVS containing a fleet and its vehicles. This report is considered as constraint to generate the sample data from FBSC's database.
  - ii. Create a report from FBSC's database that contains waypoints for every vehicle defined in *step a*.
  - iii. Generate a report with waypoints for every vehicle from LVS database. The fleet used is the one from *step a*.
  - iv. Matching records from reports obtained in *step b and c*. Thus, for every set of waypoints of LVS' report, a search operation is executed to find the same set in FBSC's report.
- 3) *Analysis of results*: Quality dimensions are analysed supported in the reports obtained from the *activity 2*, considering a private fleet and the number of vehicles equals 70, thus:

a. **Completeness**

$$\text{Ratio: } \frac{\text{missing routes records}}{\text{total routes in FBSC}} = \frac{9}{70} = 12,8\% \quad (6.3)$$

This dimension is acceptable, because only 12.8% (see Formula 6.3) of the required data are missing. Failures in hardware and procedure errors are cause of these missing records. However, these causes are out of the scope of this thesis. In addition, extra records were not found after importing the desired data.

b. **Accuracy**

$$\text{Ratio: } \frac{\text{waypoints with no differences}}{6924 \text{ waypoints evaluated}} = \frac{6924}{6924} = 100 \% \quad (6.4)$$

This dimension is acceptable. After comparing waypoints no differences were found between both reports. Coordinates, including their decimals numbers were evaluated, and both contained the same precision. The conclusion is that 100% of data are accurate (see Formula 6.4).

c. **Consistency**

$$\text{Ratio: } \frac{\text{waypoints with the same value}}{\text{waypoints evaluated}} = \frac{6924}{6924} = 100 \% \quad (6.5)$$

This dimension is acceptable. 100 % of the waypoints evaluated are the same by comparing waypoints from LVS to FBSC. Formula 6.5 shows the ratio applied.

d. **Timeliness**

This dimension is acceptable, because it applies the algorithm that imports up to date routes (see Listing 5.3) which is based on importing the last route per vehicle. In that way, 100% of the imported routes are up to date.

Chapter 7 evaluates in details results obtained in this chapter, including functional and non-functional requirements, and the data quality will be discussed.

---

## 7 Prototype assessment

---

In this chapter the success of the prototype is determined by evaluating the functional and non-functional requirements and the data quality based on the results of the tests from Chapter 6. A conclusion is given to tie the purpose and objectives of this thesis to the evaluation.

### 7.1 Evaluation of requirements

#### 7.1.1 Functional and non-functional requirements

Considering requirements defined in section 4.1.3 and 4.2.4, a set of pre-configured tests run on CRUD operations to validate and evaluate the correct running of the prototype. Tests were based on sending data by means of the Web GUI, and receiving it using JMeter. Results regarding functional and non-functional requirements are founded on JMeter's reports.

Results of functional requirements are analysed and evaluated from two perspectives. The first perspective encompasses the collecting of headers responses which are received by JMeter after server processes requests. Success of an operation is determined by these headers responses, e.g., headers' content with value equals 200, means that the operation requested was executed successfully.

The second perspective is focus on comparing expected data to real data which are generated after performing any CRUD operation. Thus, if expected data equals real data, then the requested operation is successful, otherwise unsuccessful. This perspective is based *on decision tables technique* (see section 6.2.2.1).

The performance of the prototype is based on two criteria: (1) *response time*, which is the elapsed time since a request is sent to the server, and subsequently a response is received by the user. (2) *Throughput*, which represents the number of requests accepted by servers during a time unit, thus this criterion determines if servers reach an overload point, i.e., servers do not process requests after a certain number of requests.

In this thesis, the *response time* of CRUD operation is acceptable if it is less than 10 s, which is supported on the well-known thresholds for response time, Table 6.6. In addition, the *throughput* is acceptable along as this does not reach an overload point.

Based on the above considerations functional and non-functional requirements are evaluated in the following section considering CRUD operations. The size of the sample data is equals 62, which is 31 (minimal sample required for applying the *confidence interval analysis*) multiply by 2 (number of concurrent threads). This size was applied for all tests.

##### 7.1.1.1 Creation

- a) *Functional requirements*: Test for the creation operation was conducted to validate whether this operation executes successfully or not. Results according to Table 6.1 and Table 6.3 reveal that for 62 requests to create routes the prototype created 62 routes. This

operation satisfies the use cases: *create routes* (Table 4.1), *importation of routes* (Table 4.5) and *store routes* (Table 4.6).

- b) *Non-functional requirements*: The objective of this test was to validate whether this operation is executed in less than 10 s, while the server does not reach an overload point. According to Table 6.7 the average *response time* to create a route is equals 0.347 seconds and the *standard deviation* equals 1.004 s, this means that the average is misleading, because responses contain a high variability. Thus, despite 0.347 s is below 10 seconds, the response time analysis is applied to obtain an average value closer to reality, which is equals 1.08 s, also below 10 s, i.e., creation operation is acceptable. In addition, the *throughput* obtained is equals 10.6 req/min, which is an acceptable value, since this number of request does not lead the server to an unexpected behaviour.

### 7.1.1.2 Update

- a) *Functional requirements*: The configured test to update routes was performed to validate whether this operation works properly. Thus, according to Table 6.1 and Table 6.3 a set of 62 updates was requested, and the result was that for every request the update operation was executed successfully. This test satisfies the use cases: *update routes* (Table 4.2), *import routes* (Table 4.5), and *store routes* (Table 4.6).
- b) *Non-functional requirements*: The objective is the same as the creation operation (see section 7.1.1.1). Thus, according Table 6.7 the average *response time* is equals 1.462 s, and *standard deviation* equals 0.860 s, this means that for 62 update requests the average response time is acceptable, because is below 10 s. In addition, applying *confidence interval* analysis the *response time* is equals 1.76 s, which it is still acceptable. *The throughput* for this test is 11.3 req/min, which is an acceptable value, since the server does not reach its overload point.

### 7.1.1.3 Delete

- a) *Functional requirements*: The purpose of testing the delete operation is to validate whether the execution of this operation fits its purpose. Thus, Table 6.1 and Table 6.3 expose that 62 requests to delete parent tours were performed successfully. This test satisfies the use cases: *delete routes* (Table 4.3) and *update routes transaction* (Table 4.7).
- b) *Non-functional requirements*: The objective is the same as the creation operation (see section 7.1.1.1). Thus, according to Table 6.7 the average *response time* to the delete operation is equals 0.604 s for 62 requests, and standard deviation equals 0.385 s, which means that response time values contain a moderate variability. After applying the *confidence interval* analysis the average *response time* obtained was 0.13 s, which is also below 10 s. In conclusion, average *response time* is acceptable, since it is below 10 s. *For the throughput criterion* 6.9 req/min is acceptable, because the server does not reach an overload point during the execution of this test.

### 7.1.1.4 Retrieve

- a) *Functional requirements*: Test for the retrieve operation was conducted to validate whether this operation displays the routes in the Web GUI successfully, or not. Table 6.1

and Table 6.3 provide that the retrieve operation was executed successfully in a range of 62 requests to the server. This test satisfies use cases: *display routes* (Table 4.4) and *retrieve routes* (Table 4.8).

- b) *Non-functional requirements*: The objective is the same as the creation operation (see section 7.1.1.1). According to the Table 6.7 the average response time is 0.347 s, and standard deviation 0.127 that means a reasonable variability among the response time values obtained. Likewise, *confidence interval* is equals 0.37 s, which is also below 10 s. As conclusion, the *response time* of the retrieve operation is acceptable. Logically, this operation is the fastest, because this only implies the retrieve of routes that are already stored in LVS' database, i.e., no need to bring routes from external databases.

*Throughput* is acceptable, since the server does not reach its overload point. The throughput value is equals 4.9 req/sec.

### 7.1.2 Evaluation of data quality

The aim of this section is to evaluate the data quality of the prototype. *Measurement methodology* (see section 6.4) was applied to reach this aim. Since this methodology includes analysis and evaluation, this section only presents the summary of the evaluation conducted in section 6.4.

Table 7.1 contains the summary after applying the *measurement methodology*, from which three attributes are presented: (1) *Dimensions* that contains the criteria to evaluate the data quality, (2) *Indicators* that let to measure the acceptance of criteria, and (3) *FBSC V.S Prototype* that provides results after indicators were calculated. Section 6.4 provides details about these results and their calculations.

Test fleet with 70 vehicles was considered as sample data, which means that 70 routes should be created in LVS. Thus, a conclusion for every criterion is presented in the following paragraphs considering this sample data:

- a) *Completeness*: The objective of this criterion is to assess whether data in LVS' database is the same as FBSC's database. Indicators to evaluate this criterion are *missing records and extra records*. Thus, from 70 routes created in FBSC only 61 routes are in LVS, which represents 12.8% of *missing records*. In addition *extra records* were not found by comparing both databases. To conclude, the *completeness* is acceptable despite missing records exist due to hardware failures or other possible problems.
- b) *Accuracy*: The aim of this criterion is to test if data in LVS' database represent the same data as FBSC's database. For 6924 waypoints that belong to 61 routes created in LVS, all of them represent the same precision by comparing them to FBSC. In this way, LVS handles routes with an *accuracy* of 100%, subsequently, *accuracy* indicator is acceptable.
- c) *Consistency*: This Criterion consists of evaluating if data values are the same across FBSC and LVS. Comparing 6924 waypoints that belong to 61 routes, both FBSC and LVS contain the same values. Thus, based on this comparison, LVS handles data with a *consistency* of 100 %, which is an acceptable indicator.



- d) *Timeliness*: The objective of this criterion is to determine whether data are up to date across FBSC and LVS. Thus, considering that *consistency* indicator is 100%, and the algorithm to *import routes* (see Listing 5.3) brings the latest routes. It is concluded that *up to date* is handled by LVS, which leads to an acceptable indicator.

Table 7.1 summarizes the conclusions mentioned in the previous paragraphs:

Dimensions	Indicators	FBSC V.S Prototype	Result
Completeness	Missing records	12.8 %	Acceptable
	Extra records	0 %	
Accuracy	Reflects real routes	100 %	Acceptable
Consistency	Data values are the same across different data sources	100 %	Acceptable
Timeliness	Data are up to date	100 %	Acceptable

Table 7.1 Data quality evaluation between FBSC and the new prototype

## 7.2 Conclusion

The result of the prototype validation in this chapter demonstrates that all objectives defined for this thesis are successfully achieved. Functional tests support the proper running of the prototype considering the specifications from sections 4.1.3 and 4.2.4. Reliable results regarding performance analysis ensure the correct behaviour of the application on certain conditions, while trustworthy quality of data are obtained after applying a methodology to evaluate the degree of data quality used in the prototype. These results guarantee a simulation close to the reality based on mass data generation.

---

## 8 Conclusion and future work

---

Daimler FleetBoard offers telematic services by means of a special hardware installed in customers' vehicles to collect and send data to the FleetBoard Service Centre (FBSC), which is in charge of receiving, processing and storing data generated by vehicles. The quality assurance and testing department guarantees that the telematic services meet their purpose and no errors disturb the productive system. LVS is a software tool for testing functionalities of FBSC platform, this tool behaves like a real vehicle by generating messages and sending them to FBSC platform, so that, LVS collaborates to meet the objectives of the department.

Tours are configured in LVS to simulate the behaviour of real vehicles, thus, fleet, driver, vehicle and a specific route are the principal attributes of a tour configuration. Once the configuration is finished in the Web GUI the simulation starts, LVS establishes the communication using a private protocol and SOAP messages are sent to FBSC platform which receives, processes and stores information related to the vehicle, i.e., speed, geographical position, driver name, and some additional data. Thus, if new functionalities are created, they are deployed to the test system, while testing tools, including LVS are used for testing these functionalities before they are deployed into productive environment.

Currently, tours' simulations are configured using routes that are created manually. These routes are created using Google Maps and a JavaScript to convert the positions to the GPX format. The first challenge with the existing procedure is that simulations lack of realism, because no route in LVS is equal to any route generated by real vehicles. The second challenge relates to manual process of creating routes. Thus, the objective of this thesis is to optimize the process of importing and creating routes in LVS by means of a prototype, which is supported on a concept and design that solves the current problems. Additionally, Nokia Maps is integrated in LVS to improve the current process visually.

This thesis provides a background concerning the components needed to perform tour simulations, including the internal protocol to establish communication between FBSC and LVS. Architecture, components and their interactions regarding mapping are explained, i.e., waypoints, tiles and their format representation. Quality criteria according to the quality model ISO 25000 is described to support the selection of suitable WMS provider to integrate mapping services into LVS, section 4.2 and Appendix B apply these criteria in details.

The quality assurance department concentrates its daily tasks on two parallel systems. The first is FBSC that supports the production environment. The second is the Integration Test System (ITS) that deals with the testing environment. Thus, these systems have the same technical configuration, but different purposes. This thesis develops the prototype based on the testing environment, which contains LVS to simulate vehicles' behaviour, and ITS as a copy of FBSC to test new and current functionalities by receiving messages from LVS and some other testing tools.

Two concepts are defined to tackle the two problems defined in this thesis. The first concept analyses the lack of realism that is caused by using routes from Google Maps. The solution

defined in this concept consists of replacing Google Maps service by FBSC's database to obtain simulation with more realism, since these routes are generated by real vehicles. In addition, this concept provides an algorithm that performs the creation of routes automatically based on functional requirements previously defined.

The second concept emphasizes in the optimization of the current process of importing routes using WMS. The problem defined in this section comprises the manual tasks to define a route with valid waypoints, and the solution consists of integrating Nokia Maps into LVS. Several approaches were defined to provide the best optimization, and an analysis and evaluation regarding the WMS providers were considered to come up with the most suitable provider, Nokia.

The implementation of the prototype is based on the concepts and the functional requirements. JMeter tool and black box testing strategy are used to evaluate the prototype with the functional requirements. The results of this evaluation show that the prototype runs as it was specified. Non-functional requirements are evaluated using performance analysis considering response time and throughput metrics, while data quality measurement strategy acts over quality criteria, i.e., completeness, accuracy, consistency and timeliness. The results after evaluate these non-functional requirements are successful.

Future work involves the verification and evaluation of the data completeness, because routes imported from production environment sometimes are empty. These sorts of data are come up from the human errors, e.g, drivers may forget to put the identification card in the TP, this means that a range of waypoints cannot be imported due to missing initial or final points of real routes. Another issue is with regard the mode of storing waypoints, establishing a manner to differentiate which vehicles store waypoints in tables *gpsdata* and *gpstracedata*, may improve the response time and throughput of the prototype.

---

# Appendix A

---

## Dataflow diagram (DFD)

### Definition

Dataflow diagrams are the presentation among various components in a system, including their relationships [IY10]. This a technique that helps to model systems from a general perspective showing how the input data are transformed into output. The benefits of using this technique are that a high level overview of the systems is obtained, and it is easy to understand by people with technical and no technical skills.

### Syntax and semantics

The principal symbols to represent a DFD are depicted in Table A.0.1. Circles are used for representing processes that are performed considering some business reasons. Arrows indicate how the information flows from one component to another, they are known as dataflow which contains a single piece of data. Double vertical lines are used for indicating that data are stored on it. The rectangles represent external entities, generally, end users and systems [AWT07].


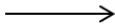
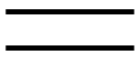
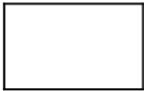
Symbol	Element name
	Process
	Dataflow
	Data storage
	External system

Table A.0.1 Fundamental components of DFDs

This technique is a hierarchical representation that consists of a top level diagram, starting from level 0, level 1, until level n diagram. Every level indicates a functionality of the system; the level 0 represents the main functionality and is known also as context diagram, this represents the boundaries of the system and the external entities [IY10], the rest of sub-levels correspond to functionalities integrated within the context diagram.

## Appendix B

This appendix contains the individual evaluations of WMS providers, considering costs, functionality, reliability and usability as factors to evaluate software according to the standards [ISO25000] and [ISO9126]. Google, Microsoft, OpenStreetMap and Nokia are the companies to evaluate using their Map services as reference.

### Individual evaluation of WMS providers

#### Google

Service: Google Maps	
Criteria	Justification
Zero cost of investment	This criterion is not fulfilled, due to the terms of use that Google exposes in its mapping services: "if your site meets any of the following criteria you must purchase the appropriate Google Maps API for Business license if: your site is only available to paying customers or your site is only accessible within your company or on your intranet or your application relates to enterprise dispatch, fleet management, business asset tracking, or similar applications"[GOO13a]. The limitations above restrict the use of any mapping service for free, because Daimler FleetBoard has as core business management of fleets as service.
Functionality	This criterion is fulfilled because the requirements: WMS and map API are offered by Google [GOO13a].
Reliability	This criterion is fulfilled because the mapping services offered by Google are highly reliable due to the large, distributed and replicated infrastructure along multiple datacentres around the world, in that way, fault tolerance and right performance is guaranteed by Google, which also reflected in 99.9 percent of uptime.
Usability	This criterion is fulfilled due to the Google's adaptation of the OGC's standards for accessing WMS by means of map APIs, from which is guaranteed a correct interoperability and easy forms to use mapping services along provider and customers [HPS08].

Table B.1 Google Maps: analysis and evaluation

## Microsoft

Service: Bing Maps	
Criteria	Justification
Zero cost of investment	This criterion is not fulfilled, because the free usage of WMS and API is limited to Services for business asset tracking and fleet management, which is stated in the terms of usage [MIC13]. Hence, because the fleet management is the core business at FleetBoard, the limitation above restricts the use of any mapping service for free.
Functionality	This criterion is fulfilled because the requirements: WMS and map API are offered by Microsoft [MIC13].
Reliability	The criterion is fulfilled because according to the SLA exists an infrastructure with a 99.9 percent of high availability, that guarantees continuity in the performance of the mapping infrastructure [MIC13].
Usability	This criterion is fulfilled using the same argument as Google evaluation table.

Table B.2 Bing Maps: analysis and evaluation

## OpenStreetMap

Service: OpenStreetMap	
Criteria	Justification
Zero cost of investment	This criterion is not fulfilled, because OpenStreetMap states the following in its Web site [OPE13]: <i>"OpenStreetMap data are free for everyone to use. Our tile servers are not"</i> , this means WMS are not free to use, they have an associated cost for usage, despite the data offered by OpenStreetMap is free of charge. Additionally, it is stated that <i>"The editing API is provided in order to edit the map data, not for read-only purposes or projects"</i> . Hence, this statement is not suitable, because the purpose of using the map API in this thesis is only with reading purposes.
Functionality	This criterion is fulfilled because the requirements: WMS and map API are offered by OpenStreetMap [OPE13].
Reliability	This criterion is not fulfilled since the limited capacity of resources does not guarantee the correct performance during failures [OPE13].
Usability	This criterion is fulfilled using the same argument as Google evaluation table.

Table B.3 OpenStreetMap: analysis and evaluation

**Nokia**

<b>Service: Nokia Maps</b>	
<b>Criteria</b>	<b>Justification</b>
Zero cost of investment	This criterion is fulfilled because Daimler FleetBoard holds a contract with Nokia for the usage of mapping services, thus, the cost of investment for using WMS and the API is zero.
Functionality	This criterion is fulfilled because the requirements: WMS and map API are offered by Google [NOK13].
Reliability	This criterion is fulfilled since the SLA offered by Nokia guarantees the right performance regarding data and hardware, even if failures occur [NOK13].
Usability	This criterion is fulfilled using the same argument as Google evaluation table.

Table B.4 Nokia Maps: analysis and evaluation

---

## Appendix C

---

This section provides a description of the tools used to develop the prototype.

### Description of tools used in this thesis

During the implementation several tools were needed to implement the prototype, thus Table C.1, describes every tool, including name, version, type of licence, type of licence, Web site to download, the package and the target operating system.

Name	Version	Type of Licences	Site to download	Package	Operating system
MySQL	5.5.27	GPL	<a href="http://dev.mysql.com/downloads/mysql/">http://dev.mysql.com/downloads/mysql/</a>	MySQL Server	Windows
Eclipse	Juno Service Rel. 1	EPL	<a href="http://www.eclipse.org/downloads/packages/release/juno/sr1">http://www.eclipse.org/downloads/packages/release/juno/sr1</a>	Base distribution	Windows
Maven	3.1.0	GPL	<a href="http://maven.apache.org/download.cgi">http://maven.apache.org/download.cgi</a>	Base distribution	Windows
Tomcat	7.0.12	GPL	<a href="http://tomcat.apache.org/download-70.cgi">http://tomcat.apache.org/download-70.cgi</a>	Base distribution	Windows
OpenJPA	2.2.0	GPL	<a href="http://openjpa.apache.org/downloads.html">http://openjpa.apache.org/downloads.html</a>	Base distribution	Windows
JMeter	2.9 r1437961	GPL	<a href="https://jmeter.apache.org/download_jmeter.cgi">https://jmeter.apache.org/download_jmeter.cgi</a>	Base distribution	Windows
Nokia JavaScript API	2.2.3	Commercial	<a href="http://api.maps.nlp.nokia.com/2.2.3/">http://api.maps.nlp.nokia.com/2.2.3/</a>	jsl.js	Independent

Table C.1 Description of software tools needed to implement the prototype. Type of licences: GPL=General Public License; Eclipse Public License = EPL.



---

## Appendix D

---

This section describes the configuration of JMeter before the validation and test of the prototype was accomplished. This configuration supports section 6.1.

### Description of the graphical configuration of JMeter

A mandatory configuration of JMeter is required before running tests scripts. Thus, this configuration comprises: a) *test plan* that contains the steps and components which take part during the test. b) *Workbench* that contains the server configuration, e.g., the port from which requests are received. The following paragraphs describe the configuration used for testing and validation the prototype.

#### a) Test Plan

This contains the ordered steps and components to run the test. Three components are relevant: (1) *Thread group* which contains the request willing to be sent to the server, (2) *Http Request Defaults* is used as reference component when several request contain the same path, and (3) *summary report* which is utilised to see the report after running the test. Workbench element is used to contain the HTTP Proxy server. Figure D.1 provides the visual result after the configuration has been performed.

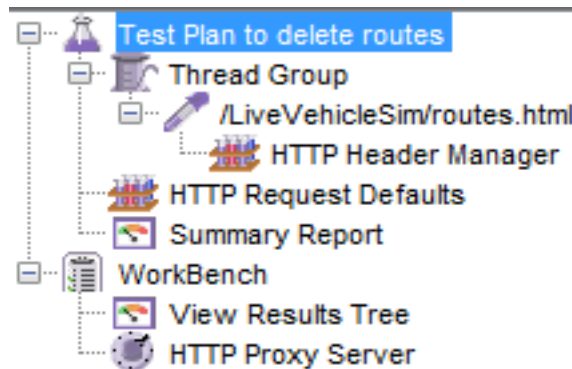


Figure D.1 Test plan configuration panel

(1) *Thread group*: This manages the number of threads of the test. Thus, considering Figure D.2, *name* and *comments* are part of the general description. If *continue* option is selected to avoid the test stops. *Thread properties* comprises number of concurrent threads, in this case 2, because it is the maximum number of possible users executing the importing of routes process. *Ramp-up* is the time to choose all threads. *Loop count* is the number of times to execute the test, 31 are chosen, because it is the minimum sample required for the confidence interval method (see section 7.1.1). The rest of properties are left it as default, because there are characteristics out of the scope of the current test and validation.

**Thread Group**

Name: Thread Group

Comments:

Action to be taken after a Sampler error

Continue

Thread Properties

Number of Threads (users): 2

Ramp-Up Period (in seconds): 1

Loop Count:  Forever 31

Delay Thread creation until needed

Scheduler

Figure D.2 Thread group configuration panel

- (2) *HTTP requests*: These are generated samples using a *recording controller* feature provided by JMeter. When the HTTP proxy server is activated, every request to the server from the WEB GUI is recorded, e.g., if 10 requests are sent to the server, JMeter generates 10 HTTP request entries, and they are used for the test. Attributes and data are display after they are recorded by JMeter.

**HTTP Request**

Name: /LiveVehideSim/routes.html

Comments:

Web Server

Server Name or IP: localhost

HTTP Request

Implementation: Protocol [http]: http Method: POST Content encoding: utf-8

Path: /LiveVehideSim/routes.html

Redirect Automatically  Follow Redirects  Use KeepAlive  Use multipart/form-data for POST  Browser-compatible

Parameters Post Body

Name:	Send Parameters With t
j_id_1l_SUBMIT	1
javax.faces.ViewState	fqWw+n0uZQLrY18C
org.richfaces.ajax.component	j_id_1l;j_id_1m
j_id_1l;j_id_1m	j_id_1l;j_id_1m
AJAX:EVENTS_COUNT	1
javax.faces.partial.event	click
javax.faces.source	j_id_1l;j_id_1m
javax.faces.partial.ajax	true
javax.faces.partial.execute	@component j_id_1l:
javax.faces.partial.render	@component
j_id_1l	j_id_1l

Figure D.3 HTTP request configuration panel

*HTTP request defaults*: This is a reference component used by JMeter when several requests belong to the same path. Figure D.4 provides the default configuration.

Figure D.4 HTTP request defaults configuration panel

- (3) *Summary report*: This summary report provides the results after running the test. This has the option to export result data to an external file setting the name and path. Although, JMeter provides many sort of reports, this is commonly used because specifies *number of samples* (# Samples) that represent requests to the server and their response, *average* that is the average response time, the minimum (Min) and maximum (Max) response time, the *standard deviation*, and the *throughput* which represents the number of request per unit of time. Figure D. 5 describes this configuration.

Label	# Samples	Average	Min	Max	Std. Dev.
TOTAL	0	0	9223372036854775807	-9223372036854775808	0,00

Figure D. 5 Description of the user interface for summary report configuration

b) *Workbench: HTTP Proxy Server*

This component of JMeter is used to observe and record every request to the server. While observing every request is added to the test plan. The important parameters to configure are: the *name* that describes the proxy, *port* which is used for listening HTTP request from the browser, *target controller* which is the test plan where JMeter records and stores requests. *Grouping* is applied to organize the request and responses. The rest of parameters are left it by default. Figure D. 6 shows this configuration.

---

HTTP Proxy Server	
Name:	HTTP Proxy Server
Comments:	
Global Settings	
Port:	8082
Test plan content	
Target Controller:	Test Plan > Thread Group
Grouping:	Store 1st sampler of each group only
<input checked="" type="checkbox"/> Capture HTTP Headers	<input type="checkbox"/> Add Assertions
<input type="checkbox"/> Regex matching	
HTTP Sampler settings	
Type:	
<input type="checkbox"/> Redirect Automatically	<input checked="" type="checkbox"/> Follow Redirects
<input checked="" type="checkbox"/> Use I	
Content-type filter	
Include:	Exclude:

Figure D. 6 User configuration interface for HTTP proxy server

---

## Appendix E

---

This section describes the results of testing the new functionalities in LVS using JMeter. Every test is built it recording every interaction between the Web GUI and the server. *Recording controller* is the feature provided by JMeter to record interactions [Jm13].

### CRUD operations results and descriptions

This section presents the results after testing the new functionalities in the prototype. The test scripts are configured to test CRUD operations using Nokia WMS provider and FBSC. Thus, after running these tests the variable to validate is *response message*, which tells whether the test was successful or not. This variable is provided by JMeter. Although, more variables are provided by JMeter only *response message* is considered, because the rest are out of the scope of this thesis.

**Retrieve routes using Nokia WMS:** This result is consequence of retrieving data from Nokia's server to load a map, and from the local server to load routes' data. In the sampler result tab is shown the response header from the server. Thus, the value contained in the variable *response message* is equals *OK*, which means that the request has succeeded, and the data requested are sent it to the browser correctly. Figure E.1 provides the result in details.

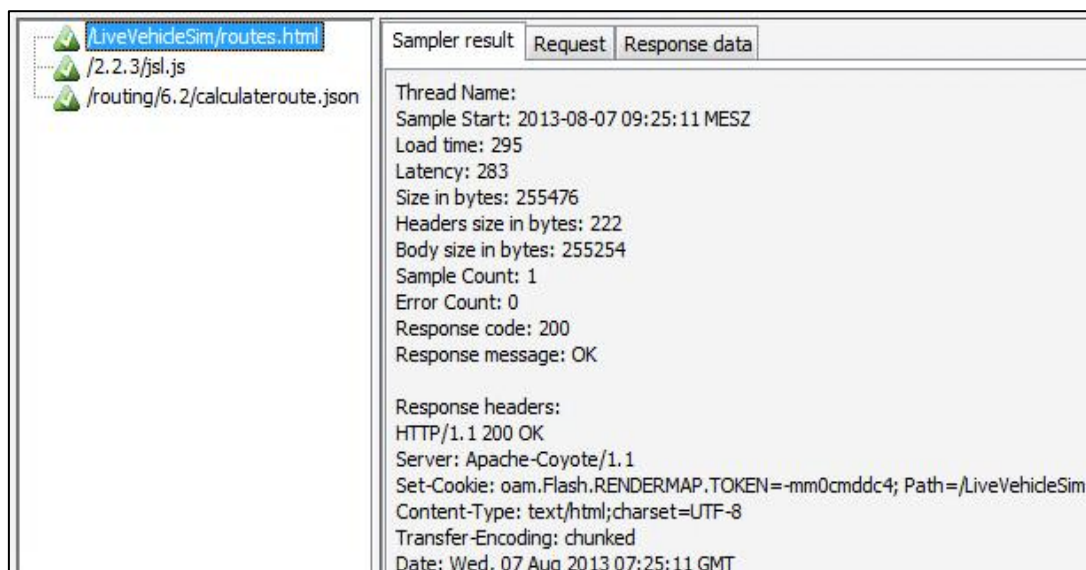


Figure E.1 Sampler result after running script that retrieves routes using Nokia as WMS provider

**Creation of routes using Nokia's waypoints:** After importing waypoints from Nokia's server to define a route, the operation to create routes was successfully performed. Response message: OK supports this result. Figure E.2 illustrates the results provided by JMeter.

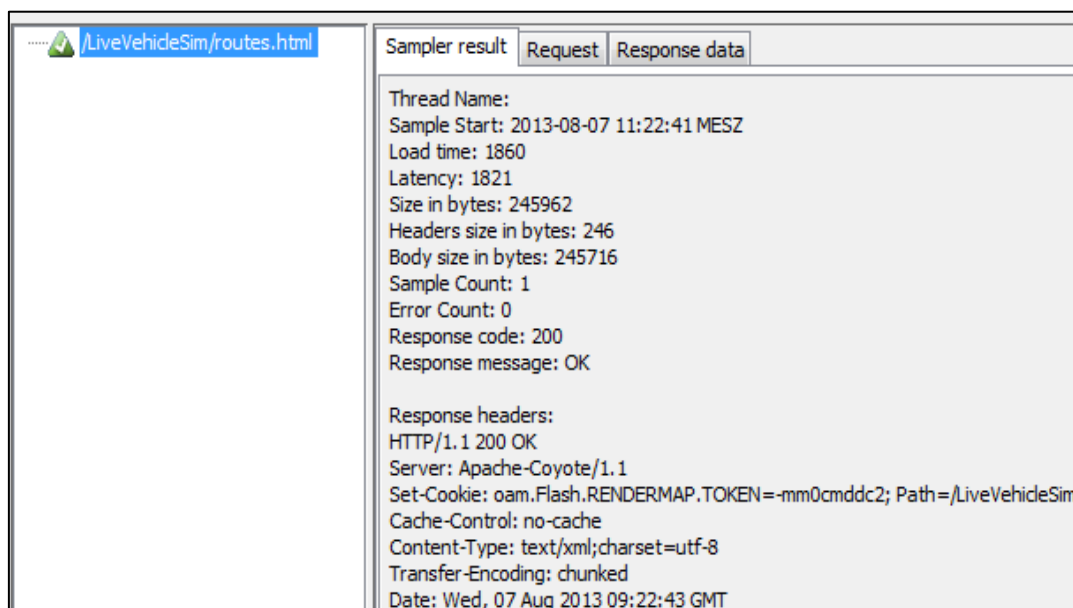


Figure E.2 Sampler result after running script that creates routes using Nokia as WMS provider

**Creation of routes using FBSC's waypoints:** After importing waypoints from FBSC's database, routes were created successfully. Response message: OK supports this result. Figure E.3 provides the description of this result.

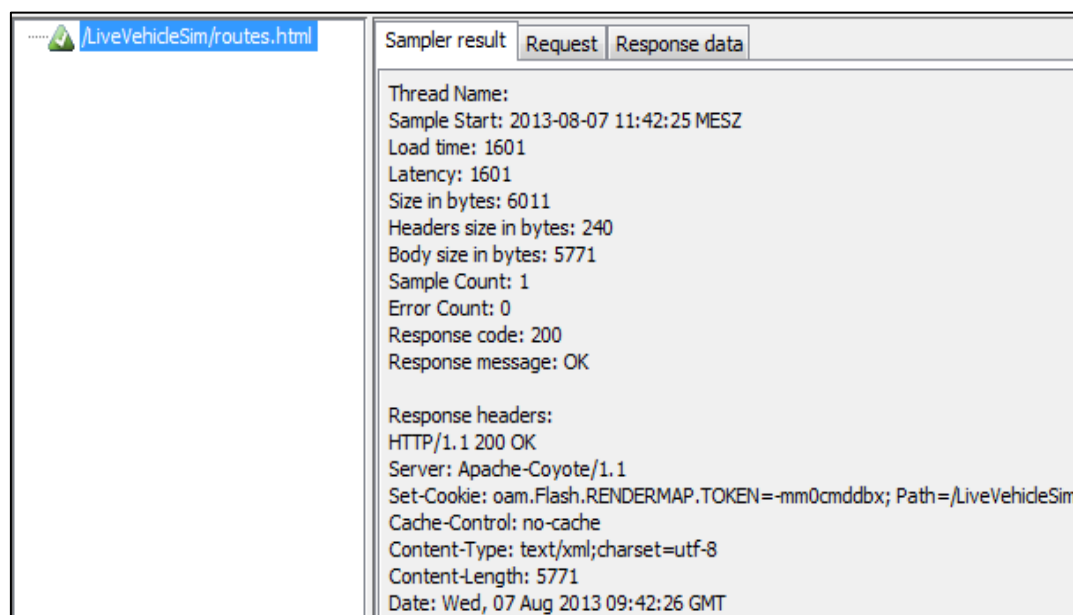


Figure E.3 Sampler result after running script that creates routes using FBSC's database

**Update routes using Nokia WMS:** The result after updating a sample route was successfully. The response header supports this success because the variable *response message* is equals OK, which means that the requested operation was performed in the Web server correctly. Figure E.4 shows the description provided by JMeter.

Figure E.4 Sampler result after running script that updates routes using Nokia as WMS provider

**Retrieve routes using FBSC's routes:** Figure E.5 provides the description provided by JMeter after the retrieve operation is performed. *Sampler result* tab contains a *response message* with *OK* value; this means that the operation was successfully executed.

Figure E.5 Sampler result after running script that retrieves routes using FBSC's database

**Update routes using FBSC's routes:** After running the test the update operation was performed satisfactorily, response message equal to *OK* guarantees this statement. Figure E.6 proves the success of the result.

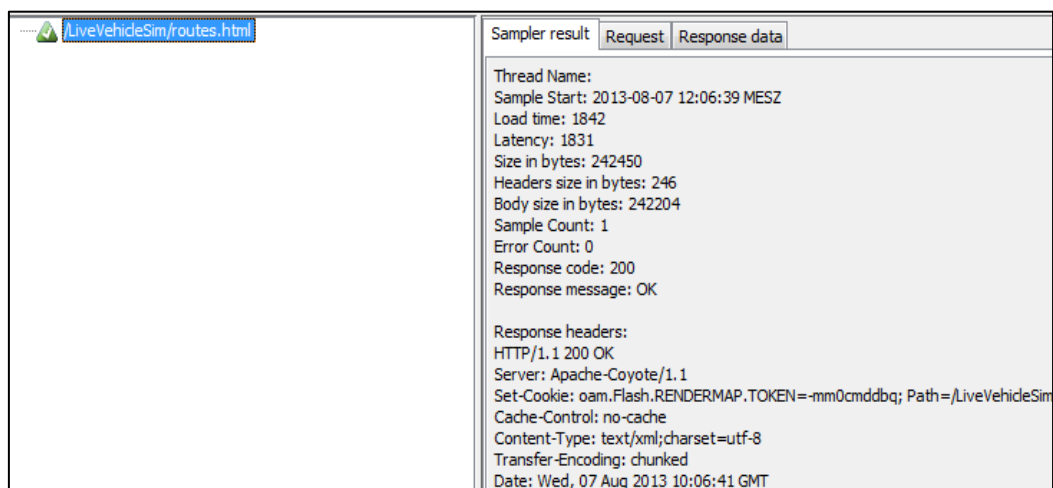


Figure E.6 Sampler result after running script that updates routes using FBSC's database

**Delete routes using FBSC's routes:** Selecting a route in the Web GUI and submitting the request to delete a sample route, is a success operation, since the response header confirms this by looking at the variable *response message*: OK. Figure E.7 proves the success of the result.

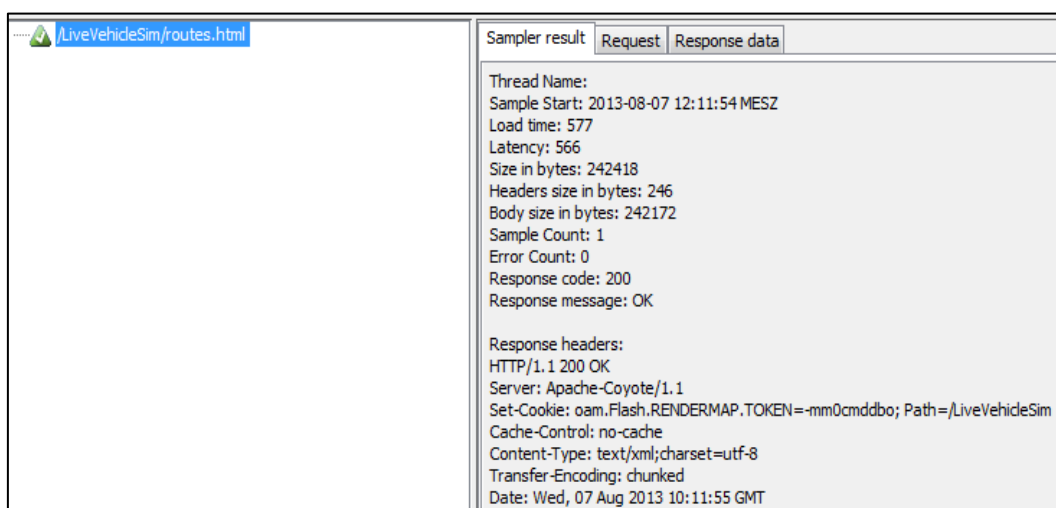


Figure E.7 Sampler result after running script that deletes routes using FBSC's database



---

## Bibliography

---

- [ACK05] H. Al-Kilidar, K. Cox, B. Kitchenham: "*The use and usefulness of the ISO/IEC 9126 quality standard*," Empirical Software Engineering, 2005. International Symposium on, vol., no., pp.7 pp. 17-18, 11.2005.
- [AG92] T. Arndt, A. Guercio: "*Decomposition of Data Flow Diagrams*", In: *SEKE: Knowledge Systems Institute*, S. 560-566, 1992.
- [AWT07] M. Abi-Antoun, D. Wang, and P. Torr: "*Checking threat modeling data flow diagrams for implementation conformance and security*", In Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. USA.2007.
- [BBC+07] P. Bansode, S. Barber, C. Farre, J. Meier, D. Rea: "*Performance Testing Guidance for Web Applications*," O'Reilly Media, Inc , 2007.
- [BKM04] D. Butorac, H. Kegalj, D. Matic: "*Data access architecture in object oriented applications using design patterns*," Electrotechnical Conference, 2004. MELECON 2004. Proceedings of the 12th IEEE Mediterranean, vol.2, no., pp.595,598 Vol.2, 12-15 May 2004.
- [BKS11] V. Bilicki, M. Kasza, V. Szűcs, A. Végh: "*Issues of Persistence Service Integration in Enterprise Systems*," Proceedings of PATTERNS 2011, The Third International Conferences on Pervasive Patterns and Applications. : 96-101, 2011.
- [BS06] C. Batini and M. Scannapieco : "*Data Quality: Concepts, Methodologies and Techniques*". Springer, Berlin, 2006.
- [CHO08] E. Chow: "*The Potential of Maps APIs for Internet GIS Applications*", Department of Earth and Resource Science University of Michigan Transactions in GIS, 12(2): 179–191, 2008.
- [CJM+10] B. Ciepluch, R. Jacob, P. Mooney and A. Winstanley: "*Comparison of the accuracy of OpenStreetMap for Ireland with Google Maps and Bing Maps*". Proceedings of the Ninth International Symposium on Spatial Accuracy Assessment in Natural Resources and Environmental Sciences 20-23rd p. 337, 09.2010.
- [CL09] P. Chen, S. Liu: "*Developing Java EE Applications Based on Utilizing Design Patterns*," Information Engineering, 2009. ICIE '09. WASE International Conference on , vol.2, no., pp.398,401, 10-11 July 2009.
- [DAI13] Daimler FleetBoard GmbH: "*Company Portrait*, Daimler FleetBoard GmbH, <http://www.fleetboard.com/info/en/company-portrait.html>, 2013.
- [DF05] G. Di Lucca, A. Fasolino: "*Testing Web-based applications: the state of the art and future trends*," Computer Software and Applications Conference. COMPSAC 2005. 29th Annual International , vol.2, no., pp.65,69 Vol. 1, 26-28 July 2005.
- [Ec13] Eclipse, Eclipse. <http://www.eclipse.org/>.
- [FLZ10] J. Fu, Y. Li, K. Zhu: "*Research the performance testing and performance improvement strategy in Web application*," Education Technology and Computer (ICETC), 2010

- 
- 2nd International Conference on, vol.2, no., pp.V2-328, V2-332, 22-24. June, 2010.
- [GOO13a] Google Inc., “Terms of Use”, <https://developers.google.com/maps/terms>, 29.04.2013.
- [GOO13b] Google Inc., “*Google Maps for Business*”, <http://www.google.com/enterprise/>, 29.04.2013.
- [Hi05] R. Hightower: “*The JSF application lifecycle, Walk through the 6 phases of JSF's request processing lifecycle.* IBM, 2005. URL <http://www.ibm.com/developerworks/library/j-jsf2/>.
- [HPS08] M. Haklay, C. Parker and A. Singleton: “*Web mapping 2.0: the neogeography of the GeoWeb*”. Geography Compass, 2, 2011-2039, 2008.
- [ISO9126] ISO/IEC, ISO/IEC 9126:1: 2005. *Software engineering: Product quality*. Geneva, Switzerland: ISO/IEC, 2011.
- [ISO 19142] ISO/TC, ISO/TC 211:2009: “*Geographic Information/Geomatics: Standards Guide*”. USA: ISO/TC, 2011.
- [ISO25000] ISO/IEC, ISO/IEC 25000:2005. *Software product Quality Requirements and Evaluation (SQuaRE)*, Geneva, Switzerland: ISO/IEC, 2010.
- [IY10] R. Ibrahim,S. Yen: “*Formalization of the data flow diagram rules for consistency check*”, International Journal of Software Engineering & Applications (IJSEA), Vol.1, No.4, 04.2010.
- [Jm13] JMeter, Apache JMeter. <http://jmeter.apache.org/>
- [Jo04] O. Johnson: “*Information theory and the Central Limit Theorem,*” Imperial College Press, London, 2004.
- [LR01] A. Leff, J. Rayfield: “*Web-application development using the Model/View/Controller design pattern*”, Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International, vol., no., pp.118,127, 2001.
- [LS11] S. Li, L. Sun: “*Advantages analysis of JSF technology based on J2EE,*” Computer Science and Service System (CSSS), 2011 International Conference on, vol., no., pp.2008, 2010, 27-29 June, 2011.
- [LZ10] Y. Lai, S. Zhongzhi: “*An Efficient Data Mining Framework on Hadoop using Java Persistence API,*” Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on , vol., no., pp.203,209, June 29 2010-July, 2010.
- [LZ05] W. Li, C. Zhang: “*The Roles of Web Feature and Web Map Services in Real-time Geospatial Data Sharing for Time-critical Applications*”, Cartography and Geographic Information Science. Vol. 32, Iss. 4, 2005.
- [Ma13] Maven Project, Apache Maven Project. <http://maven.apache.org/download.cgi>.
- [MIC13] Microsoft Inc., “*Microsoft® Bing™ Maps Platform APIs' Terms Of Use*”, <http://www.microsoft.com/maps/product/terms.html>, 21.05.2013.
- [My13] MySQL, Download MySQL Community Server. <http://dev.mysql.com/downloads/mysql/>
- [Ni93] J. Nielsen: “*Usability Engineering,*” Academic Press, London, 1993.

- 
- [NOK13] Nokia, “Nokia Developer”, <http://www.developer.nokia.com>, 21.05.2013.
- [Op13] OpenJPA. Apache OpenJPA project. <http://openjpa.apache.org/>
- [OPE13] OpenStreetMap, “API usage policy and Tile Usage Policy”, [http://wiki.openstreetmap.org/wiki/API\\_usage\\_policy](http://wiki.openstreetmap.org/wiki/API_usage_policy), 21.05.2013.
- [Or13] Oracle. Java Server Faces Technology <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>
- [Ro06] Road safety authority: “*Guide to EU rules, on drivers’ hours regulation*”. Regulation EC No. 561, 2006.
- [SCH13] A. Schneider: “*About GPS Visualizer*”, Adam Schneider, <http://www.gpsvisualizer.com/>, 7 July 2013.
- [Su13] Subversion, Apache Subversion Project, TortoiseSVN. <http://tortoisesvn.net/>, 7 July 2013
- [SW12] M. Schmidt, P. Weiser: “*Online Maps with APIs and WebServices*”, Springer, Berlin, 2012.
- [To13] Tomcat, Apache Tomcat. <http://tomcat.apache.org/>.
- [Tu06] A. J. Turner: “*Introduction to Neogeography*”, O’Reilly Media, Inc, 2006.
- [Va08] Y. Vasiliev: “*Beginning Database-Driven Application Development in Java™ EE*,” Apress, Reading, New York, First edition, 2008.
- [WW10] Y. Wang, Q. Wu: “*Performance Testing and Optimization of J2EE-Based Web Applications*,” Education Technology and Computer Science (ETCS), Second International Workshop on , vol.2, no., pp.681,683, 6-7 March, 2010.



---

## Declaration

I hereby declare that the work presented in this thesis is entirely my own.  
I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.  
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.  
The electronic copy is consistent with all submitted copies.

John Velandia:

Stuttgart, 07.10.2013