Final Report

Project Group ALISE

Project Group 2012/2013

Advisors: Daniela Nicklas, Stephan Janssen, Michael Wurst

Reviewer: Jun.-Prof. Dr. Daniela Nicklas

Staff and Authors:

Andreas Rehfeldt, Dennis Höting, Jens Runge, Kamil Knefel,
Lena Eylert, Marcus Behrendt, Marina Borchers, Mischa Böhm,
Mustafa Caylak, Robert Friedrichs, Sabrina-Cynthia Schnabel,
Timo Lottmann

Abstract:

This document describes the project of twelve students from the fields of business informatics and computing science. The project is supported by IBM Research Dublin and aims to produce a large distributed application for realistic synthetic sensor data processing from the generation to the visualization phase. Furthermore, it contains all organizational documents, documentation of the project progress, and the necessary working with documentary material.

Published on: Friday, May 10, 2013

Acknowledgment

We would like to acknowledge and express our gratitude to the contributions of the following group and individuals who made this project possible:

Our advisor Daniela Nicklas for her stimulating suggestions, knowledge, experience, her encouragement and her commitment.

Our second advisor Stephan Janssen for his numerous suggestions and his help with Odysseus, and our advisor at IBM Research in Dublin Michael Wurst for the help and inspiration he extended.

We would like to thank Ralf Krause for providing us with the needed hardware and supporting us in many other ways.

Gratitude is given to Marco Grawunder and Dennis Geesen for their help and support with Odysseus.

Manfred Baumgart and the University of Oldenburgfor supporting in general and the financial support.

We would also like to thank the OFFIS, the institute ofcomputer science in Oldenburg, for the possibility to use rooms and hardware.

Special thanks goes to the department of regional development in Oldenburg for their significant financial support. This support made it possible to visit the CeBIT in Hannover and the BTW in Magdeburg, and to publish high-quality documentation.

A thanks also goes to IBM Research Dublin for providing us with software and for the loose cooperation during the project.

The state of Lower Saxony deserves thanks for making it possible to exhibitthe project results at the CeBIT 2013 in Hannover.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ALISE	Advanced Live Integration of Smart City Environments
EJB	Enterprise JavaBeans
CC	Simulation Control Center
DSMS	Data Stream Management System
HLA	High-level Architecture
IBM	International Business Machines Corporation
JPA	Java Persistence API
OC	Operation Center
SOA	Service-orientated Architecture
SVN	Subversion
RTI	Run-Time Infrastructure
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Introduction

The present document is the Final Report of the project group ALISE. It describes the development of a smart city application for generation and visualization of sensor data. The project was performed by twelve students from the Carl von Ossietzky University of Oldenburg over a period of 12 months.

The smart city application models a fictional city in which synthetic sensor data are generated using statistical methods. The road and public transportation network of the city of Oldenburg in Lower Saxony, Germany is used as a starting point. Optionally, the application is extensible by other cities. The synthetic sensor data comprise the most important data from the traffic, weather and energy sectors. The project team followed an approach of a context-based sensor simulation. The data generation runs periodically and is affected through scalable parameters. The adjustable parameters include, among other things, the weather phenomena (e.g. rainfall, heat, etc.), the population density, the amount of vehicles and sensors (e.g. low to high density of sensors). Another important design criterion is the loose coupling of the individual components of the system. Through this advantage the application is easily extensible.

The focus of this project was the generation as well as visualization of sensor data. The representation of sensor data is performed using a map service and a dynamic dashboard of a business intelligence solution in a web application. The web application also includes settings, which influences the data generation and visualisation.

This document descipes the final application of this project as well as the process towards this application. It is divided into eleven chapters. These chapters bascily represent different parts of the project reaching from the fundamental research, the project organization and the design over the implementation and the description of the views to userguides, project evaluation and the project conclusion.

1.1 Motivation

Due to the increasing Internet networking and the ubiquitous sensors, ahuge amount of data is created. Thus, the logical data connection between different Smarter Planet initiatives from several companies is intended [SIE12, CIS12].The student project group ALISE (Advanced Live Integration of Smart city Environments) builds with the aid of IBM Research Dublin a smarter city application to simulate a fictitious city.

One part of the IBM „A Smarter Planet“ initiative aims to connect different sensor data. To create a smarter planet and to solve problems, it is necessary to deduce expressive knowledge by using sophisticated analysis methods. These problems cannot be solved without networked knowledge [IBM12].

The processing of the amount of data and data streams needs efficient systems. Those systemsareoften tested insufficient, because there are less or not enough synthetic sensor data. As a conclusion there are special data generators e.g. BerlinMOD [BDG09] or „Network-based Generator of Moving Objects“[Bri02]. Those generators create synthetic sensor data, which can be injected to a system.However, most of the systems do not have a comprehensive and event-driven simulation of traffic, energy and weather as well.

The focus of the project group ALISE (Advanced Live Integration of Smart City Environments) is the creation and visualization off synthetic sensor data. The groupcooperates with IBM. Those data is the basis for the city simulation. It includes several areas from the traffic (e.g. the private transport and the local public transport as well), weather, energy generation and energy consumption. The uniquecharacteristics of the system are the individual adjustable parameters and events (e.g. weather events can be influenced by a graphical interface). The effects of those adjustments affect immediately the simulation, the sensors and the visualization.

1.2 Purpose of the System

The purpose of the system is to create a smarter city simulation that simulatesa real or fictitious city including the street network. This street network is represented as an abstract graph. This graph consists of nodes and routes. All nodes are divided into work and home nodes, based on the zone classification of the Open Source map from OpenStreetMap. Furthermore, this simulation is able to create synthetic sensor data for traffic, energy and weather. Using mathematic algorithm and several synthetic sensors inside the simulation generates this data. It is also possible to read in real sensor data by using a special interface.

The system is able to generate cars manually by using the Operation Center and to generate cars automatically by choosing several events e.g. a heavy rainfall. This simulation shows the traffic density depending on the weather, population, work- and holiday and as well on the time of day.

1.3 Structure of this Document

This Document is separated into ten parts. The first part contains the introduction and the purpose of the system. The second part is about the fundamentals. It contains IBM Cognos, IBM InfoSphere Streams and the Smarter Planet Initiative.The third part describesthe project organization within the procedure model SCRUM and the corresponding documents. In addition, it contains a record about the SVN of the group, some rule documents and social media applications. The fourth part presents the design of the software. It contains the design document, which describes the design structure, the components of the system, and the functional and non-functional requirements. Furthermore, it describes the architecture, first ideas of the visualization (Operation Center and Simulation Control Center) and different types of sensors, which are important for the simulation. The fifth part documents the Implementation of the software. In the sixth part the created views are described in detail. Sucsequent the Seminars are outlined, followed by a User guide in the eigth part. A project evaluation is given in the ninth part. The participant in the fairs where the group presented the system is described in the tenth part. The last part gives a conclusion on the project.

2 Fundamentals

This section describes the fundamentals of this project. It contains the IBM Smarter Planet Initiative and our realization. Furthermore this section contains a description of the tools IBM InfoSphere Streams and IBM Cognos. As well this chapter contains several attributes from the seminars. Those seminars can be reviewed by looking in chapter 7.

2.1 Smarter Planet/City Initiative

"The 19th century was a century of empires. The 20th century was a century of nation states. The 21st century will be a century of cities" [Wellingtion E. Webb, 200 DK09, Page 13]. This statement from E. Webb shows that the relevance of cities will be grown in the future. Prognoses also show that the city population will grow up till five billions. In fact of this, rising cities will be confronted with several problems. They must handle the water and energy supply and the infrastructure as well. These reasons build the foundation of the IBM Smarter Planet/ City Initiative. IBM has the vision that "[…] a smarter city [...] makes optimal use of all the interconnected information available today in order to better understand and control its operations and optimize the use of limited resources" [IBM, 2009 IBM09]. This vision contains new technologies and has two main targets and five other targets:

- **Quality of living:** A smarter city should be organized and build up under the aspect of an optimal use.
- **City evolution:** A smarter city should concentrate of innovative technologies and a smart infrastructure.

The other five targets of the Smarter Planet Initiative contain the water supply, traffic, buildings, public security and energy. The main important targets for the Project ALISE are the traffic and energy supply.

- **Traffic:** This target influences a city. The amount of cars influences the traffic flow as well as the traffic behavior. Some people use their car to get to work every day and some use them just for a short distance. There are some methods to improve the traffic flow, for example build bridges or to raise the number of busses, but all of those methods cannot fix the traffic problems.
- **Energy Supply:** Energy is one aspects of a city. Mostly, in every home are some electronic devices, whichneed power. In a Smarter City for example, those electronic devices should use power in an environmentallycompatible way.

All other targets can be looked up the seminar paper "A view about the Smarter City Vision".

2.2 IBM InfoSphere Streams

This chapter is about the structure of the high-performance data stream middleware IBM InfoSphere Streams. In this section IBM InfoSphere Streams is only named InfoSphere.

InfoSphereis a software to analyze data streams in real time without saving the data. This is an important feature, because the amount of data could be very high and potentially infinite. The softwareis a distributed middleware, which is separated into three parts: Development Environment, Runtime Environment, and Toolkits and Adapters.

- **Development Environment:** This component is conceptualized for the process of data streams by an Eclipse Plug-In called Stream Studio and a Stream Process Language (SPL).
- **Runtime Environment:** This component is based on a Red Hat Enterprise Linux System. It allows the parallel usage of several hosts and several servers.
- **Toolkits and Adapters:** This extends InfoSphere with some functions like data mining.

More details about the architecture and the Streams Processing Language (SPL) is given in the seminar paper “IBM InfoSphere Streams” (see section 7.4).

There are three different patterns, which are important for the development process:

- **Filtering Pattern:** removes unnecessary data.
- **Merge/Aggregate Pattern:** To get a high performance real-time visualization it is necessary to merge and aggregate the data streams.
- **Parallel Processing Pattern:** To ensure the scalability of the application, a pattern for the parallel processing of a data stream can be used.

2.3 IBM Cognos

IBM Cognos is a Business Intelligence platform. This software serves the planning of operations, visualizations of the currentcompany situation and the optimization. The architecture of IBM Cognos is separated into three parts: presentation area, application area and data area. This architecture follows the SOA Architecture and serves the realization of reliability and the scalability.

- **Presentation area:** The presentation area is the user interface. Within this area the user is connected to the Business Intelligence platform. It is realized with a client program or a web surface.
- **Application area:** The application area is a loosely connected service, which contains the components: IBM Cognos Content Manager, IBM Cognos Report Server and IBM Cognos Dispatcher.
- **Data area:** The data area contains the IBMCognos security, the configuration settings and the meta data information.

IBM Cognos has four main components. The first component is the Framework Manager. This component administrates the disposable packages and makes them available for the users. The second component is the real-time monitoring. It administrates and visualizes the real-time data streams. The fourth component is the Report Studio, which edits and publishes data. The last component is called Business Insight. This component enables the user a fast creation of a dashboard and an adaption and expansion of existing resources.

More details about IBM Cognos can be looked up in the seminar paper "IBM Cognos Business Intelligence".

2.4 Enterprise JavaBeans

The Enterprise JavaBeans (EJB) are standardized components within a Java-EE-Server. They simplify the development of complex separated software systems with Java. EJB are important for the implementation of a company application. EJB can be used by several different characteristics and can be divide into three various components, which are described short in the following:

- **Entity Bean:** Entity Beans model persistent durable system data.
- **Session Bean:** Session Beans describes user interaction with the system. Session beans can also be divided into stateless and stateful Session Beans.
- **Message Driven Bean:** Message Driven Beans are necessary for an asynchronous communication between EJB Systems.

3 Project Organization

This section presentsthe project organization in detail. First of all, a general introduction of the used process model SCRUM follows. It continues with our realization of the process model. After that the related Product Backlog and the Sprints are described. General rules for the tasks of the Product Backlog are clarified next. The project plan, a description of the working meetings and the defined decisions of the project group follow. Afterwards, the management of documents, the realized team events and the used social media tools to publish new information about the project progress finalize this section.

3.1 Process Model SCRUM

SCRUM is an iterative and incremental process model, which is used for software development. It contains several roles, events and artifacts for the development process. Short introductions of these components are described in the following. More details about SCRUM can be found in the seminar paper “SCRUM und seine Anwendungen für die PG ALISE” in chapter 7.6.

There are three important roles:

- **The Product Owner** is the product manager. He or she takes the role of the customer and is responsible for the quality of the product and teamwork. The task of the product owner also compromises the administration of some relevant documents like the Product Backlog.
- **The Development Team** is a team of maximal nine members who are self-organized specialists. This team creates the final product. Only the team decides the processing of the tasks from the Product Backlog. Another important characteristic is the workspace. Every member has different workspaces, but the other members should have enough knowledge to assume the work of every other member.
- **The SCRUM Master**can be a member of the team. He or she represents a servant-leader, which means that he takes the responsibility for the observance of the SCRUM rules.

Different documents, which are described below, are created during the process model progress:

- **The Product Backlog** is a priority-sorted list of tasks that are necessary for the final report.
- **The Sprint Backlog** is a document, which contains a subset of tasks from the Product Backlog. These tasks have to be done in the currentsprint.
- **A release plan and burn down charts** are used for the milestone planning to describe a deadline for the software and to show a list of tasks, which must be done to create the software.

SCRUM is separated into five events:

- **A sprint** describes a time window of one to four weeks. Every sprint is followed by the next sprint and so on. Every sprint ends with a complete increment of the product or component.
- **The Sprint Planning Meeting** acts as the opening of every sprint and creates the work plan. This meeting should not take longer than eight hours.
- **The Daily SCRUM** is an everyday and short meeting about 15 minutes for the development team. Its assignment is only to inform the team about the arranged tasks and the problems that might have appeared.
- **The Sprint Review** takes place in the end of every sprint. It presents the results of the previous sprint and takes round about four hours.
- **The Sprint Retrospective** is a three hours meeting in the end of a sprint, after the sprint review. This meeting lays down new rules for the improvement of performance.

3.1.1 SCRUM Realization of the Project Group

The project group decided that it does not need all of the SCRUM aspects explained above. All roles are assigned to the participants. Due to the cooperation with IBM Research Dublin the Product Ownerswere the client from IBM and the project advisors. The development team consisted of the whole project group and the SCRUM Master was also one person from that team. The SCRUM Master changed biweekly.

The mentioned documents, like the Product Backlog and the Sprint Backlog, were used in the implementation process. A release plan and a burn down chart were not necessary because other documents for the milestone planning replaced their functions.

Four events of the SCRUM process were satisfied. Only the meeting for the Sprint Retrospective was shorter and included in the Sprint Review. The project group normally had a two-week sprint including two working meetings a week.

3.2 Backlogs and Sprints

Due to the fact that the project group uses the SCRUM process model, the Product Backlog and the Sprint Backlog were the important organizational documents. The two documents contain all tasks that had been completed by the group members during the sprints. Theywere updated biweekly during the sprint planning meeting. The group listed priority-sorted tasks, which should be done in the upcoming sprint. Those tasks are written down in a detailed way. If a task was too comprehensive, it was necessary to minimize the work, because it might be difficult to complete it in the sprint. Small and detailed tasks are easier to manage then the bigger ones.

The sprint was a period of work for two weeks. If there were special events like an exam phase, the duration of a sprint could be shortened or the amount of distributed tasks could be minimized. As

already mentioned, all given tasks were collected into the Sprint Backlog and the Product Backlog. Every group member received several tasks in each sprint. The goal was that these tasks were finished during the particular sprint. After the current sprint had ended, the team came together on a Sprint Review to talk about the completed and uncompleted tasks. These uncompleted tasks were transferred to the next sprint and marked in the Sprint Backlog as "in progress". Figure 1 shows an extract of the Product Backlog as of 14.01.13.

Product Backlog PG ALISE			14.01.13					
ID	type	topic	description	status	priority	target effort	operator	co-operator
0	general	Communication platform	install and configure Microsoft SharePoint	double checked	1	1	Jens	
1	visualization	IBM Cognos	configure IBM Cognos and test a baseline scenario	double checked	1	10	Kamil	
2	processing	IBM Infosphere Streams	configure IBM Infosphere Streams, model and send data	double checked	1	10	Mischa	
3	generation	Framework MASON	compare MASON with Desmo J	double checked	1	5	Mustafa	Marcus, Marina, Mischa
4	generation	Sensor data (weather)	generate sensor data (weather)	double checked	2		Andreas	
5	generation	Sensor data (energy)	generate sensor data (energy)	double checked	2		Timo	
6	generation	Sensor data (local passenger transport)	generate sensor data (local passenger transport)	double checked	2		Mustafa	
7	generation	Sensor data (public transport)	generate sensor data (public transport)	double checked	2		Lena	
8	visualization	Mobile visualization	visualization for mobile phones	delayed	4		Dennis	
9	visualization	Web server	install an IBM web server	canceled	1	5	Timo	
10	general	Integration test	Run the process with sample data	double checked	2		Mischa	
11	generation	Sensor mistakes	create mistakes for the sensors	in progress	2		Marcus	
12	visualization	Visualization with Google Earth	visualisation with Google Earth including sensor data	open	4		Dennis	
13	visualization	Visualization with IBM Cognos	visualization with IBM Cognos including sensor data	open	4		Kamil	
14	general	Final test	start a final test including all of the program parts	open	6		Jens	
15	general	Final documentation	handling in the final documentation	open	6		Andreas	
16	general	Organize journey	organize the journey to Dublin if necessary	delayed	5		Sabrina	
17	general	Trade shows	if necessary plan the meeting at the trade show [CeBIT, Hannovermesse,...]	in progress	3		Sabrina	
18	general	Homepage	fill the external Homepage	double checked	1	4	Dennis	Sabrina
19	general	Homepage	Create an internal area on the homepage to easily share (upload) documents with selected persons	double checked	1	3	Dennis	
20	general	Project group vision	revise and complete the vision	double checked	1	3	Sabrina	
21	general	Draft document	create a draft document	double checked	1	1	Jens	Marcus, Marina, Mischa

Figure 1: Extract of the Product Backlog

3.3 Definition of Done

Every task of the sprints should be reviewable. As a consequence, the "Definition of Done"highlights some general and some programming rules. Every group member should follow these defined rules. They might help to reach the aim without unnecessary problems. These rules are described in the following:

In general:

1. All Product Backlog requirements for this Work package/ User Story are fulfilled.
2. A Work package/ User Story must be reviewed by another team member (at least 1).

For programming:

1. All Product Backlog requirements for this Work package/ User Story are fulfilled.
2. Work package/ User Story must be reviewed by another team member (at least 1).
3. Code was produced (all 'to do' items in code completed).
4. Code was commented, checked in and run against current version in source control.
5. Peer reviewed (or produced with pair programming) and meeting development standards.
6. Builds without errors.
7. Unit tests were written and passed.

8. Deployed to system test environment and passed system tests.
9. Any build/deployment/configuration changes implemented/documented/communicated.
10. Relevant documentation/diagrams were produced and/or updated.

3.4 Milestone Plan

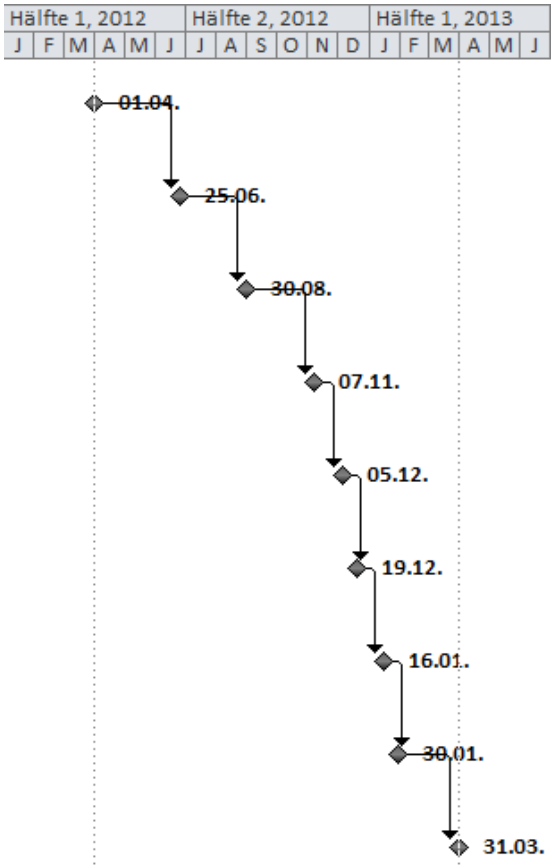


Figure 2: Milestone plan of the project group

The roadmap above illustrates the different milestones of the implementation process. Each milestone had specific tasks, which have been finished. The first milestone shows the end of the seminar phase. The next deadlines refer to the design and implementation process. For instance, the milestone from the 5th December 2012 had the following tasks:

- Running simulation with one traffic server.
- Energy, weather and GPS sensors produce an output.
- Sensor output are processed by IBM InfoSphere Streams and are forwarded to the servlets.
- Energy and weather sensors can be placed in the simulation by the Simulation Control Center.

The complete description of the every milestone can be found under "Milestone Planning" on page 199 in the appendix.

The milestone plan was created and managed with Microsoft Project 2010. A roadmap helped the project administration to coordinate the sprints. Moreover, it fixed the goals for the project to specific

dates. As a result, the planning could be managed and the project administration had a time schedule for the project progress.

3.5 Working Meetings

For one part every group member worked on their tasks at home, for the other part the group decided tomeettwice in aweek, on which the whole group worked together. These two days were connected with the daily SCRUM meeting. Moreover, every two weeks was a Sprint Planning Meeting. The SCRUM Master discussed the current Sprint Backlog and opened the next Sprint Backlog with new tasks for the group members on these meetings. The location for the meetings was a room in the OFFIS.

These working meetings were necessary to get in touch with the group members and to discuss currentproblems. The coordination of tasks was also an important topic. Furthermore, the group reflected the previous sprints to enhance the current progress. At the end, the high number of working meetings was very important for the project group's welfare.

3.6 Decision Document

This section illustrates the defined decisions of the project group. The decisions are ordered into three topics: organizational, seminar and programming issues.

Table 1: Organizational issues

Nr.	Date	Decision
1	2012-04-12	For all documents .docx shall be used. LaTeX (.tex) shall be used for scientific publications.
2	2012-04-12	SharePoint shall be used for managing the documents and for team collaboration.
3	2012-04-12	Alise is the team name and the corporate identity.
4	2012-04-12	Every member has to moderate and protocol at least one meeting.
5	2012-04-15	Visual Paradigm is used to create UML diagrams.
6	2012-04-19	Agendas should be in SVN at 0:01 clock on Wednesday before a meeting.
7	2012-04-19	Protocols should be in SVN at 0:01 clock on Saturday before a meeting.
8	2012-04-19	The protocol shall be reviewed from the leader of the next meeting.
9	2012-04-19	Whoever comes too late without excuse must bring a cake or cookies to the next appointment.
10	2012-04-19	Our meeting takes place every Thursday 16:00 to 17:00, before (and after) is our core business hours [Thursday as "PG-Thursday"].
11	2012-04-19	The agenda shall be sent via email and shall be uploaded to our SVN.
12	2012-04-26	All meetings take place in room U-64.

13	2012-04-26	Word documents are uploaded to SharePoint and LaTeX documents into SVN.
14	2012-08-02	The working areas are left clean and tidy after use.
15	2012-08-16	A review of the results is performed more frequently.
16	2012-10-08	If the SCRUM Master is too late for meeting, he has to bake cake for all members. Alternatively, he can bring biscuits to the next three meetings.

Table 2: Seminar issues

Nr.	Date	Decision
1	2012-04-05	The length of the seminar paper is 12 pages.
2	2012-04-05	The presentation of the seminar paper should take approx. 20 minutes.
3	2012-04-12	The working hours for the seminar will be recorded.

Table 3: Programming issues

Nr.	Date	Decision
1	2012-04-05	The documentation of the project will be written in English.
2	2012-04-05	The used process model is SCRUM
3	2012-04-12	The UTF-8 standard is used.

3.7 Management of Documents

The main administration tools to manage the documents of the project group are:

- Subversion (SVN)
- Microsoft SharePoint

The installed Apache Subversion (SVN) manages the source code and LaTeX documents. It contains also all important documents of the project group for backup purpose. The Microsoft SharePoint is a web based content management program and it is the main interaction point of the project members. It contains all documents with the file extension .docx and structures all organizational belongings. In particular, it administrates the working hours, the sprint reports, the product backlog, and so on. Furthermore, the calendar and the board are important tools of the project planning.

There are two options to publish documents of the project group. The first possibility is that the documents can be shared with the described Microsoft SharePoint. Another option is to publish the documents on the password locked page of the project group homepage[1].

[1] http://www.pg-alise.com/internal/index.php?dir=data

3.8 Team Events

Team events are important to connect all group members with each other. Regarding the project group, these events were non-working meetings to get in touch with other group members. In addition to many small team events, e.g. LAN parties or cinema visits, some bigger events were celebrated by the whole project group. The first bigger event of the project group was a go-kart race in Rastede. After the winner, Kamil, was crowned, the group members went to a restaurant in Oldenburg. The next team event was the Christmas party. The product owners were also invited. This party took place in a restaurant called Antalya in Oldenburg.

3.9 Social Media

For the social interaction the group decided to use Facebook, Twitter and an own homepage to represent our work and to inform other people about our current status. The project group used Facebook[2] and Twitter[3] to inform other people about the newest stuff.

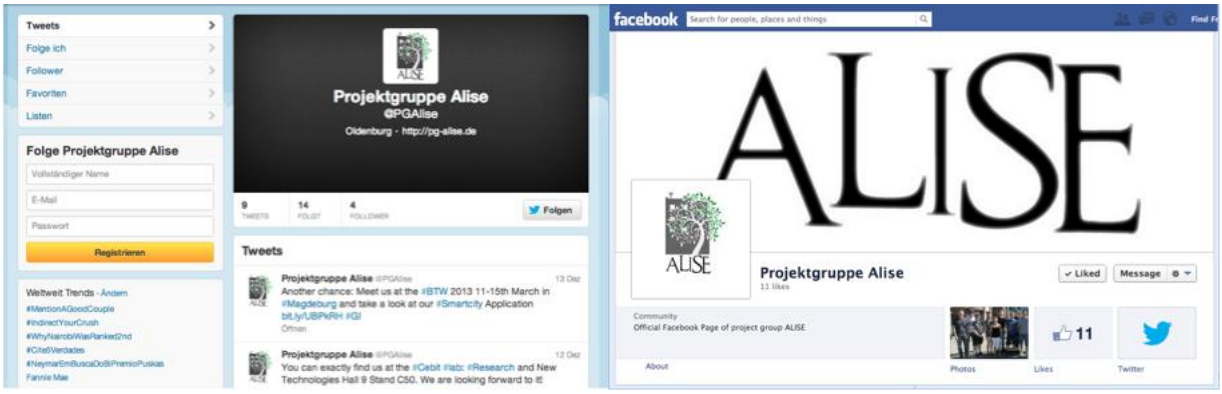


Figure 3: Twitter and Facebook pages of the project group

If a new goal was reached, one team member published a short statement. In contrast to the other social media, the homepage[4] does not show news about the current status. It informs people about the concept of the project group, the group members and lists the dates for the upcoming demonstrations.

[2] http://www.facebook.com/ProjektgruppeAlise
[3] http://twitter.com/#!/PGAlise
[4] http://pg-alise.com/

Figure 4: Homepage of the project group

4 Design

The next section describes the software draft, which is created during the earlier SCRUM meetings. It explains the requirements of the project group software, the first drafts concerning the architecture, the visualization and the general components of the simulation. Other important parts of the design document are the description of the implemented sensors, the management of the persistent data, and the possibilities of IBM InfoSphere Streams.

4.1 Design Document

The present document is the design document of the project group ALISE. It describes the development of a smart city application for generation and visualization of sensor data. The smart city application models a fictional city in which synthetic sensor data are generated using statistical methods. The road and public transportation network of the city of Oldenburg in Lower Saxony is used as a starting point. Optionally, the application should be extensible by other cities. The synthetic sensor data comprise the most important data from the traffic, weather and energy. The project team's approach is that the sensors can be simulated in the application and that they are context-based. The data generation runs periodically and may be affected through scalable parameters. The adjustable parameters include, among other things, the weather phenomena (e.g. rainfall, heat, etc.), the population density, the increase amount of vehicles and sensors (e.g. low to high density of sensors). Another important design criterion is the loose coupling of the individual components of the system. Through this advantage the application is easily extensible. The simulation of an entire city should be regarded as an optional feature.

The focus of the application is the generation of sensor data, and the visualization of the generated sensor data. The representation of sensor data is performed using a map service and a dynamic dashboard of a business intelligence solution in a web application. The web application should also include triggers, which can influence the data generation, such as how many tuples per second or what data are generated by the sensors. Optionally, the data can be visualized through a mobile application. The general data flow is divided into three phases. First, the sensor data will be generated in a program. Subsequently, the data are processed/aggregated into a streaming processing software and finally visualized by the mentioned visualization tools. Historical data are initially stored in a database. Via the previously mentioned triggers a return flow of data occurs from the web application to the program generating the of sensor data.

The project group chooses for the business intelligence solution IBM Cognos and for the map service Google Earth. The streaming processing is executed by IBM InfoSphere Streams. A DB2 database is responsible for the management of persistent data.

The structure of this document is divided into eight parts. At first, the general requirements of the smart city application are explained. In particular functional and non-functional requirements are

determined with the help of use case diagrams. Subsequently, a description of the general architecture follows and an overview of the different components is given. The next sections underline specific parts of the architecture. So the visualization, the simulation and the different sensor types are described. After that the management of persistent data and the exception handling of the smart city application are illustrated. Finally, the various features of the steaming processing software are given.

4.2 Requirements

At first, the functional requirements of the smart city application are described. After that the non-functional requirements of the general system follow.

4.2.1 Functional Requirements

In this section the functional requirements for the system are specified. The requirements are divided into the following subsections: Simulation Control Center and Operation Center. The Simulation Control Center contains starting the simulation, shutting down the simulation and simulating. The Operation Center is for observing the simulation. The two centers are explained in section 4.5.

4.2.2 Simulation Control Center

The following use case diagram shows the different functions of the Simulation Control Center (see Figure 5). The functional requirements are outlined according to the individual use cases of this diagram.

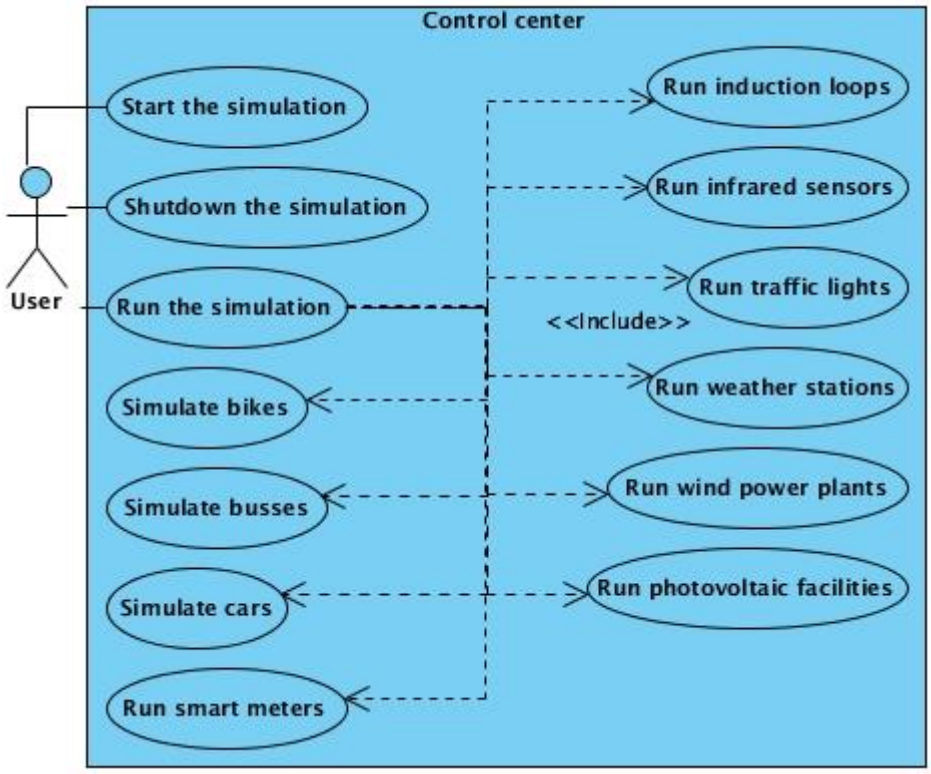


Figure 5: Use cases of the Simulation Control Center

Start the simulation

R1: The system must offer the user the possibility to start the simulation.

R2: The system must offer the user the possibility to set the start parameters for a simulation.

R3: The start parameters must at least be: City size, number of sensors (induction loops, infrared sensors in busses, traffic lights, cars, busses, bikes, photovoltaic facilities, wind power facilities, weather stations), simulated date and events.

R4: The system must offer the user the possibility to start an identical simulation.

R5: The system must be able to limit the creation of certain sensors.

Shutdown the simulation

R6: The system must offer the user the possibility to end the simulation.

R7: The system must offer the user the possibility to request a confirmation before shutting down the application.

Run the simulation

R8: The system must offer the user the possibility to simulate sensors of a city.

R9: The system must be able to simulate sudden events during the simulation.

R10: The system must be able to differentiate between work and home nodes.

R11: The system must be able to simulate weather dependencies.

R12: The system must adapt the simulated street network to a real street network.

R13: The system must adapt the simulated bus stops to a real bus network.

R14: The system must be able to update the sensors automatically.

R15: The system must offer the user the possibility to pause the simulation.

R16: The system must offer the user the possibility to end the simulation.

R17: The system must offer the user the possibility to restart the simulation.

R18: The system must offer the user the possibility to resume the simulation.

R19: The system shall offer the user the possibility to slow down the running simulation.

R20: The system shall offer the user the possibility to speed up the running simulation.

R21: The system will not simulate future days.

Simulation – Induction loops

R22: The system must offer the user the possibility to add induction loops to the simulation automatically from a previously stored file.

R23: The system must offer the user the possibility to add induction loops to the simulation manually.

R24: The system must be able to recognize whether the positions of the manually set inductions loops are on a street or not.

R25: The system must be able to distribute induction loops on the street network.

Simulation – Infrared sensors

R26: The system must offer the user the possibility to add infrared sensors to the simulated busses automaticallyfrom a previously stored file.

R27: The system must offer the user the possibility to add infrared sensors to the simulated busses manually.

R28: The system must be able to recognize whether the positions of the manually set infrared sensors are inside a bus or not.

R29: The system must be able to distribute infrared sensors on the simulated busses.

Simulation – Traffic lights

R30: The system must be able to add traffic lights to the simulation automaticallyfrom a previously stored file.

R31: The system must offer the user the possibility to add traffic lights to the simulation manually.

R32: The system must be able to recognize whether the positions of the manually set traffic lights are on a street or not.

R33: The system must be able to distribute traffic lights on the street network.

Simulation – Cars

R34: The system must offer the user the possibility to add new cars to the simulation.

R35: The system must offer the user the possibility to add cars automatically from a previously stored file.

R36: The system must offer the user the possibility to add cars manually.

R37: The system must be able to recognize whether the positions of the manually set cars are on a street or not.

R38: The system must be able to distribute cars on the street network.

Simulation – Bus

R39: The system must offer the user the possibility to add new busses to the simulation.

R40: The system must offer the user the possibility to add busses automatically from a previously stored file.

R41: The system must offer the user the possibility to add busses manually.

R42: The system must be able to recognize whether the positions of the manually set busses are on a street or not.

R43: The system must be able to distribute busses on the street network.

Simulation – Bike

R44: The system shall offer the user the possibility to add new bikes to the simulation.

R45: The system shall offer the user the possibility to add bikes automaticallyfrom a previously stored file.

R46: The system shall offer the user the possibility to add bikes manually.

R47: The system shall be able to recognize whether the positions of the manually set bikes are on a street or not.

R48: The system shall be able to distribute bikes on the street network.

Simulation – Photovoltaic facility

R49: The system must offer the user the possibility to add new photovoltaic facilities to the simulation.

R50: The system must offer the user the possibility to add photovoltaic facilities automaticallyfrom a previously stored file.

R51: The system must offer the user the possibility to add photovoltaic facilities manually.

R52: The system must be able to distribute photovoltaic facilities in a linear way in the simulated city.

Simulation – Wind power facility

R53: The system must offer the user the possibility to add new wind power facilities to the simulation.

R54: The system must offer the user the possibility to add wind power facilities automaticallyfrom a previously stored file.

R55: The system must offer the user the possibility to add wind power facilities manually.

R56: The system must be able to distribute wind power facilities in a linear way in the simulated city.

Simulation – Weather stations

R57: The system must offer the user the possibility to add new weather stations to the simulation.

R58: The system must offer the user the possibility to add weather stations automaticallyfrom a previously stored file.

R59: The system must offer the user the possibility to add weather stations manually.

R60: The system must be able to distribute weather stations in a linear way in the simulated city.

Simulation – Smart meter

R61: The system must offer the user the possibility to add new smart meters to the simulation.

4.2.3 Operation Center

The following use case diagram shows the different functions of the operation center (see Figure 6: Use cases of the operation center). The functional requirements are outlined according to the individual use cases of this diagram.

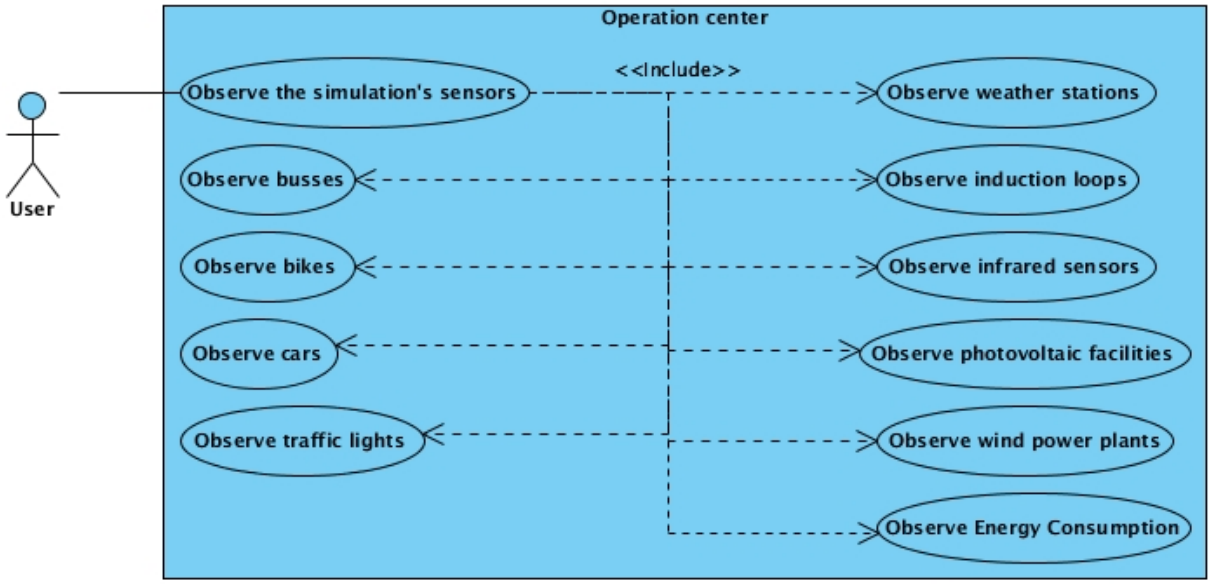


Figure 6: Use cases of the operation center

Observe the simulation

R62: The system must offer the user the possibility to observe the simulation through an operation center.

R63: The system must be able to show the updated sensors in the operation center.

R64: The system must be able to show detailed information about selected sensors.

R65: The system must be able to mark a running sensor in the city, where the detailed view is opened.

R66: The system must be able to save and load start parameters.

Observe the simulation – Induction loops

R67: The system must offer the user the possibility to display set induction loops in the city.

R68: The system must offer the user the possibility to hide all set induction loops in the city.

R69: The system must offer the user the possibility to unhide all set induction loops in the city.

R70: The system must offer the user the possibility to hide certain induction loops.

R71: The system must offer the user the possibility to unhide certain induction loops.

R72: The system must offer the user the possibility to display detailed information about the selected induction loop.

Observe the simulation – Infrared sensors

R73: The system must offer the user the possibility to display set infrared sensors in the city.
R74: The system must offer the user the possibility to hide all set infrared sensors in the city.
R75: The system must offer the user the possibility to unhide all set infrared in the city.
R76: The system must offer the user the possibility to display detailed information about the selected infrared sensor.

Observe the simulation – Smart meters

R77: The system must offer the user the possibility to display set smart meters in the city.
R78: The system must offer the user the possibility to hide all set smart meters in the city.
R79: The system must offer the user the possibility to unhide all set smart meters in the city.
R80: The system must offer the user the possibility to display detailed information about the selected smart meters.

Observe the simulation – Traffic lights

R81: The system must offer the user the possibility to display set traffic lights in the city.
R82: The system must offer the user the possibility to hide set all traffic lights in the city.
R83: The system must offer the user the possibility to unhide set all traffic lights in the city.
R84: The system must offer the user the possibility to hide certain traffic lights.
R85: The system must offer the user the possibility to unhide certain traffic lights.
R86: The system must offer the user the possibility to display detailed information about the selected traffic light.

Observe the simulation – Cars

R87: The system must offer the user the possibility to display driving cars in the city.
R88: The system must offer the user the possibility to hide all driving cars in the city.
R89: The system must offer the user the possibility to unhide all driving cars in the city.
R90: The system must offer the user the possibility to simulate cars with GPS-sensors.
R91: The system must offer the user the possibility to simulate cars without GPS-sensors.
R92: The system must offer the user the possibility to display the cars in a heatmap.
R93: The system must offer the user the possibility to highlight heavy loaded streets via a heatmap.
R94: The system shall offer the user the possibility to display a percentage of cars.
R95: The system must offer the user the possibility to display detailed information about the selected car.

Observe the simulation – Busses

R96: The system must offer the user the possibility to display driving busses in the city.

R97: The system must offer the user the possibility to hide all driving busses in the city.

R98: The system must offer the user the possibility to unhide all driving busses in the city.

R99: The system must offer the user the possibility to hide certain busses.

R100: The system must offer the user the possibility to unhide certain busses.

R101: The system must offer the user the possibility to hide certain bus lines.

R102: The system must offer the user the possibility to unhide certain bus lines.

R103: The system must offer the user the possibility to display detailed information about the selected bus.

Observe the simulation – Bikes

R104: The system shall offer the user the possibility to display driving bikes in the city.

R105: The system shall offer the user the possibility to hide all driving bikes in the city.

R106: The system shall offer the user the possibility to unhide all driving bikes in the city.

R107: The system shall offer the user the possibility to hide certain bikes.

R108: The system shall offer the user the possibility to unhide certain bikes.

R109: The system shall offer the user the possibility to display detailed information about the selected bike.

Observe the simulation – Photovoltaic facilities

R110: The system must offer the user the possibility to display set photovoltaic facilities in the city.

R111: The system must offer the user the possibility to hide all set photovoltaic facilities in the city.

R112: The system must offer the user the possibility to unhide all set photovoltaic facilities in the city.

R113: The system must offer the user the possibility to hide certain photovoltaic facilities.

R114: The system must offer the user the possibility to unhide certain photovoltaic facilities.

R115: The system must offer the user the possibility to display detailed information about the selected photovoltaic facility.

Observe the simulation – Wind power facilities

R116: The system must offer the user the possibility to display set wind power facilities in the city.

R117: The system must offer the user the possibility to hide all set wind power facilities in the city.

R118: The system must offer the user the possibility to unhide all set wind power facilities in the city.

R119: The system must offer the user the possibility to hide certain wind power facilities.

R120: The system must offer the user the possibility to unhide certain wind power facilities.

R121: The system must offer the user the possibility to display detailed information about the selected wind power facility.

Observe the simulation – Weather stations

R122: The system must offer the user the possibility to display set weather stations in the city.
R123: The system must offer the user the possibility to hide all set weather stations in the city.
R124: The system must offer the user the possibility to unhide all set weather stations in the city.
R125: The system must offer the user the possibility to hide certain weather stations.
R126: The system must offer the user the possibility to unhide certain weather stations.
R127: The system must offer the user the possibility to display detailed information about the selected weather station.

4.3 Non-Functional Requirements

In this section the non-functional requirements for the system are specified. The requirements are divided into the following subsections: technical-, process-, judicial-/contractual-, user interface- and quality-requirements. The quality requirements are subdivide into dependability, usability, efficiency and maintainability.

Technical requirements

R128: A relational database must be available for the application.
R129: Historical data of the simulation must be saved into a database.
R130: At least one server must be available for the application.
R131: An instance of IBM InfoSphere Streams in version 3.0 or later must be available for the application.
R132: The application must simulate its objects based on data send by InfoSphere Streams.
R133: The Streams application must receive the sensor data of the simulation on a TCP-Port.
R134: The Streams application must send data to the visualization component.
R135: The Streams application must save the progressed sensor data in a database.
R136: The Streams application shall aggregate the sensor data.
R137: The Streams application must use a dynamic filter to change to processed data at runtime.
R138: An instance of Cognos in version 10.1 or later must be available for the application.
R139: IBM Cognos shall assimilate the data output of the application.
R140: IBM Cognos shall assimilate the historical data from a database.
R141: IBM Cognos shall give the user possibilities to interact with reports.
R142: IBM Cognos shall give the user possibilities to interact with dashboards.
R143: The Operation Center shall call IBM Cognos to show reports of the running simulation.
R144: The system must have an Interface for Odysseus.
R145: The Simulation Control Center must be executable in Mozilla Firefox from version 19.
R146: The Simulation Control Center must be executable in Internet Explorer from version 10.
R147: The Simulation Control Center must be executable in Chrome from version 26.

R148: The Simulation Control Center must be executable in Safari from version 6.

R149: The application must use OpenStreetMap data for the street network.

R150: The application must use Google Maps to visualize the street network.

R151: Each sensor type must be controlled by a specific sensor controller.

R152: The application must be scalable.

R153: The application must be distributed.

R154: The application must be platform independent.

R155: The application must use a consistent simulation time.

Process requirements

R156: The Application must be programmed in Java and Javascript.

R157: SCRUM must be used as process model.

R158: The development must be test-driven.

R159: The development environment shall be NetBeans or Eclipse.

R160: For modeling UML-diagrams the software Visual Paradigm must be used.

R161: The UML-diagrams must be designed according to the UML 2.1-Standard.

R162: The document must be created with MS Word an exported as a PDF document.

R163: A test document must be created.

R164: The Documents must be written in English.

Judicialand contractual requirements

R165: Primary the application is not conceived for commercial use.

R166: The developers do not take responsibility for problems that occur during the runtime of the application.

R167: The developers do not take any responsibility for the violation of the system by third person.

R168: After the release of the application, the program must be open source.

User interface requirements

R169: The user interface shall be created as in the user interface guidelines.

R170: The controls shall be carried out with keyboard and mouse.

Quality requirements

Reliability

R171: The application shall be executable anytime in case the servers are accessible and no maintenance is being executed.

Usability

R172: The simulation shall be visualized in a comprehensive way.

Efficiency

R173: The application shall not have any performance issues during runtime.

R174: The application shall be programmed with efficient data structures.

R175: The data shall be free of redundancy.

Maintainability

R176: The application must not be maintained after the release.

R177: In the programming code comments must be created in JavaDoc.

R178: The programming follows the common Java code conventions (http://java.sun.com/docs/codeconv/).

4.4 Architecture

This section describes the general architecture of the simulation. The following component diagram (see Figure 7) shows all objects of the simulation. Referring to this diagram the general architecture is outlined.

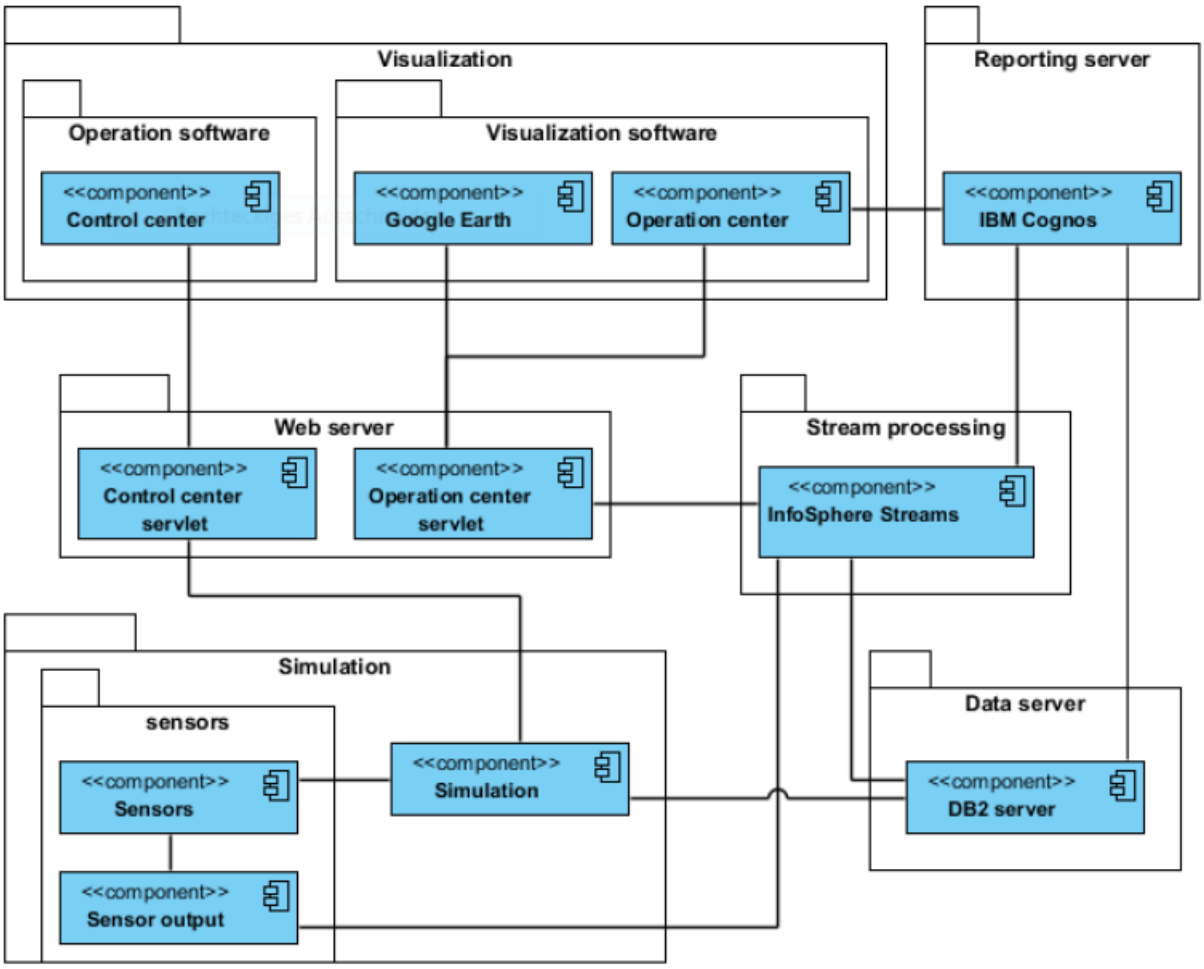


Figure 7: General Architecture

4.4.1 Package Simulation

The package *simulation* represents the maincomponent in the system's hierarchy. It contains controllers with whichthe user or administrator can interact (SimulationController in Simulation). Every action that can be performed during the simulation (for example all kinds of events) has to pass the SimulationController. Then it decides which lower component has responsibility for the action and delegates it by tunneling. Thus the SimulationController is aware of each component in the simulation. Furthermore this package holds all instances of Sensor that has been created during the simulation

The package *Environment* contains the environment for the virtual city. This is simulated by the *Simulation* component. The component provides interfaces to the outside of which the status of the environment can be queried at any time.

The *Sensors* package includes all of the sensors (see more in section "Sensors"). The number of sensors is arbitrary and is structured according to the types of sensors: Traffic Sensor, Weather Sensor and Energy Sensor. The different types of sensors can retrieve information from the *Simulation* component and use it for their measurements. The sensors have *Output* components, which assume the transmission of the measured values. Each sensor can have its own *Output* or an *Output* summarizes

the values of sensors to one data stream. The data streams of the *Output* components end up into the package *Stream Processing* and thus in the Data Stream Management System *IBM InfoSphere Streams*.

The specific architecture of the simulation is explained in section “Simulation”.

4.4.2 Package Stream Processing

The component *IBM InfoSphere Streams* is located in the package *Stream processing*. It handles a large number of sensor data streams coming from the sensor package. The processed data streams can end up in the package *data server*, *reporting server* and *web server*.

4.4.3 Package Data Server

The package *Data Server* includes an *IBM DB2* database to store historical data persistent. These data come from the package *Stream Processing* and can be queried in the package *Reporting Server*.

The management of the persistent data is explained in section “Management of persistent Data”.

4.4.4 Package Web Server

The package *Web server* is represented by a Jetty servlet container. In this servlet container, the components *operation center* and *simulationcontrol center* are deployed. The *simulationcontrol center* can influence the simulation via *Enterprise Java Beans* and the *operation center* can be influenced by the simulation via a restful-interface or by *IBM InfoSphere Streams*.

4.4.5 Package Reporting Server

The component *IBM Cognos* is located in the package *Reporting server*. With the help of business intelligence functions IBM Cognos can help analysis to take place. The basis for this data can be current data from the packet *Stream processing*, historical data from the packet *Data server* or different kind of data from the *Web server* package.

4.4.6 Package Visualization

The package visualization is composed of two subpackages, namely operation software and visualization software. The operation software package consists of the client side control center component which serves as a user interface for simulation initialization and manipulation at runtime. The package visualization software comprises the client side operation center which contributes the visualization of the simulation state and its objects attribute values. Moreover, this package contains a Google Earth component for both visualization and persistent storage in KML.

The visualization components are described in further detail within the next chapter.

4.5 Visualization

The following section explains the interaction points of the user with the simulation. The section begins with the description of the Simulation Control Center that influences the simulation lifecycle. Subsequently, the operation center that shows the sensors of the simulation during runtime is presented.

4.5.1 Control Center

The Simulation Control Center constitutes the starting point of the simulation. The web application is password protected and designed to serve one user at a time to prevent unwanted false administration. The simulation initialization includes the definition of start parameters and general features, such as the simulation time, the example city and weather parameters, and the initial sensor distribution as well as technical features of the distributed architecture. The user input is validated by the system, summarized for the user and may be saved persistently before the input is to be confirmed. During runtime, the control center provides event-driven manipulation of the simulation lifecycle including weather events, variation of sensor quantity and so forth. These functionalities are located in an altered view of the control center. The system dynamically presents the correct view to the user. Figure 8 illustrates the expected appearance of the control center at simulation initialization.

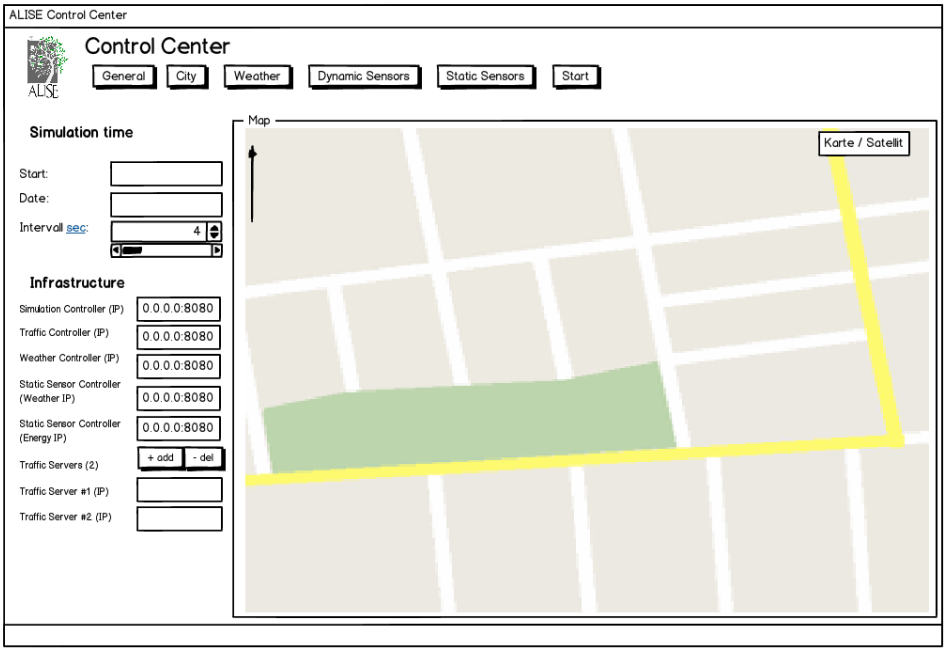


Figure 8: Mock-up of the Simulation Control Center

4.5.2 Operation Center

The operation center represents an end point of the simulation. First and foremost it is a visualization of the sensor distribution on a map. Furthermore, it is possible to show detailed information about the simulated objects. As a consequence, the operation center is linked to the simulation itself as well as to the data stream tool. Take note that the user of the operation center has no influence on the simulation

itself but perceives a defined part of the simulation aspect. For that reason, the operation center is meant for multi-user usage.

The user can freely alter the visualization of the sensors. For instance, the user may or may not visualize a certain amount of simulated cars, certain bus lines, static sensors or anything else by choosing from a list of available sensors. Large quantities of sensors can be observed individually or via a heat map. The values of the visualized sensor data update automatically or user-driven. For the purpose of a good user experience, the user can define favored sensors and save the operation center preferences persistently. Figure 9 outlines the planned composition of the operation center.

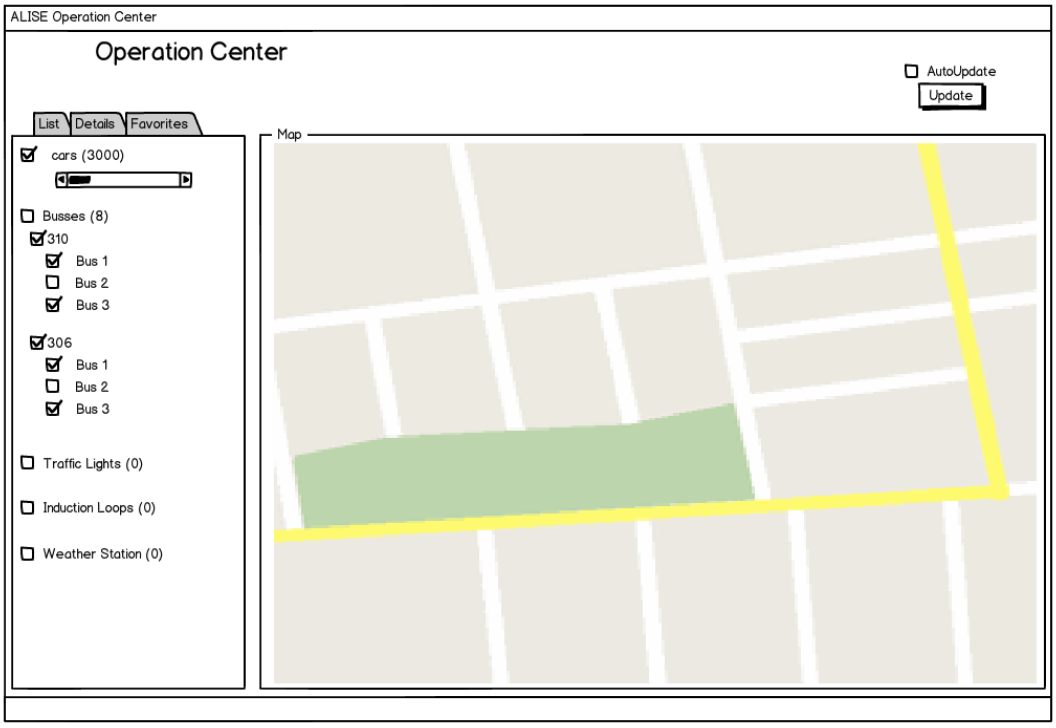


Figure 9: Mock-up of the operation center

4.6 Simulation

This section specifies the package simulation of the general architecture (see Figure 7). Therefore the individual components of the simulation are explained. These components are the Generic Controller, the Simulation Controller, the Weather Controller, the Static Sensor Controller, the Traffic Controller, the Traffic Server and the Traffic Sensor Controller.

4.6.1 Generic Controller

The Generic Controller defines the interface, which contains the basic methods for handling sensors. The following classes use the interface: Traffic Controller, Traffic Server, Static Sensor Controller and the Simulation Controller. To implement a state pattern for the other controllers the interface inherits the methods of the State Controller (see Figure 10). The State Controller determines the methods for a state pattern. Additional own methods are for creating and deleting sensors as well as requesting the sensor status.

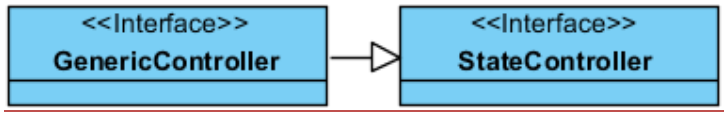


Figure 10: Scheme of the Generic Controller

4.6.2 Simulation Controller

The Simulation Controller interface inherits the basic sensor methods from the Generic Controller and gets implemented by the SimulationControllerImpl class (see Figure 11). It implements the methods of the Generic Controller and passes the sensors on to the related controller. Furthermore the Simulation Controller defines methods for adding and deleting events, adjusting the timestamp, initializing just as starting and stopping the simulation.

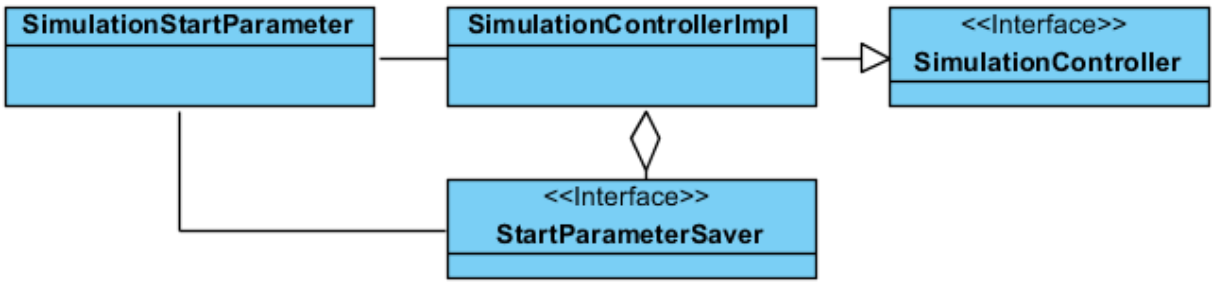


Figure 11: Scheme of the Simulation Controller

4.6.3 Weather Controller

The Weather Controller interface inherits as well as the other controllers the State Controller to implement the state pattern (see Figure 12). It defines a method to access weather data from weather stations and weather services and a method to start and restart the controller. Unlike the other controllers, the Weather Controller doesn't handle sensors. Its main purpose is to provide access to the weather data on the basis of a key, timestamp and position.

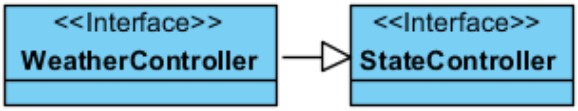


Figure 12: Scheme of the Weather Controller

4.6.4 Static Sensor Controller

The Static Sensor Controller interface defines methods to handle weather and energy sensors (Figure 13). Its methods are starting and restarting the controller.

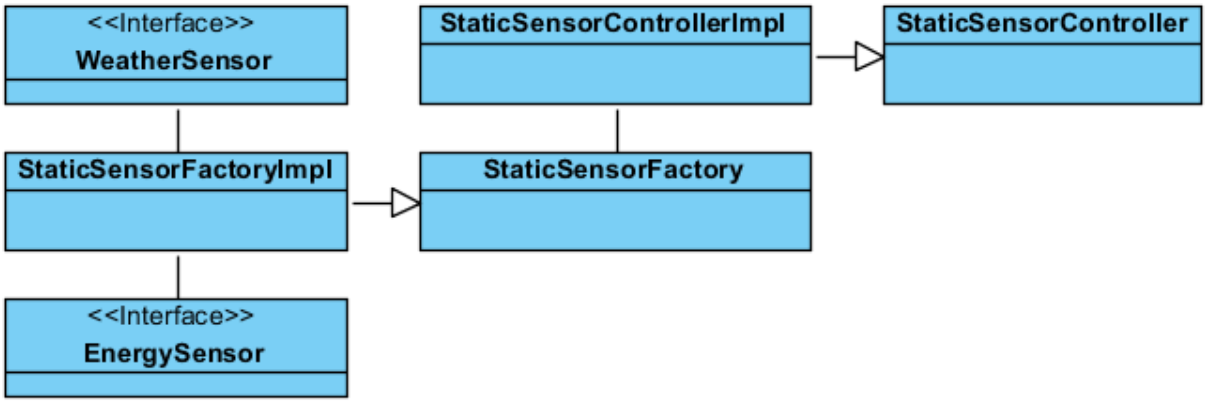


Figure 13: Scheme of the Static Sensor Controller

4.6.5 Traffic Controller

This Traffic Controller interface defines methods for starting the controller, parsing home and work nodes, creating trips and registering the server. Unlike the other controllers the Traffic Controller divides the created graph into a certain number of chunks afterwards they get distributed onto the given number of registered servers. The following diagram shows the scheme of the Traffic Controller.

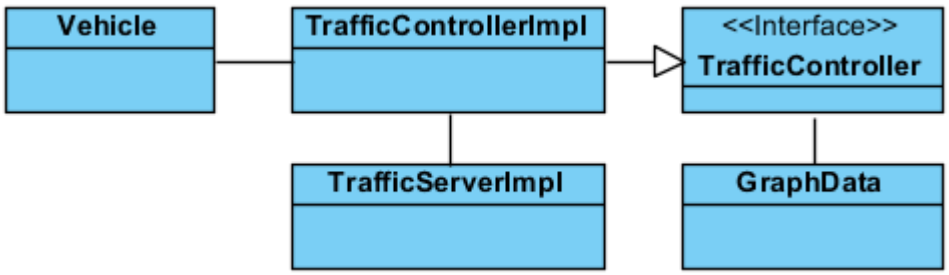


Figure 14: Scheme of the Traffic Controller

4.6.6 Traffic Server

The Traffic Server interface contains methods, which provide the vehicle functionalities applied for the simulation (see Figure 15). These functionalities allocate creating cars/busses and deleting as well as updating the vehicle positions. In addition, the interface provides a method for the vehicle's current status.

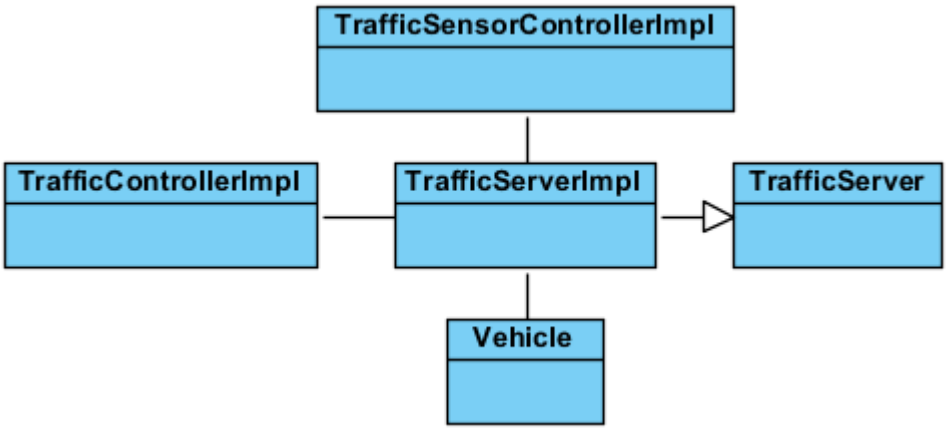


Figure 15: Scheme of the Traffic Server

4.6.7 Traffic Sensor Controller

The Traffic Sensor Controller interface is conceived for starting, stopping and restarting the used sensors. On the contrary to the Traffic Server the Traffic Sensor Controller takes care of the sensors currently used in the simulation. These sensors are infrared-, induction loop- and GPS sensors (see Figure 16).

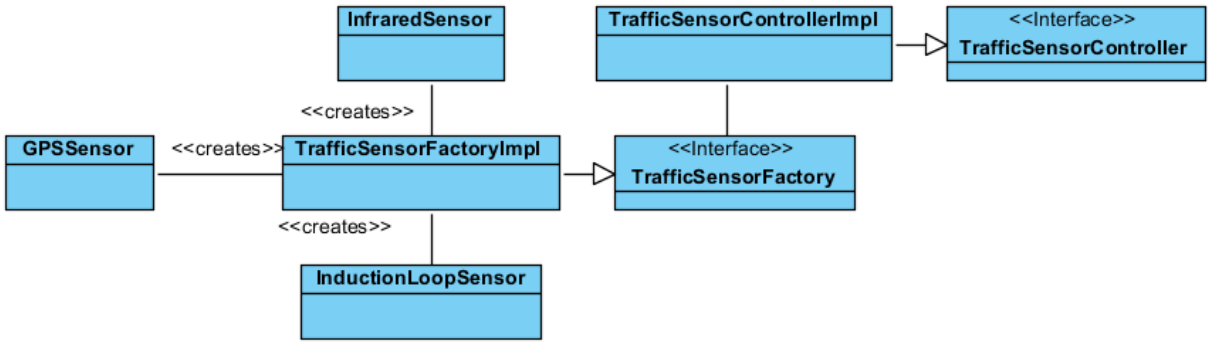


Figure 16: Scheme of the Traffic Sensor Controller

4.7 Sensors

The following part describes the various sensor types of the simulation. At first a short description is given. Than the different sensor moldings and the expected sensor data are declared. After that a transfer to the project is demonstrated and at the end possible independencies between the sensor types and simulation are given.

4.7.1 Meteorological Sensors

4.7.1.1 Description

Meteorological sensors measure the current weather from their environment. The sensors are equipped with several measuring instruments that are able to record some metrological events.

4.7.1.2 Sensor molding

There are two options to create those sensors:

1. There is a weather station that includes some several measuring instruments and returns comprehensive data about the weather.
2. There is only one measuring instrument in a sensor. It can return just a specific value.

4.7.1.3 Sensor data

Table 4: Sensor data for weather

Indicator	Description	Unit	Measuring instrument
Atmosphere			
Air pressure	Air pressure is the force per unit area exerted into a surface by the weight of air above that surface in the	Hectopascal (hPa)	Barometer

	atmosphere of Earth.		
Humidity	The amount of steam in the air.	relative: % absolute: g/m^3	Hygrometer and others
Temperature	Physical ratio for the warmth of an object.	Celsius, Fahrenheit	Thermometer
Radiation			
Global radiation	The amount of the direct and indirect radiation.	Watt-hour per square meter (Wh/m^2)	Pyranometer
Duration of sunshine	The total number of the direct duration of sunshine.	Watt-hour per square meter (Wh/m^2)	Sunshine autograph
Light intensity	Illumination of a light source.	Lux	Luxmeter
Air motion			
Wind speed	The way that the wind needs to move in a defined time.	Meter per second (m/s)	Hemispherical cup anemometer
Wind force	Classification from the wind force.	Beaufort (Bft.)	Beaufort scale
Wind direction	Direction of the wind.	degree	Wind vane
Clouds and rainfall			
Rainfall	The amount of water that rains down on the earth's surface.	Millimeter (mm)	Rain gauge
Precipitation amount	The amount of rainfall water that is collect in a cup for a defined time.	Liter per square meter (l/m^2)	Rain gauge
Cloudiness	The amount of clouds at the sky.	Eighth scale	Guess

Probability of rainfall	Forecast for a rainfall at a special point.	Percent	Forecast
Visibility	Describes the biggest distance from a visible point to another.	Kilometer (km)	Assessment, transmissiometer

4.7.1.4 Transfer to the project group

All the weather sensors have a common context, which they have to share. One implementation opportunity is a root node that delivers initial values to other sensors. The root node is the only one who is listening to its environment. Another opportunity is that all sensors listen to the environment. The information about the weather will be calculated by the environment dependence on the position of the sensors.

The realization of the meteorological sensors follows three phases:

1. The setup of a data basis with historic and current data from weather stations[5]. It is possible to divert average value.
2. An extension of the data basis by using the forecast and information from a weather service. It is possible to use reference parameters for various cities to match these forecasts with city of the simulation.
3. Implementation of a simplified simulation. For this phase it is necessary to include the data basis from the phase one and two. Afterwards regularities of the city climate, day and season and meteorological sizes, weather proverbs and weather phenomena are added incrementally.

4.7.1.5 Dependencies to other sensors

The meteorological sensors strongly influence all other sensors. In the following a few examples are listed:

- Weather phenomena influence other sensors like energy sensors.
- Duration of sunshine and light intensity are important for the data generation from the photovoltaic.
- The traffic can be influenced by the visibility and rainfall.
- Wind force and direction are the basis for a wind energy system.
- The traffic can influence the energy consumption.

[5] A possible source for data of weather stations are here:http://www.uni-oldenburg.de/dezernat4/wetter/

4.7.2 Local public transport sensors

4.7.2.1 Description

Local public transport sensors are divided in two main categories: sensors to record the position of an object and sensors that measure the capacity for busses and trams. Sensors that record the position of an object are based on fixed spots (beacon) or on moving objects like cars, busses or undergrounds. Also the sensors include loading sensors who count the passengers in the busses.

4.7.2.2 Sensor molding

We distinguish two areas between the sensors:

- The first type of sensors determines the position of a vehicle. It can be an IR-equipped sensor, GPS and beacon.
- The second type of sensors counts the passengers in the vehicle. IR-equipped, step sensors and others can be used.

4.7.2.3 Sensor data

The only standard format is given from the GPS sensors. All the other sensors (see section Sensor molding) do not have a popular format. The visualization of the position data is based on geographical data (degree of longitude and latitude). The capacity visualization is based on value of an integer.

The GPS sensors have typically the following format:

```
$GPGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,M,<10>,M,<11>,<12>,*<13><C
R><LF>
```

An example of this format can be this:

```
$GPGGA,104549.04,2447.2038,N,12100.4990,E,1,06,01.7,0078.8,M,0016.3,M
,,*5C<CR><LF>
```

The format of IR-equipped sensors for the example of counting passengers are only numeric.

4.7.2.4 Transfer to the project group

It is necessary to simulate the bus route, the bus stop and a timetable by loading the schedule from a bus company[6] to transfer a public transport network map. Afterwards it is possible to generate the bus routes depending on GPS data. In connection to this it is possible to generate data of the capacity. After that it is imaginable to integrate other bus routes by using GTFS.

4.7.2.5 Dependencies to other sensors

The only dependence is given by the private transport. The local public transport is even influenced by the weather sensors. The recorded weather can influence the data of the capacity sensors and the position sensors.

[6] The schedules of the VWG of Oldenburg can be used.

4.7.3 Private Vehicle Traffic

4.7.3.1 Description

The sensors of the private vehicle traffic have the main function to catch the vehicles. In that context the sensors produce data like the number of cars that passed a point or the average speed of vehicles.

4.7.3.2 Sensor molding

There are a lot of different sensor moldings:

- Induction loop
- Magnetic field sensor
- Infrared-, radar-, laser- und ultrasonic sound sensor
- Touch sensor
- Video detector
- Acoustic sensor
- Weight in motion system

4.7.3.3 Sensor data

Through the differences between the unequal sensors, the calculated data might be different. The sensors, which are counting vehicles, contain an integer who represents an aggregated count of vehicles as a measured value. Induction loops besides return more data than the number of vehicles. They produce the average speed of the vehicles or the average of holding time as well. These data are not relevant for the project group yet.

4.7.3.4 Transfer to the project group

First, it is adequate to simulate stationary sensors for the project group. The project group should aim a focus on induction loops, because those sensors are predominant in practice. Induction loops send the number of the vehicles in a defined period.

There is no standard format for the traffic sensor in practice. But you can see a trend to create new standards in Europe by using OCIT (Open Communication Interface for Road Traffic Control Systems) and in the USA by using NTCIP (National Transportation Communications for Intelligent Transportation System Protocol).

Simulated vehicles can be realized as mobile sensors or GPS-sensors by sending continuous positions of themselves.

To sum up the project group realizes two different sensors (vehicles as a GPS-sensor and as an Induction loop) for private vehicle traffic. The first step of the implementation procedure will be to create a passive sensor, who listens to the car-passing event. This event will be created in a simple

way. The next step is to create a sensor with geographic position. The first simulation will be the counting of the vehicles.

4.7.3.5 Dependencies to other sensors

The number of vehicles depends on the weather (e. g. more people use public transports when the weather is rainy) and on the local public transport (e. g. when more people use the local public transport, there will be less vehicles on the street).

4.7.4 GPS sensors / mobile sensor

4.7.4.1 Description

Normally a GPS sensor sends a GPS location. The GPS location consists of the latitude and longitude of the earth model. It is also possible that the altitude is included in the GPS location. There are different coordination systems available for the latitude and longitude.

4.7.4.2 Sensor molding

A sensor of this type has a GPS mouse, a logger, a GPS plus navigation, a navigation system, precision instruments and military receivers. The GPS can use different coordination systems.

4.7.4.3 Sensor data

The typical formats of the sensor data are the NMEA0183 and GPX standard format. It is also common to send only the GPS coordinates.

An example of the NMEA0183 format is following:

```
$GPRMC,162614,A,5230.5900,N,01322.3900,E,10.0,90.0,131006,1.2,E,A*13
$GPRMC,HHMMSS,A,BBBB.BBBB,b,LLLLL.LLLL,l,GG.G,RR.R,DDMMYY,M.M,m,F*PP
```

An example of the GPX format is following:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpxxmlns="http://www.topografix.com/GPX/1/1" creator="byHand"
version="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">
<wptlat="39.921055008" lon="3.054223107">
                    <ele>12.863281</ele>
                    <time>2005-05-16T11:49:06Z</time>
                    <name>CalaSantVicenç - Mallorca</name>
                    <sym>City</sym>
               </wpt>
</gpx>
```

4.7.4.4 Transfer to the project group

GPS sensors will be implemented in buses and cars. They are used as mobile sensors.

4.7.4.5 Dependencies to other sensors

The simulation environment and especially the weather influence the GPS sensors. There dependencies can cause errors of the global position system. Possible GPS errors are:

- Atmospheric errors influenced by weather
- Ionospheric error due to solar winds
- Multipath errors due to urban climate
- Ephemeris errors (possibly related to solar wind)
- GPS-itself (the behavior of moving objects)
- Weather, other events, road network, …

4.7.5 Sensors in the energy sector

4.7.5.1 Description

These Sensors measure the energy flow from various facilities and devices over time. The following show two examples of this sensor type: If the sensor is a smart meter in a household, the power consumed over a period of time should be recorded. If it is a sensor of a plant, the power generated in a given period should be recorded.

4.7.5.2 Sensor molding

There are different sensor types:

- Sensors for wind turbines,
- Sensors of photovoltaic systems,
- Sensors of hydroelectric power plants or
- Sensors in Smart Meters

4.7.5.3 Sensor data

The transmitted data depend on the particular sensor. In general the amount of energy transferred per time unit is transmitted.

4.7.5.4 Transfer to the project group

The project group restrict themselves to the renewable energy generation, in particular sensors for wind turbines and photovoltaic systems. Therefore the first implementation procedures will be to implement sensors for photovoltaic systems (based on the meteorological data basis) and sensors for wind turbine (based on the basis of meteorological data).

4.7.5.5 Dependencies to other sensors

The sensors of the renewable energy generators have a strong dependency on the weather because the amount of energy produced is based on the weather. For example the photovoltaic systems depend on

the sunlight: it generates more energy if the sunlight is stronger. The wind turbine depends on the wind: it generates more energy if the weather is windy.

4.8 Management of persistent Data

Due to the fact that the stream processing software should handle the storage of historical data as well as the simulation should use realistic weather information, a management of persistent data is required. Therefore, this section explains the different storage mechanisms.

At first, a description of the technical infrastructure is given. Afterwards, different kinds of persistent data with their general structures are underlined. In particular, the needed weather information, sensor data, measured values and stored start parameters are presented.

4.8.1 Technical Infrastructure

The technical infrastructure is represented by several virtual servers on a machine sponsored by the OFFIS.

- Windows Server 2008 R2 Enterprise (x64) with *IBM Cognos in version 10.1.1* and *IBM DB2 in version 9.7.0.441*. The machine has an Intel Xeon q2.40 GHz and 4 GB RAM.
- CentOS release 5.8 (Final) Server with *IBM InfoSphere Streams in version 2.0.0.4* and an *IBM DB2 version 9.7*.
- CentOS release 5.8 (Final) Server with a Jetty Servlet Container in version *Stable 8.1.7.v20120910* and *Java SE Development Kit 7u7*. This server is for the *operation center* and *simulationcontrol center*.
- CentOS release 5.8 (Final) Server with *openEJB* in version *4.0.0* and *Java SE Development Kit 7u7*. This server is for the all the Enterprise Java Beans. Later we plan to get more servers and divide the controllers.

4.8.2 Persistent Data

The smart city application needs different kinds of persistent data. In this project the management of persistent data is divided in three main areas: Weather Data, Simulation Parameter and the Sensor Data.

4.8.2.1 Weather Data

Keeping in mind that the application does not want to forecast future weather conditions or simulate weather information itself, a basis of weather data is required. Indeed, the simulation uses two sources for weather information.

First, a reference weather station gives the fundament of realistic value curves. Due to the fact that the project group has chosen Oldenburg as a starting point, it makes sense to use data from a weather station in Oldenburg. An example of the weather station can be the one from the Carl von Ossietzky

University of Oldenburg[7]. Further reasons for this choice are the free use of data, the many provided measure values and the easy access to the data.

The subsection "Meteorological Sensors" has shown different kinds of measuring instruments. To realize a lot of instruments it is required to have access to the following measure values: Air pressure, light intensity, precipitation amount, temperature (optionally perceived temperature), radiation, relative humidity, wind direction and wind velocity. Another function of the smart city application is to use aggregate or exact weather data. The following entity relationship model (see Figure 17) illustrates the scheme of persistent weather station data to satisfy all the mentioned functions.

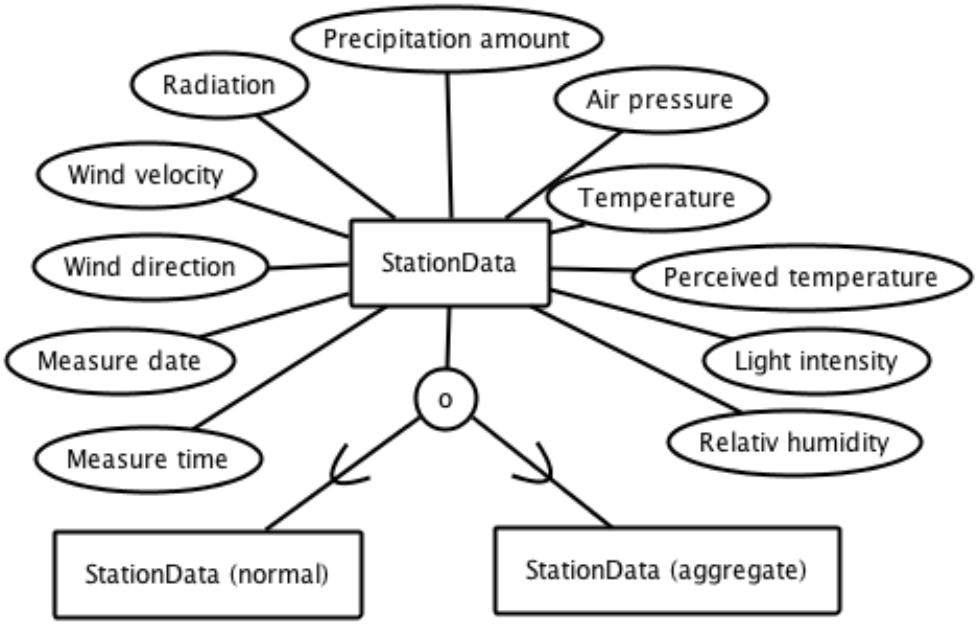


Figure 17: Schema of weather station data

The various measure values, measure time and measure date belongs to the entity *StationData*. The special *forms StationData (normal)* and *StationData (aggregate)* allow the storage of aggregate and exact weather information.

The second source of weather information is data from different weather services. These services provide data to simulate other city climates and regions. The scheme of weather service data is shown in the following entity relationship model (see Figure 18). The entity *ServiceData (current)* stores the current measure values temperature, relative humidity, wind direction, wind velocity and visibility. The choice of these values is based on the offer of free weather services[8]. Besides, the measure date and measure time as well as sunrise and sunset are linked to that entity. The entity *ServiceData (forecast)* with the attribute *Measure date* represents future weather forecasts. The attributes *Temperature low* and *Temperature high* belong to the forecasts. Every instance of the entities

[7] http://www.uni-oldenburg.de/dezernat4/wetter/
[8] For instance, the weather services from Google or Yahoo offer this weather information.

ServiceData (current) or *ServiceData (forecast)* is assigned to a city and is described by a condition. The Entity *City* has the following attributes: *Name*, *Country*, *Altitude* and *Population*. Other possible information can be added to the city. For instance the attributes can be *Near sea* or *Near river*. The entity Condition has the two attributes *Code* and *Description*. The conditions codes are the numerical explanation[9].

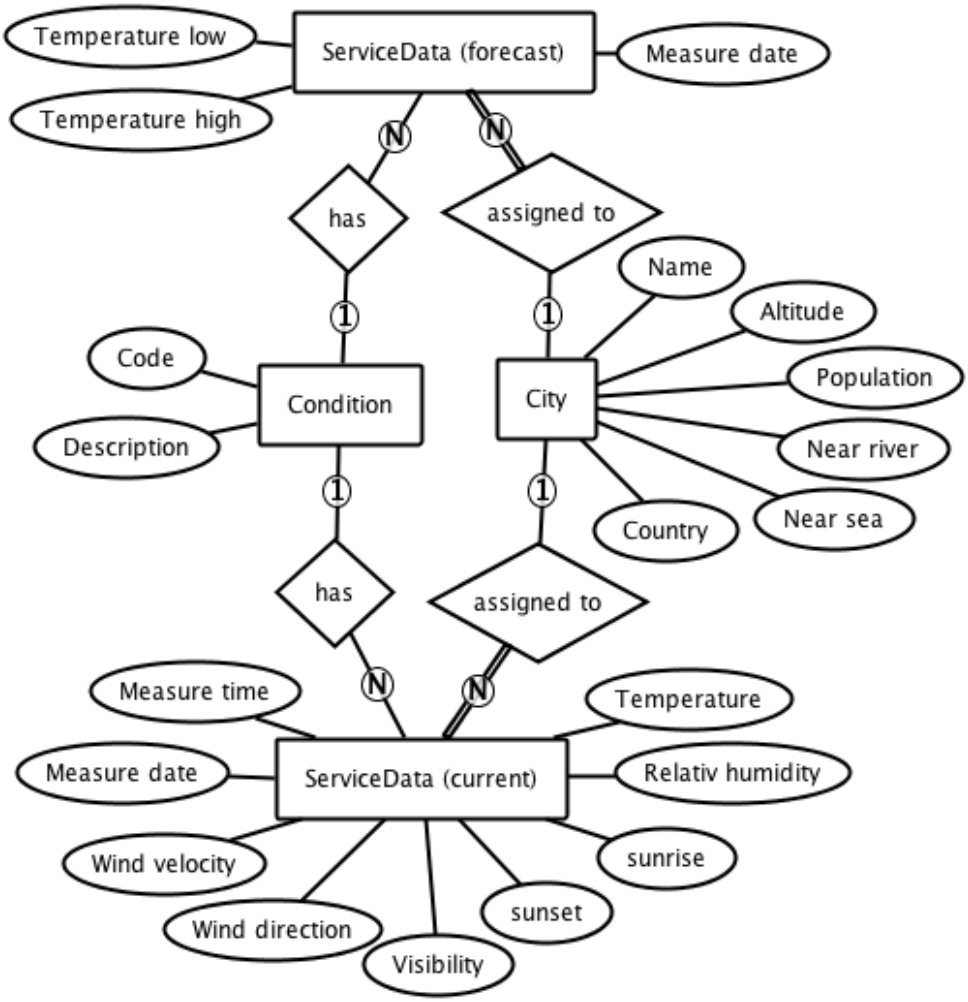


Figure 18: Schema of weather service data

[9] The different condition codes can be found under the following url: http://developer.yahoo.com/weather/#codes

4.8.2.2 Simulation Parameter

Before starting a simulation a multitude of parameters have to be set by the user. Figure 19 shows what simulation parameters can be configured and saved to a XML File.

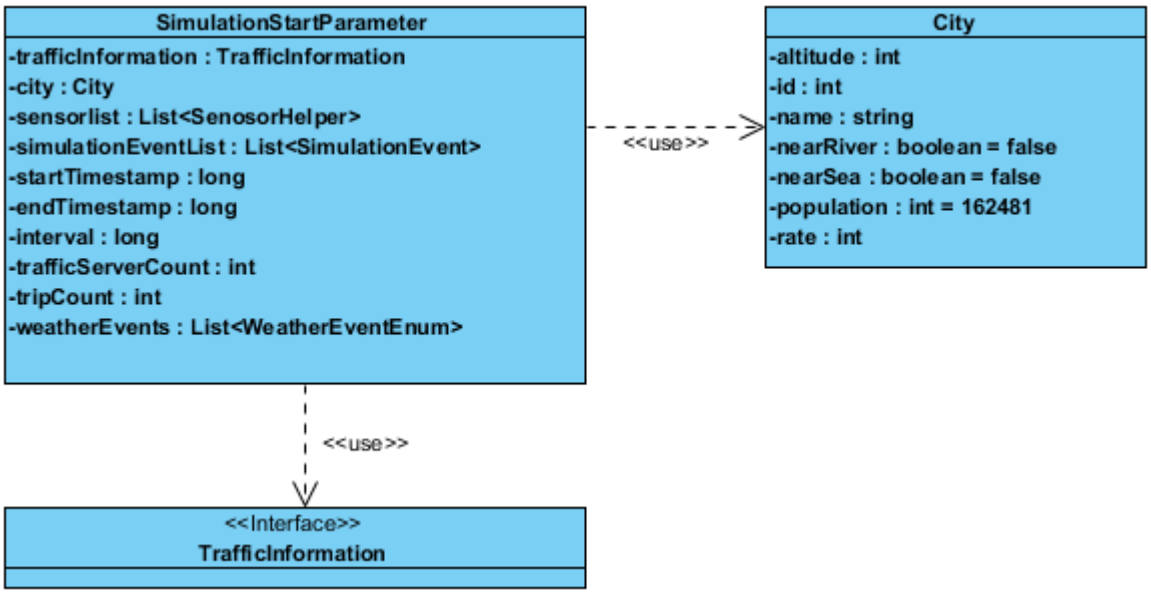


Figure 19: Schema of the SimulationStartParameter

The *SimulationStartParameter* contains *trafficInformation* for the Traffic Controller. The simulated *City* is described by an altitude, a name, a population, a rate and properties whether the city is near a river or near sea.

4.8.2.3 Sensor Data

The *Sensorlist* includes a list of sensors that are available at the beginning of a simulation. The *startTimestamp* and *endTimestamp* define the period of simulated time. Interval describes the length of a simulation step in milliseconds. The *TrafficServerCount* specifies the number of Traffic Servers in the simulation. *TripCount* shows the number of trips simulated. *WeatherEvents* contain all events that influence the weather of the simulation.

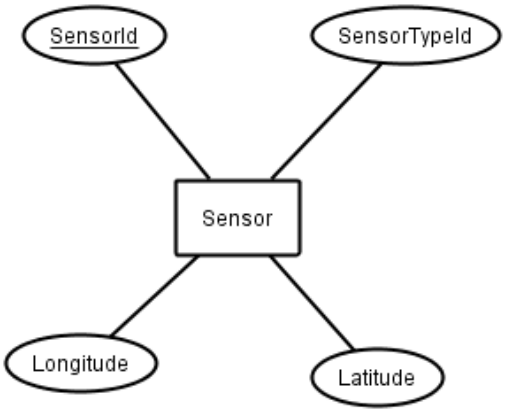


Figure 20: Schema of the sensors

To gain the best possible performance the sensor's scheme for persistence is kept simple (see Figure 20). A *sensor* can be identified by its *SensorId* represented by an integer value. Furthermore each sensor has a SensorTypeId because the fact that each of them belongs to a special class of sensor. Also geolocation data is included in the scheme to find sensors easily during the simulation. It has to be said that the geolocation data might contain null values (i.e. GPS Sensors).

The simulation creates large amounts of sensor data. The *Sensor* component creates a data streams out of the started simulation, which will be send via TCP to InfoSphere Streams. The first data stream contains the ID of the sensor, the ID of the sensor type, a timestamp and the measure value. While processing the sensor data in InfoSphere Streams, all of this data will be saved persistently in a database tuple by tuple and also aggregated (see Figure 45). After all the output data stream will be send via TCP, which can be used by the reporting components. Further Output mechanisms of IBM InfoSphere Streams are explained in Source-Interfaces InfoSphere Streams.

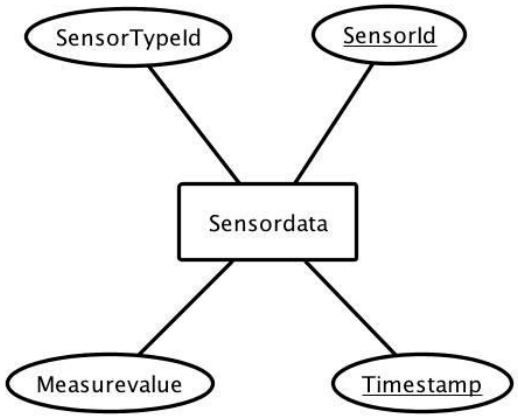


Figure 21 Schema of the Sensor data

4.9 Exception Handling

The system's exceptions can be divided into two types of exceptions. In the first case wrong user input can cause exceptions thrown in the system. These are the simplest exceptions to handle. They are displayed to the user by the Simulation Control Center and the user is able to change his decision. Each transaction caused by this kind of wrong input has to be undone by the system to keep it in a consistence state.

The other kind of exceptions that may occur are caused by hardware errors - for instance a connection loss of one of the servers. In such special cases the system attempts to stop the simulation and informs the user about the malicious exception. In order to rerun the simulation a restart by an administrator is necessary.

4.10 Source-Interfaces InfoSphere Streams

The following section explains the different sources of IBM InfoSphere Streams that can be used in this project. Moreover examples to the different sources are given. At the end the supported data types of IBM InfoSphere Streams are mentioned.

4.10.1 TCPSource - Recommended

The TCPSource operator reads data from a TCP socket and creates tuples out of it. It can be configured as a TCP server (listens for a client connection) or as a TCP client (initiates a connection to a server). In both modes it handles a single connection at a time. It works with both IPv4 and IPv6 addresses. The operator has no input port and a single output port.

Format: txt, csv, line, binary, block

```
stream  <rstring name, uint32 age> Person = TCPSource() {
    param
        role                : server;
        port                : 12345u;
        reconnectionPolicy  : InfiniteRetry;
}
```

Figure 22: Example of TCPSource

Important parameters: address, port, format and maybe the role (server or client)

4.10.2 UDPSource

The UDPSource operator reads data from a UDP socket and creates tuples out of it. Each tuple must fit into a single UDP packet and a single UDP packet must contain only a single tuple. There are no input ports and only a single output port.

Format: txt, csv, line, binary, block

```
stream <rstring name, uint32 age> Person = UDPSource() {
    param
        port                : 12345u;
        address             : "some.host";
        receiveBufferSize   : 10240u;
        initDelay           : 5.0;
}
```

Figure 23: Example of UDPSource

Important parameters: address, port, format and maybe the role (server or client)

4.10.3 ODBCSource – Recommended for enrichment or historical data

The ODBCSource operator generates a stream from the result set of an SQL SELECT statement. The SELECT statement is specified as the value of the query attribute of the query element of the access specification named by the access parameter. For each row in the result set, the ODBCSource operator produces a tuple by automatically assigning the values of the columns of the result set to the output stream attributes with the same name and data type. When a column of the result set contains null data, the corresponding output stream attribute is set to 0 for numeric data type and to the empty string for the String data type.

```
streamLowStream ( id: Integer, item: String, quantity: Integer )
      :=ODBCSource() [
            connectionDocument: "configuration.xml";
            connection: "InventoryDB";
            access: "LowInventory";
            initDelay: 3] {}
```

The corresponding statement in the XML-document:

```
<access_specification name="LowInventory">
<query query="SELECT id,item,quantity FROM inventory WHERE quantity
<5"/>
<external_schema>
      <attribute name="id" type="Integer" />
      <attribute name="item" type="String" length="32" />
      <attribute name="quantity" type="Integer" />
</external_schema>
</access_specification>
```

Supported products and drivers of IBM InfoSphere Streams are the following:

Database Product	Version	Supported Driver
DB2 Runtime Client and DB2 Client	9.7	DB2 ODBC
IBM Data Server Client and IBM Data Server Runtime Client	9.5	IBM Data Server ODBC
IBM Informix Dynamic Server	11.50	IBM Informix ODBC
Oracle Database	11*g* Release 2	UnixODBC
IBM solidDB	6.5	UnixODBC
MySQL	5.1	UnixODBC
Microsoft SQL Server	2008	UnixODBC
Netezza	6.0	UnixODBC

Figure 24: Supported products and drivers of IBM InfoSphere Streams

4.10.4 InetSource

The InetSource Operator generates output Tuples from one or more text-based data feeds accessible via the http, https, ftp, or file protocols. The basic operation is to fetch data from one or more URIs at a given interval and output either the entire content or only the new content added since the last fetch. The InetSource Operator does not attempt to parse the data in any way, and the output stream schema may define only one Attribute of type String or StringList. Parsing can be done by a downstream Operator, such as a Functor.

```
streamWxObservations ( metarObsRecord: String )
      :=InetSource ()
      [ URIList:
      "http://noaa.gov/pub/data/observations/cycles/07Z.TXT",
      "http:// noaa.gov/pub/data/observations/cycles/08Z.TXT";
initDelay: 5;
incrementalFetch;
fetchIntervalSeconds: 60
] {}
```

Supported source types are: Http, Https, RSS via HTTP, RSS via HTTPS, FTP, FTPS, File

4.10.5 Supported Data Types

The given Figure 25 shows the supported Data Types of IBM InfoSphere Streams.

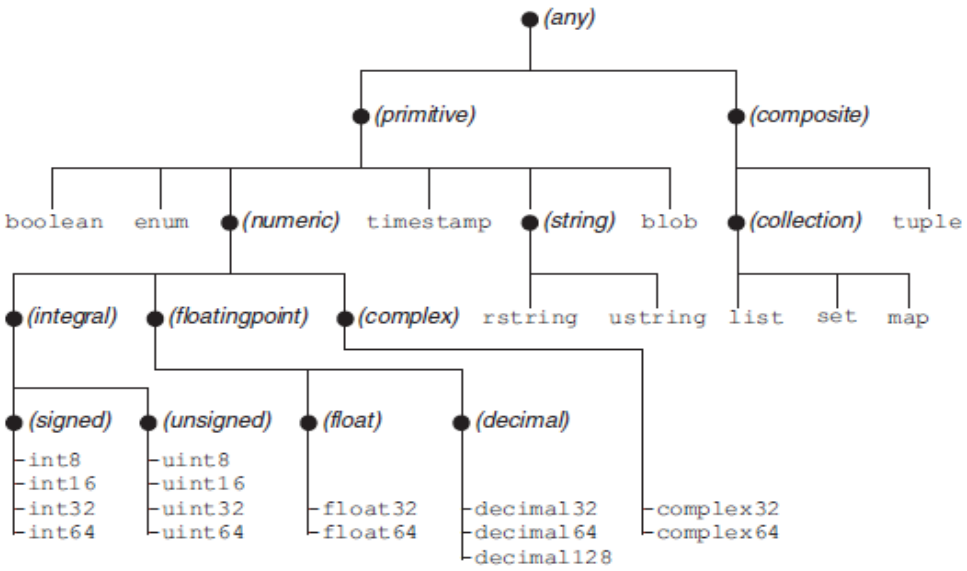


Figure 25: Supported Data Types of IBM InfoSphere Streams

5 Implementation

This section explains the final realized architecture of the software at first. The changes to the architecture, illustrated in the design document on page 25, are highlighted. After that, a description of the individual components follows.

5.1 Implemented Architecture

The implemented architecture has not changed much. JavaEE is used instead of SOA to realize the architecture. JavaEE covers servletsm JSP, JPA and EJB-modules. SOA turned out to create a big overhead in the first part of the implementation progress. Another reason for this was the statefulness of our controllers. In SOA all services are stateless. The general concept of using controllers for the different dimesions was implemented as planed. As a consequence of using JavaEE some additional controllers and services, e.g. FrontController and ServiceDictionary, were added to the architecture. Thereby it was possible to use these services in a distributed architecture. Figure 26 shows the implemented architecture.

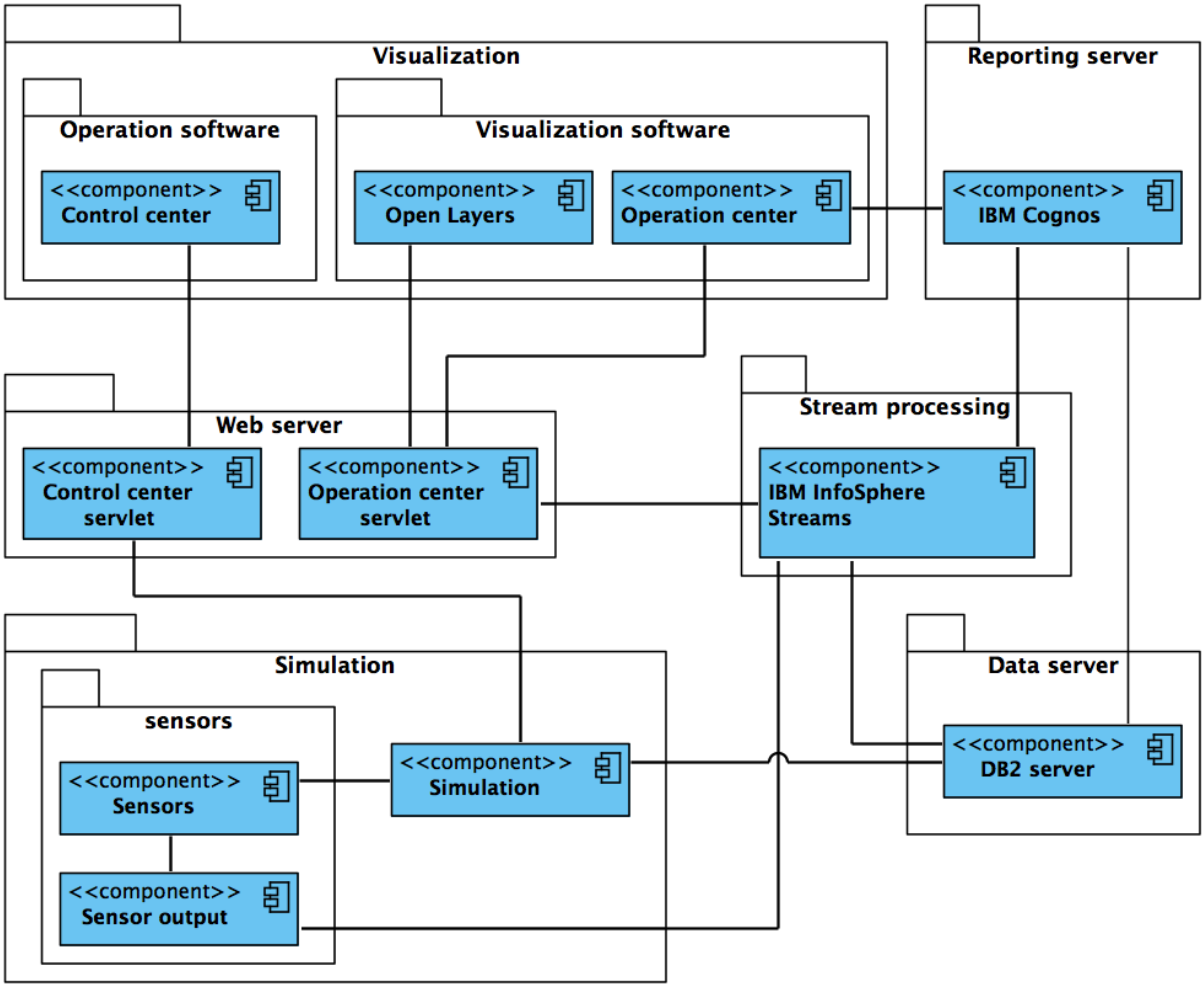


Figure 26: Overview of the implemented architecture

Further JPA was used to connect and interacte with databases. The Operation Center and the Simulation Control Center were implemented with servlets and JPS for WebApps. All created EJB-modules are services that can be utilized from third party applications. TomEE as an EJB-container is used to deploy these services. A detailed descriptions of each component follows in the next sections.

5.2 Description of all Components

In the next step the different components of the simulation are introduced in alphabetical order. Thereby a description of the general function, an UML diagram of the scheme, the configuration, the extension points and the exception handling of the different components follow.

5.2.1 Component: Simulation Control Center

The simulation control center is the starting point for the user. One can configure the simulation via the simulation control center's web interface. It is possible to define the simulation time, city, weather, vehicles, static sensors and events.

The simulation control center has two basic states. For specifying the simulation parameters, the initialization state serves as the basis for creating simulation start parameters as such. After the simulation was started, that is, during runtime, the runtime state is active in the simulation control center. As a result, the respective view components change and the user can now interact with the simulation. The current simulation time is pushed into the simulation control center for user experience purposes. However, sensor data is dealt with in the operation center.

The web application is comprised of a Java Servlet and a Java Script Client, which communicate through WebSockets. The communication is message based, that is, there are predefined types of messages, each including a type, a unique message id for callback purposes and some content as payload. The interpretation of the content is dependent on the message type.

The following message types (including denoted <content type>) exist within the simulation control center:

- *CCWebSocketMessage<T extends Object>*: Super class of all messages. All messages include message type, id and generic content.
- *AccessDeniedMessage<String>*: Sent by server if access was denied, for instance, if more than one user wants to connect to the simulation control center. In this case, the UI is getting disabled.
- *AskForValidNodeMessage<Node>*: Sent by client if a user adds a sensor to a specific position on the map. This position has to be validated in order to match is with the existing graph representation of the street network. The *Node* parameter represents such position which may not be conform to the graph representation within the simulation. On the server, the nearest node to the received node is calculated and then sent to the client in form of a

ValidNodeMessage.

- *ValidNodeMessage<Node>*: Sent by server if a Node was validated and can now be sent to the client. The respective simulation object that was previously created by the user is updated concerning the position.
- *CreateAttractionEventsMessage<Collection<AttractionData>>*: Sent by client if one or more attractions shall be pushed into the simulation at runtime. *AttractionData* includes data about the amount of cars, bikes, motorcycles and trucks concerning the attraction, as well as their GPS ratio, attraction position, start and end time, etc.
- *CreateRandomVehiclesMessage<RandomVehicleBundle>*: Sent by client if random sensors shall be pushed into the simulation at runtime. *RandomVehicleBundle*includes data about the amount of vehicles and their GPS ratio.
- *CreateSensorsMessage<Collection<SensorHelper>>*: Sent by client if one or more sensors are pushed into the simulation at runtime. *SensorHelper* represents such sensors.
- *DeleteSensorsMessage<Collection<SensorHelper>>*: Sent by client if one or more sensors shall be deleted from the simulation at runtime. *SensorHelper* represents such sensors.
- *ErrorMessage<ErrorMessageData>*: Sent by either client or server if some error occurred.
- *GenericNotificationMessage<String>*: Sent by server if some process finished successfully.
- *ImportXMLStartParameterMessage<String>*: Sent by client if XML file is to be imported. The String type parameter represents the XML file.
- *LoadSimulationStartParameterMessage<String>*: Sent by client if a previously exported or started scenario is to be loaded. The String type message represents the name of the scenario.
- *OnConnectMessage<OnConnectParameter>*: Sent by server if the connection is established. The *onConnectParameters* include
    - *osmResourcesList*: a list of available OSM files,
    - *busstopResourcesList*: a list of available GTFS files,
    - *sensorTypeMap*: a mapping of sensor type numeral and type name in enumeration,[10]
    - *weatherEventTypeMap*: such mapping for weather events,
    - *simulationEventTypeMap*: such mapping for events,
    - *vehicleTypeMap:* such mapping for vehicle types,
    - *vehicleModelMap*: such mapping for vehicle models,

[10]Note that GSON needs the String-representation, that is the enumeration entry name, for parsing components including Enums correctly.

- *vehicleTypeRandomModelMap*: mapping between vehicles models and types,
- *savedStartParameterList*: list of saved start parameters,
- *busRouteList*: list of available bus routes.

- *OSMAndBusstopFileMessage<OSMAndBusstopFileData>*: Sent by server on startup, including a list of available OSM and GTFS files.
- *OSMParsedMessage<Boundary>*: Sent by server if the OSM file is parsed, including the extracted boundaries of the scenario on the map.
- *SimulationEventListMessage<SimulationEventList>*: Sent by client if events shall be pushed into the simulation at runtime.
- *SimulationExportStartParameterMessage<SaveStartParameterData>*: Sent by client if start parameters shall be exported to XML format. The content includes the start parameters and the desired filename.
- *SimulationExportedMessage<String>*: Sent by server if start parameter export was successful. Content includes the name of the file which can then be downloaded by the user.
- *SimulationStartedMessage<IDWrapper>*: Sent by server if the simulation was started. Content includes a list of assigned ids. These IDs will be pushed into the id service to prevent id collisions.
- *SimulationStopMessage<Void>*: Sent by client if the simulation is to be stopped.
- *SimulationStoppedMessage<Void>*: Sent by server if the simulation was stopped.
- *SimulationUpdateMessage<Long>*: Sent by server is a new simulation step is finished. The content includes the current simulation time in milliseconds.
- *UsedIDsMessage<IDWrapper>*: Sent by server if one or more ids were forgiven. These IDs will be pushed into the id service to prevent collisions.

5.2.1.1 Scheme

Server-side architecture:

The simulation control center does not provide an API, because it is completely exchangeable.
The following UML class diagrams show the server-side architecture of the control center with its classes and packages and their interaction points. Only the important method signatures are shown. The parameter and return types are hidden. Every diagram is about one package of this component.

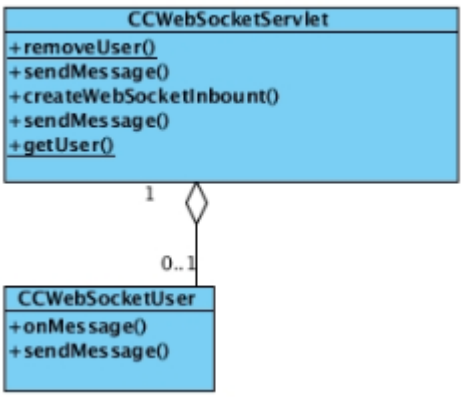


Figure 27: Simulation Control Center Servlet

The components *CCWebSocketServlet* and *CCWebSocketUser* are the cardinal point of the simulation control center.They are shown in the diagram Figure 27: Simulation Control Center Servlet *CCWebSocketServlet* can create an instance of *CCWebSocketUser*, which represents the client user on server-side. The CCWebSocketServlet receives important information from the *SimulationController*, e.g., if the simulation is stopped, caused by an error, and gives all the information to the *CCWebSocketUser*. The communication of between client and server is done by messages that can be sent or received via the WebSocket protocol. Possible messages are shown in Figure 28. The messages can influence the simulation. E.g., the user can start a new simulation or stop it. These messages will be serialized and deserialized via GSON from a Java instance into JSON or from JSON into a Java instance. Messages from the user can be received and handled in the method *CCWebSocketUser.onMessage()*. If a message needs to be sent, the *CCWebSocketUser.sendMessage()* method can be used. Every message from the client will be response with a message from the server. To perform asynchronous call on the client-side, every message has a message ID. This ID is the same as in the request message. E.g. the *AskForValidNodeMessage* with message ID 3 will be response with a *ValideNodeMessage* with message ID 3.

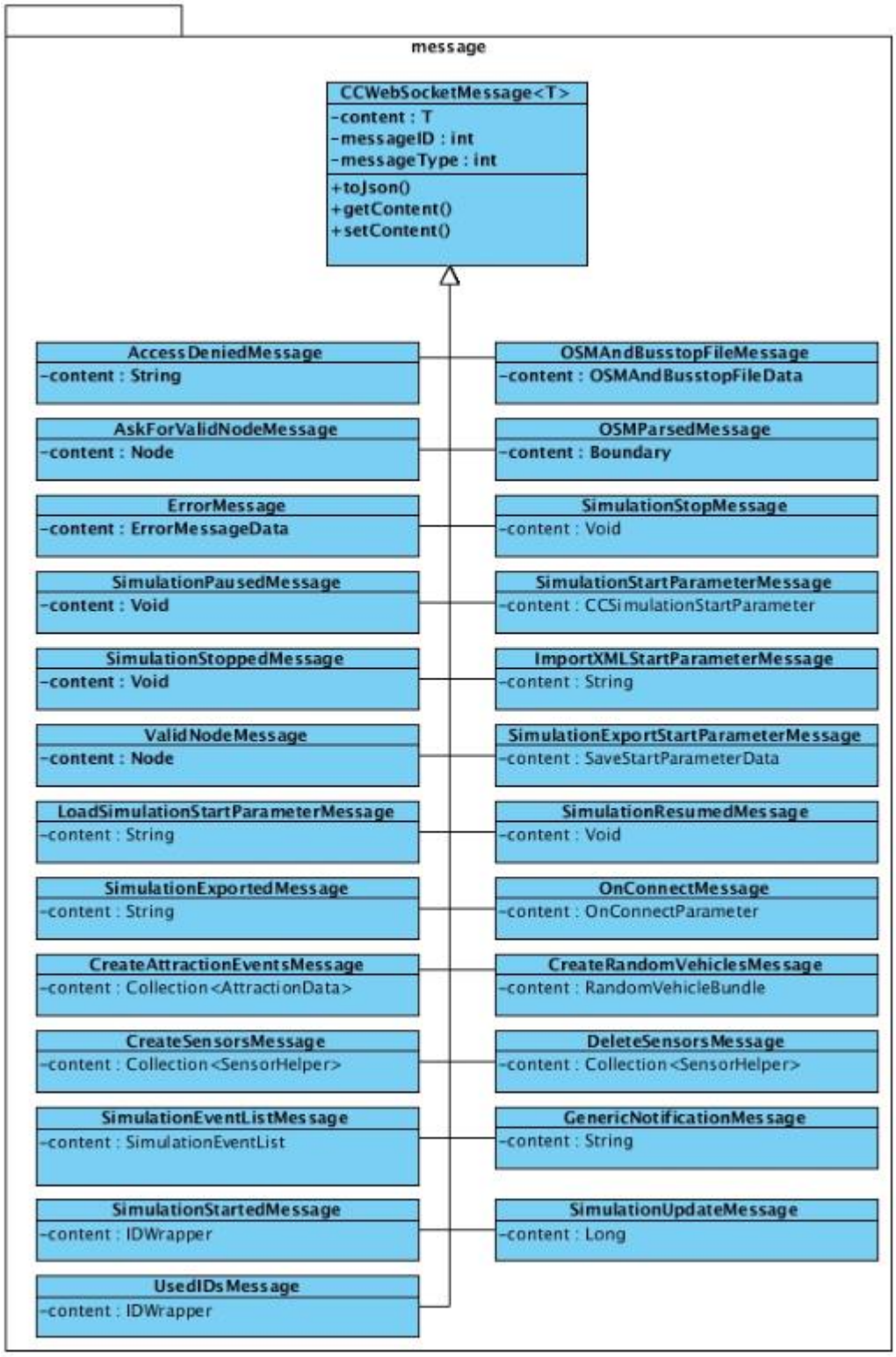


Figure 28: Simulation Control Center Messages

To handle all the messages and to avoid unnecessary network traffic, the simulation control center has own models. They are shown in Figure 28. E.g., the *CCSimulationStartParameter* is a class only for the simulation control center. The server creates the *InitParameter* and *StartParameter* for the simulation with this information and sends it to the SimulationController.

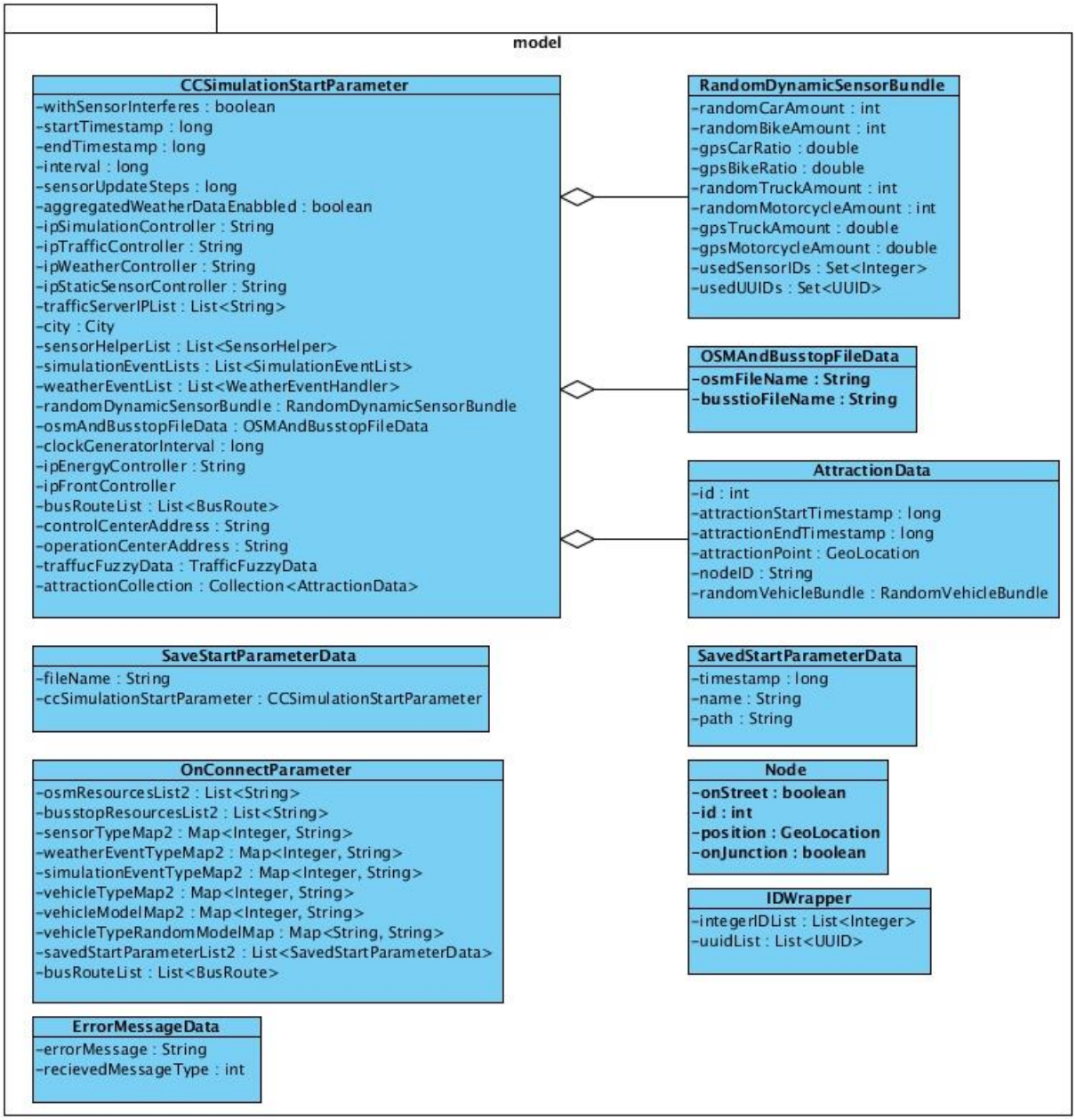


Figure 29: Simulation Control Center Model

The *CCWebSocketUser* also uses some services to handle the messages. They are shown in Figure 30.

- **ControlCenterModel**: The class ControlCenterModel contains all the used bindings.
- **CreateAttractionEventService**: The service CreateAttractionEventService can create attraction events from *AttractionData* instances.
- **SensorHelperWithInterfererInstantiationService**: With the SensorHelperWithInterfererInstantiationService it is possible to create *SensorHelper* instances with the needed *SensorInterferers*.
- **OSMCityInfrastructureDataService**: To avoid long waiting time and reparsing the OpenStreetMap files on every start, the OSMCityInfrastructure will parse an OpenStreetMap and bus stop file only one time and will save it binary to allow faster loading.

- **StartParameterSerializerService**: To provide different ways of saving the start parameter, the StartParameterSerializerService can be used. The service can also deserialize already exported files.

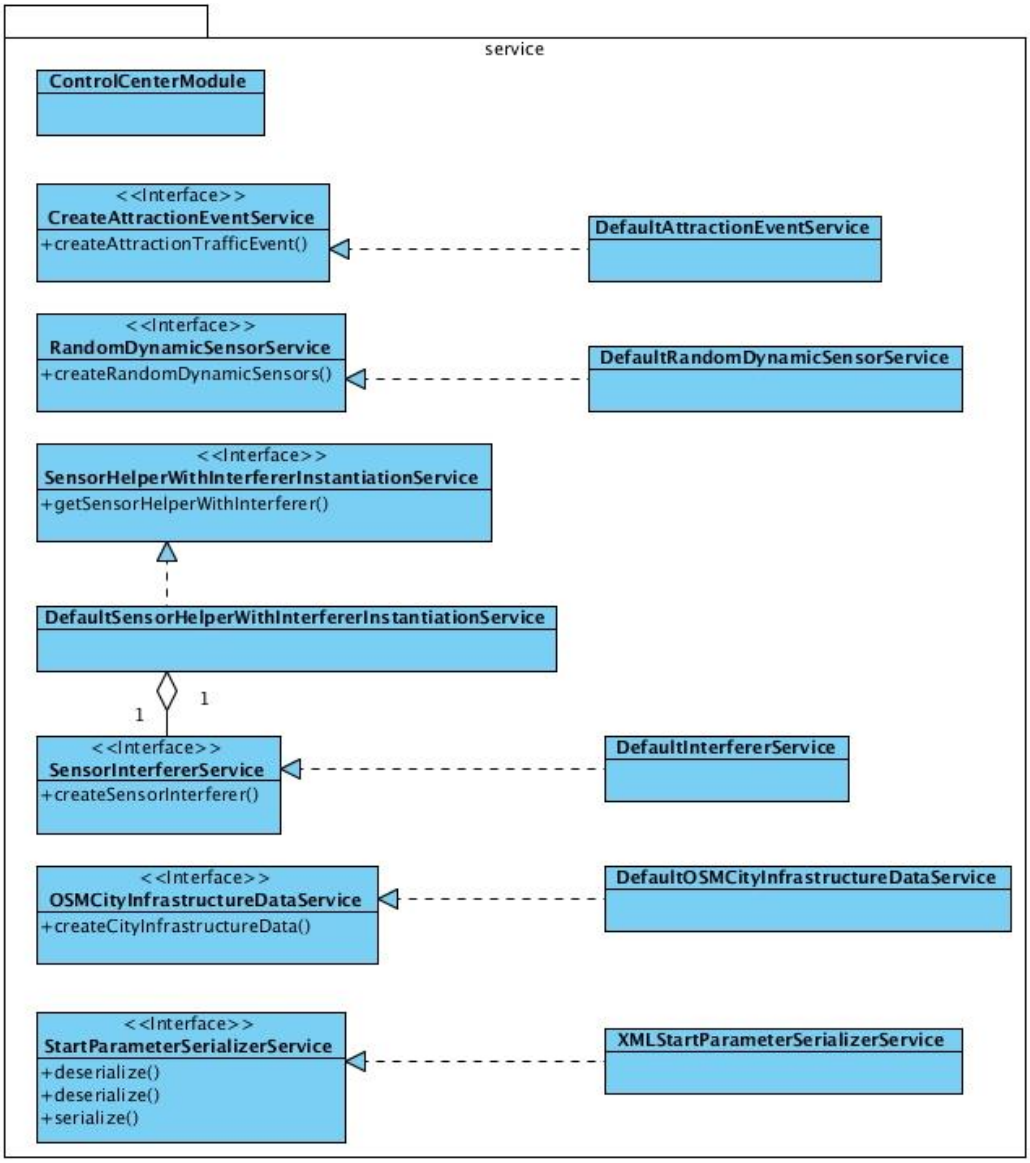


Figure 30: Simulation Control Center Service

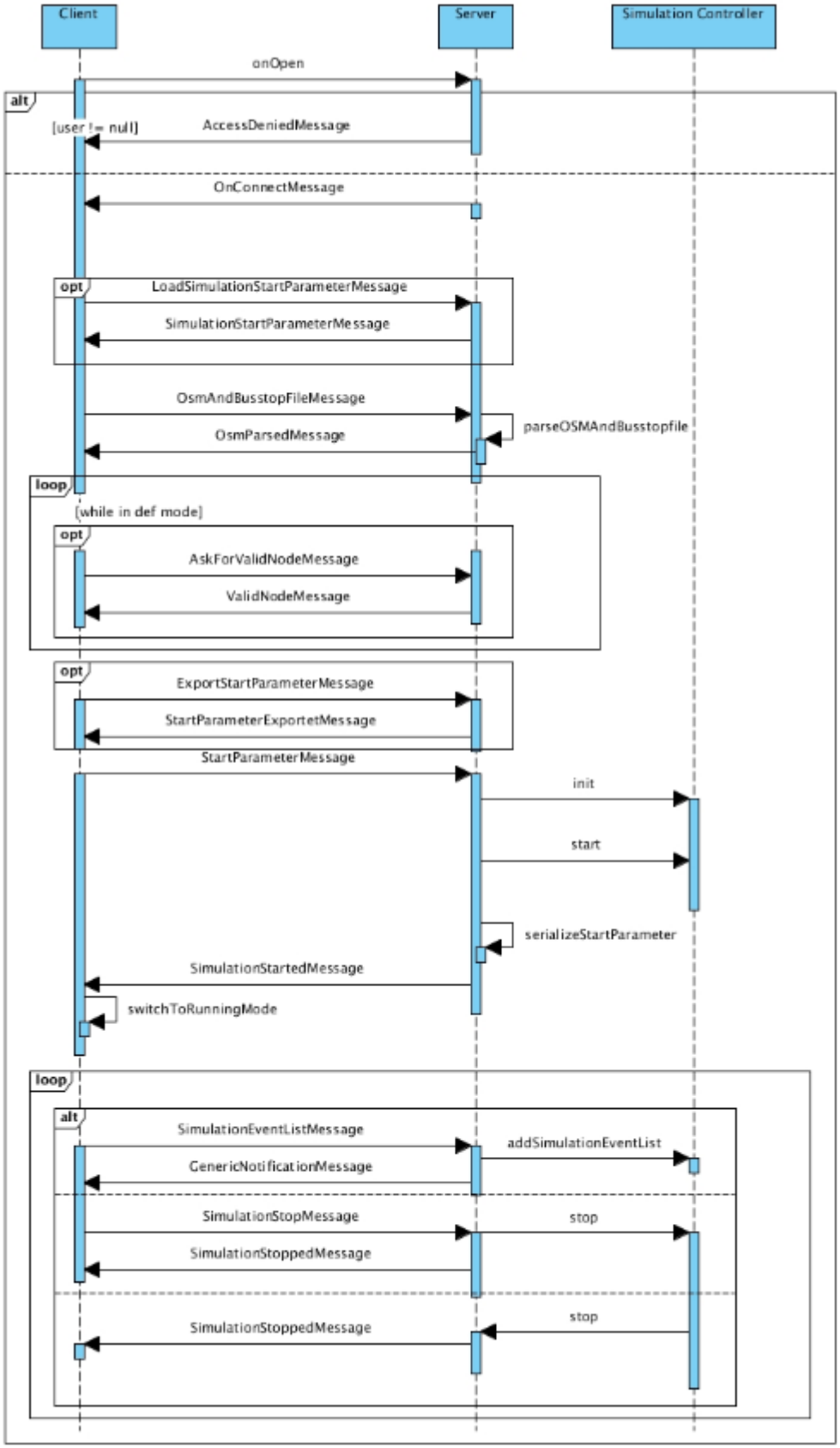


Figure 31: Control Center Sequence Diagram

To display the flow of the simulation control center, Figure 31 shows a sequence diagram of it.

- After the user entered the web interface, the server will proof if there is already a user online. If yes, it will send the *AccessDeniedMessage*, if not, it will send the *OnConnectMessage*.
- If the user wants, he or she can load previous start parameters with the *LoadSimulationStartParameterMessage*. The Server will load the parameters and responses with a *SimulationStartParameterMessage*.
- The user needs to send an *OsmAndBustopFileMessage* to inform the server that a specified OpenStreetMap and busstop file shall be loaded. An OpenStreetMap is necessary for parsing the street network and the home and work areas of the city. The busstop file is a list of busstops with name, id and position. An example of the GTFS file is in the appendix 29. After the server parsed the files, it will response with an *OsmParsedMessage*.
- Now the user is in the definition mode and can set the simulation settings. He or she can ask the server for valid nodes on the graph with the *AskForValidNodeMessage*. The nodes will be validated and answered with a *ValidNodeMessage*.
- When the user finished the settings he can export the start parameter by sending an ExportStartParameterMessage. The server will serialize the start parameter and send a link to them in the *StartParameterExportetMessage*.
- The user can start the simulation by sending the *StartParameterMessage*. The server will process the message and init and start the *SimulationController*. After this, it will send the *SimulationStartedMessage*.
- The client switches into the running mode.
- *SimulationEventListMessages* can be sent in the running mode. They will be given to the SimulationController and answered with a *GenericNotificationMessage*.
- The simulation can also be stopped by the user. He can send the *SimulationStopMessage*. The server will call stop on the SimulationController and answered with a *SimulationStoppedMessage*.
- Because the simulation has a specified time window, it is also possible that the time if over. The *SimulationController* will call stop on the server and the server will inform the user with a *SimulationStoppedMessage*.

Client-side architecture:

The client side is made up of view components, controller components, model components and some modules, including services. The following class diagram depicts the view and controller components, which are described below. The relationships between views and controllers are depicted. Note that

each controller or view component exists in one single HTML or Java Script file. For that reason, the files are represented in the diagram.

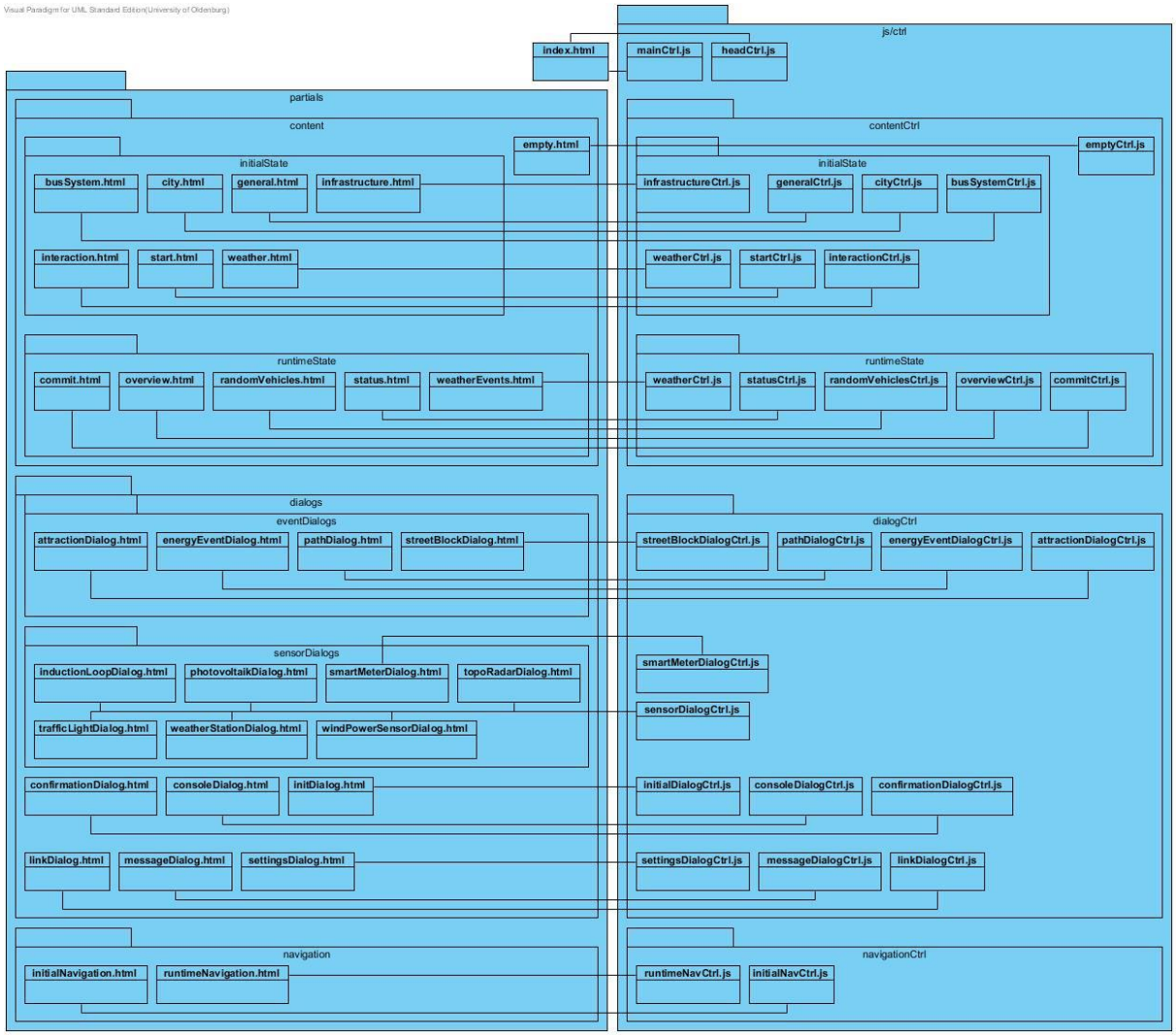


Figure 32: View and Controller Components of the Simulation Control Center

The control center is a single side web application, which means, that there is only one HTML file which is loaded at the beginning. Later interactions are WebSocket based. The *body* tag in the *index.html* includes a *ng-app* attribute which binds the view components to its controller components. Specifically, view components are bound to each other with the *ng-controller* tag. *js/modules/modules.js* serves as the starting point of the application. All components are loaded with the HTML but initialized by AngularJS.

View components:

View components are HTML files in *webapp/partials* directory. The *ng-model* tags represent the respective controller, as described above. Each controller has a scope for properties which can be altered by the controller and presented by the view components. There are three main folders for the view components:

- *content* includes the content view components, that is, the *index.html* serves the main site of the application. In this site, the central area is disposable, as there are several contents which will be represented in this area depending on the current user navigation. The HTML files describing these main content areas are deposited in the *content* folder. The folder is subdivided into two folders, namely *initialState* and *runtimeState*, which include the content views for both states of the application – before and during runtime.
- *dialogs* includes the view components for the dialogs that are available in the simulation control center. In fact, every single sensor type is related to one dialog type that will be shown when the sensor is selected on the map. These view components are deposited in the subfolders *eventDialogs* and *sensorDialogs*.
- *navigation* includes the view components for the navigation area of the web application. As described above, there are two basic states of the UI which differ in the navigation view component which is presented to the user. The HTML files describing these navigation view components are present in the *navigation* folder.

Controller components:

Each view is controlled by its controller. The controller components are deposited in the *js/ctrl* folder. For reasons of clarity and comprehensibility, there are two subfolders *contentCtrl* and *dialogCtrl*.

- *contentCtrl* includes all controllers for the content view components. There are two subfolders *initialState* and *runtimeState*, which makes the assignment of view-controller-pairs even simpler.
- *dialogCtrl* includes all controllers for dialog view components.
- *navigationCtrl* includes two controllers for the two navigation view components.
- Furthermore, there are two controllers, which are not present in a subfolder.
    - *headCtrl* is a controller for the head area of the view.
    - *mainCtrl* is the main controller of the simulation control center. Every controller can use the main controller's scope, as it is attached to the *body* tag in the DOM. Consequently, every function of the simulation control center has its starting point in the main controller, that is, communication or simulation services will call functions in the main controller, which then calls functions in the appropriate components.

Model components:

The model components encapsulate the models for the client side of the simulation control center in Java Script objects. The following diagram depicts the model components in a class diagram. Note that the Java Script files serve as a container for multiple classes.

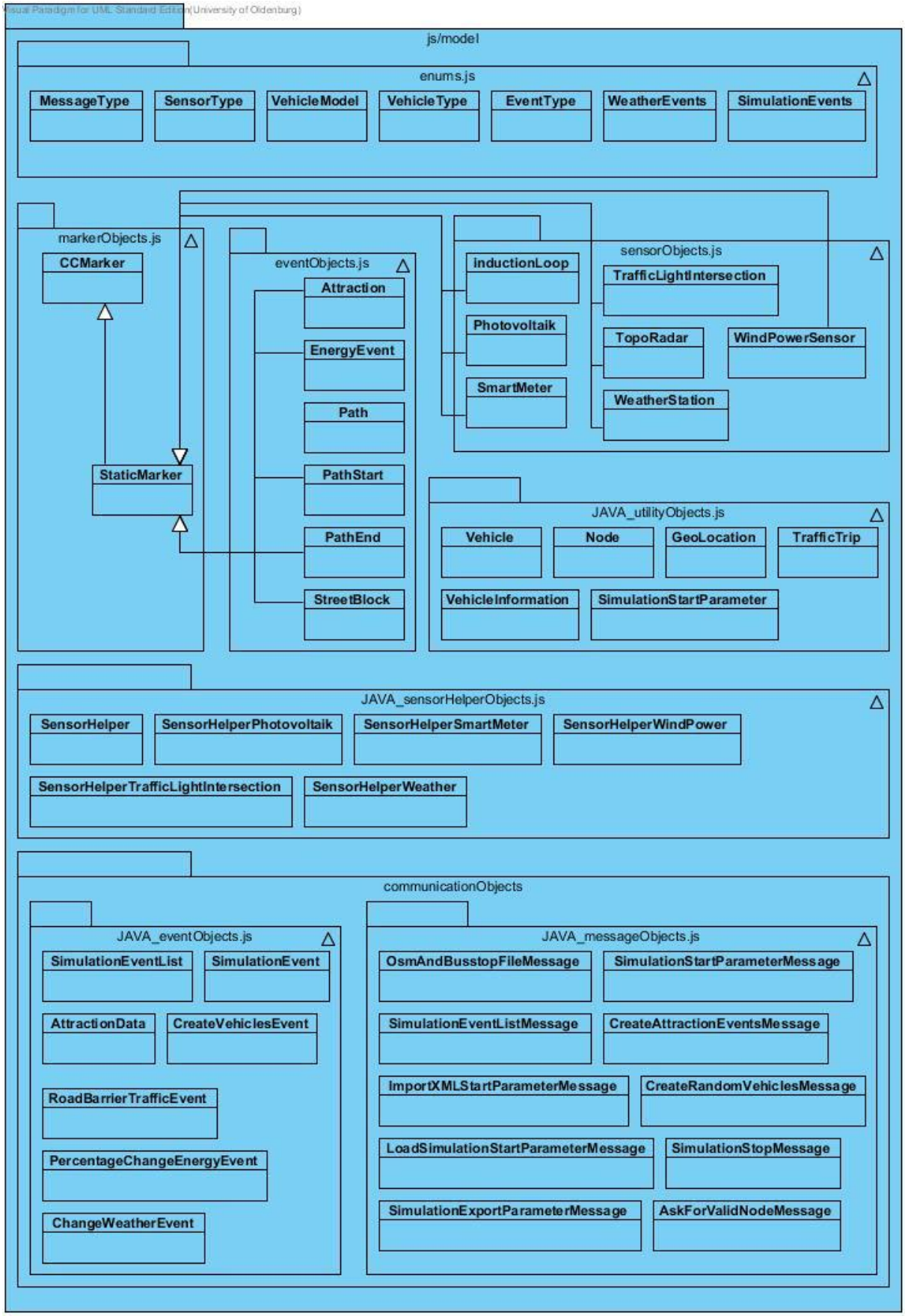


Figure 33: Model Components of the Simulation Control Center

- *enums.js* includes all the enumerations of the operation center on the client side. Those are stored in the global variables
    - *model.MessageType*,
    - *model.SensorType*,
    - *model.VehicleModel*,
    - *model.VehicleType*,
    - *model.EventType*,
    - *model.WeatherEvents*and
    - *model.SimulationEvents*.

  Note that there is no automatic dependency resolving mechanism between Java components and Java Script client components. As a result, every alteration in components which are represented in a Java Script equivalent must be carried out on the client components as well.
- *markerObjects.js* contains the top level markers that are drawn onto the Open Layers map. *StaticMarker* extends *OCMarker* and inherits its properties to all objects which are shown on the map.
- *sensorObjects.js* contains the sensors that are shown on the map. All sensors extend *StaticMarker*.
- *eventObjects.js* contains the events that are shown on the map. All events extend *StaticMarker*.
- *JAVA_sensorHelperObjects.js* contains Java Script equivalents of Java classes.
- *JAVA_utilityObjects.js* includes Java Script equivalents of Java classes. The most important object of these is the *SimulationStartParameter* object, which is the product of all user activities before runtime, that is, it represents the simulation parameters before runtime. Furthermore, this object is the basis of the import and export functionality of the simulation control center.
- The folder *communicationObjects* containts three Java Script files, each including Java Script equivalents of Java classes that are directly sent to the server.
    - *JAVA_messageObjects.js* contains Java Script equivalents of complex messages that are sent to the server.
    - *JAVA_eventObjects.js* contains Java Script equivalents of complex simulation events that are sent to the server.

Note that the objects in the files starting with *JAVA_* are equivalents of Java classes on the server side. These classes are sent to the server, parsed by GSON and, in most cases, directly pushed into the simulation. As a result, altering those Java classes on the server side must involve a modification of the relevant client side code in Java Script.

Angular modules:

The angular models are provided In the *webapp/js/modules* directory. The following class diagram depicts the modules. Important relationships, functions and properties are depicted.

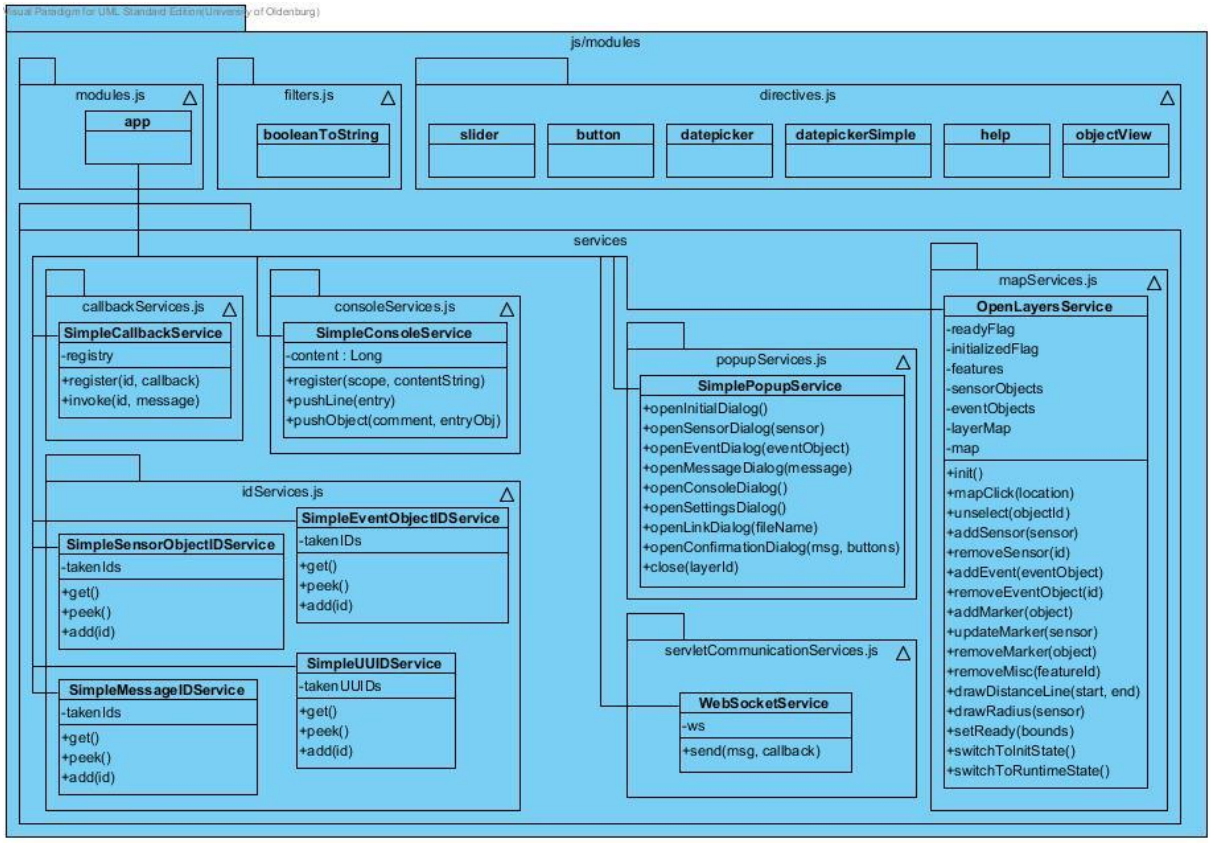


Figure 34: Module Components of the Simulation Control Center

- *modules.js* is the Java Script starting point of the application. In *index.html*, there is an attribute *ng-app="app"*, which binds the body tag to the controller components. This is where the dependency injection is carried out, that is, an injectable service like *MapService*is bound to a concrete implementation like *OpenLayersService*.
- *directives.js* contains some AngularJS directives, which are bound to HTML elements. These directives serve a constructor function which will be invoked when the respective HTML element is added to the DOM tree. Those constructor functions are used to call JQuery UI functions and constructors, as well as, for initialization of the component.
- *filters.js* can contains the *booleanToString* filter, which parses a Boolean expression into a string.
- *services* directory contains all the service modules including the implementations for the services. Currently, there are just simple implementations for the services. However, more complex or alternative implementations can be added here. Dependency injection is then conducted in *modules.js*. The following services are available:
    - *callbackServices.js* contains a simple callback service, including a method for registering functions with message ids as callbacks. When a server side message with

the same message id is received, the stored function will be called. As a result, simple server callbacks can be registered and invoked, which enhances both code readability and performance.

- *consoleServices.js* contains a simple console service, including methods for registering at the service and pushing individual lines or complex objects into the console. Components can register their scope and the name of an array object within this scope. As a result, when some content is pushed into the console, the console service redirects the content to all registered components and applies the respective views. When a complex object is pushed into the console, an intended string representation of the object will be printed to the console.
- *idServices.js* contains a simple implementation of an id service which returns an incrementing integer number. For each service, the used IDs are stored and only unique IDs are created. If IDs were created on the server side or in another component, those can be added to the ID service in order to prevent ID collisions. There are four services available:
  - *SimpleSensorIDService*: Returns increasing integer.
  - *SimpleEventIDService*: Returns increasing integer.
  - *SimpleMessageIDService*: Returns increasing integer.
  - *SimpleUUIDService*: Returns a UUID starting with 00000000-0000-0000-0000-000000000001 and increasing.
- *mapServices* contains an Open Layers implementation of the map service. The service provides methods for showing objects on a map and storage of such objects.
- *popupServices.js* contains a implementation of a popup service. The service can be injected in every controller and, as a result, a popup can be invoked and presented easily. Popups include a so-called *layerId*, which represents a popups logical layer in the user interface. The popup closing method is based upon these layers. As a result, more than one popup can be organized on the same layer and closed in chorus. Alternatively, popups can be organized on different layers.
- *servletCommunicationServices.js* serves as a service for communication with the WebSocket component on the server side. The service has two main purposes:
  - React on incoming messages: For this purpose, the client side component binds functions to the *onopen*, *onmessage* and *onclose* variables of the WebSocket object. Messages are analyzed with regard to their message type and then sent to the main controller, assuming that there is no registered in the callback service. In this case, the callback is invoked and the message is by then processed.

- Send messages to server: For this purpose, there is a *send(msg, callback)* method. The message is parsed into JSON and then sent to the server. If the callback parameter is set, a new callback is registered at the callback service.

Global Java Script objects:

The *js* folder itself contains two Java Script files:

- *global.js* contains the global variable of the application, including the packages *ctrl*, *model* and *util*.
- *util.js* contains a number of utility functions and extensions.

Frameworks used on the client side

Furthermore, the client component is supported by a number of JavaScript plugins and frameworks. The following plugins are used:

- *AngularJS*: Open source framework by Google which eases model view controller patterns and dependency injection.[11]
- *Date.Format*: Extention of Date-Object for simple date formatting.
- *Open Layers*: Open source framework for depiction of geospatial data.[12]
- *JQuery*: Open source framework for ease of DOM operations and utility functions.[13]
- *JQuery UI*: Open source framework based on JQuery for common user interface modules.[14]
- *JQuery TreeView*: Open source framework based on JQuery for tree view.[15]

5.2.1.2 Configuration

This section will give an overview about possible configurations of the simulation control center.

- New OpenStreetMap and busstop files can be added in the resources folder. OpenStreetMap filesneeds to end with "*.osm" and busstop files with "*.gtfs".
- The resources folder also contains the "properties.props" file. In this prototype it is only possible to change the suffix of serialized start parameters. This is used by the *XMLStartParameterSerialzer*.

```
# Save startparameter:

suffixStartParameterXML=xml
```

- The bindings for the services can be set in *OCModule*.

[11] See http://angularjs.org/
[12] See http://openlayers.org/
[13] See http://jquery.com/
[14] See http://jqueryui.com/
[15] See http://jquery.bassistance.de/treeview/demo/

- The simulation control center can be mocked in the global “simulation.conf” file. With
  `simulation.configuration.server.ccc.mock=true`
  With this setting the simulation control center will never receive any update from the SimulationController.

5.2.1.3 Extension Points

This section will give an overview about possible extension points of the simulation control center.

- Every service implementation can be extended and set in *OCModule*.
- New messages can be added by extending *CCWebSocketMessage* and adding the behavior into *CCWebSocketUser.onMessage()*. On the client side, the file *servletCommunicationService.js* in *js/module/services/* must be extended.
- On the client side, a new *OCWebSocketMessage.MessageType* needs to be added. This also needs to be added on the client side as well in *webapp/js/model/enums.js*.
- For new services, there are some very easy possibilities for extension. For instance, all services are served via dependency injection. The modules in *webapp/js/modules/services* can be extended by new implementations of the services and then bound in *webapp/js/modules/modules.js*.

5.2.1.4 Exception Handling

Exceptions from the simulation will be sent to the client via the ErrorMessage. On the client side, these errors are then processed and, in most cases, presented to the user in a message dialog.

Java Script exceptions are either presented to the user in a message dialog or available in the browser’s development tool console.

5.2.2 Component: Energy Controller

The energy controller is able to depict the energy consumption in kWh for a specified spatial point and timestamp. It is asked by the smart meter sensors about the current energy consumption in their measure radius.

5.2.2.1 Scheme

This UML class diagrams show the architecture of the energy controller. Only the important method signatures are shown. The parameter and return types are hidden.

The class diagram in Figure 35 shows the API of the energy project. The local interfaces are only the local views of the EJBs.

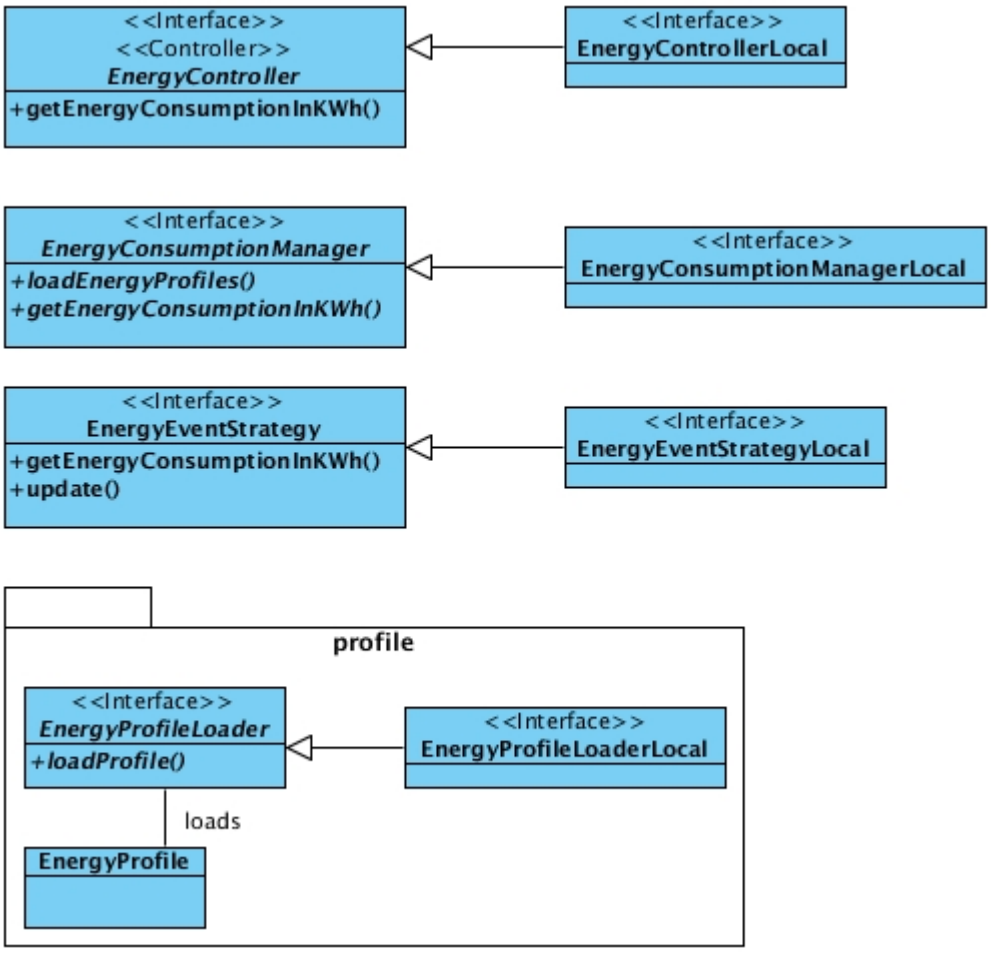


Figure 35: Energy API

- **EnergyController:** The energy controller is the most important class in the energy project. Its implementation can depict the current energy consumption for a specific area on a specific time.
- **EnergyConsumptionManager:** EnergyConsumptionManagers can be used to provide different behavior for an *EnergyController*. It can load profiles, e.g. CSV files, and they return the current energy consumption.

- **EnergyEventStrategy:** An EnergyEventStrategy can be used to handle *EnergyEvents*. The *EnergyEvents* influence the current energy consumption and needs to be handled. E.g. they can increase the current energy consumption.
- **EnergyProfileLoader:** EnergyProfileLoaders can be used to load energy profiles. Energy profiles give information about the typical energy consumption of different building types.

The class diagram in Figure 36 shows the implementations of the energy project.

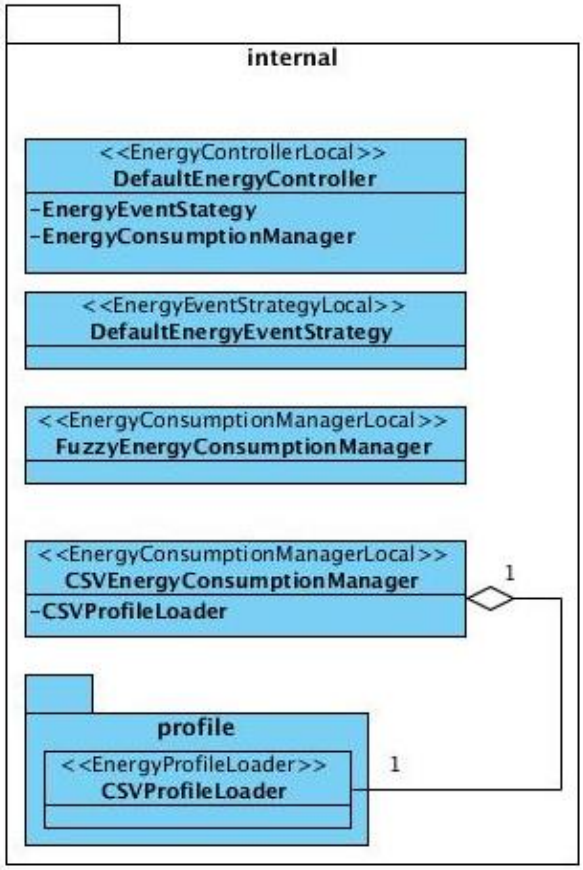


Figure 36: Implemented classes of Energy Controller

- **DefaultEnergyController**: The DefaultEnergyController is an*EnergyController*, that uses the via dependency injection set versions of *EnergyConsumptionManager* and *EnergyEventStrategy*. Whenever the current energy consumption is asked, it will call its *EnergyConsumptionManager* and then it will change the value with its *EnergyEventStrategy*.
- **DefaultEnergyEventStrategy**: DefaultEnergyEventStrategy uses PR-trees to calculate, if a spatial position is inside one of the *EnergyEvent* areas or not. If yes, it will change the given energy consumption by the EnergyEvents change method, if not it will just return the current energy consumption.
- **FuzzyEnergyConsumptionManager:** The FuzzyEnergyConsumptionManager uses maximum energy consumption values for every energy profile and changes their values by

applying fuzzy rules. The most important influence of this *EnergyConsumptionManager* is the weather.

- **CSVEnergyConsumptionManager:** The CSVEnergyConsumptionManager uses CSV files to retrieve the current energy consumption for the needed energy profile.
- **CSVProfileLoader:** The CSVProfileLoader is able to load the CSV files and gives them to the *CSVEnergyConsumptionManager*.

5.2.2.2 Configuration

This section will give an overview about possible configurations in the energy project.

- The bindings for *EnergyConsumptionManager*, *EnergyEventStrategy* and *EnergyProfileLoader* can be changed in "META-INF/ejb-jar.xml".
- Fuzzy rules for *FuzzyEnergyConsumptionManager* can be changed in "fuzzy/energy_fuzzy.fcl". The file is written in the FCL language[16].
- Other CSV files for the CSVProfileLoader can be added in the folder "profile".

5.2.2.3 Extension Points

This section will give an overview about possible extension points in the energy project.

- Fuzzy rules can be extended by changing the file "fuzzy/energy_fuzzy.fcl", which is written in the FCL language[16].
- Every interfaces can be implemented in another way in the concrete implementation can be set in "META-INF/ejb-jar.xml".

5.2.2.4 Exception Handling

Possible exceptions will be handled by the simulation controller and static sensor controller.

[16] Here is a description of the FCL language: http://jfuzzylogic.sourceforge.net/html/index.html

5.2.3 Component: Graph

The graph component includes classes to handle the structure of the graph. Therefore two different algorithms are implemented to disassemble the graph.

5.2.3.1 Scheme

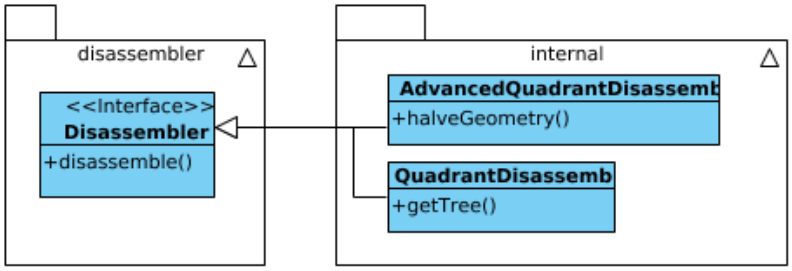


Figure 37: State transitions of a controller

The interface *Disassembler* offers a method to disassemble a given graph. The *QuadrantDisassembler* implements the *Disassembler*. Therefore the algorithm uses a binary tree to disassemble the graph into equal sized pieces, whereas the *AdvancesQuadrantDisassembler* is not using a tree for the process. The method *halveGeometry* divides the geometry object into two pieces. This is used to decide in which way the biggest part will receive a cut. The cut comprises a vertical or horizontal one. One of those two algorithms will be used by the traffic component to assist in working with the graph on different traffic servers. At the current time the *QuadrantDisassembler* is being used.

5.2.3.2 Configuration

There is no specific configuration.

5.2.3.3 Extension Points

There are no extension points.

5.2.3.4 Exception Handling

The exception handling is not needed within the disassembling process.

5.2.4 Component: Operation Center

The operation center is a web application for depiction of the simulation state, including high level functions and interfaces to an advanced data analysis tool.

Similar to the Simulation Control Center, the web application is comprised of a Java Servlet and a Java Script client, both communicating through WebSockets. The communication is message based, that is, there are predefined types of messages, each including a type flag, a unique message id and content. The interpretation of the content of a message is dependent on both the messages type and id.

The following message types (including denoted <content type>) exist within the operation center. However, refer to the documentation in the code for detailed information:

- *OCWebSocketMessage<T extends Object>:* Super type of all other messages. A basic message consists of a type, an id and content.
- *OnConnectMessage<OCOnConnectData>:* Sent by server if client establishes WebSocket connection. *OCOnConnectData* includes maps of sensor type names and sensor measure units for synchronization purposes.
- *SimulationInitMessage<OCSimulationInitParameter>:* Sent by server if simulation is initialized (but not started). *OSSimulationInitParameter* include general simulation properties, such as start and end time, simulation time interval and relation to real time, as well as, boundary information of the simulated city.
- *SimulationStartMessage<OCSimulationStartParameter>:* Sent by server if simulation is started. *OSSimulationStartParameter* include city name and population.
- *GateMessage<Map<Integer, Double>>:* Sent by client if user changes gate settings. The content is a mapping of sensor type and value (0 <= value <= 1.0).
- *NewSensorsMessage<Collection<SensorHelperTypeWrapper>>:* Sent by server if new sensors are added to the simulation. Content is a collection of sensors.
- *NewVehiclesMessage<Collection<VehicleData>>:* Sent by server if new vehicles are added to the simulation. Content is collection of vehicles. Vehicles may include sensors, such as GPS sensors or other sensors, for instance, a bus may contain an additional infrared sensor.
- *SensorDataMessage<Collection<SensorData>>:* Sent by server if new sensor data was processed by the data stream management system. Content is a collection of *SensorData*objects, each including sensor id, sensor type, payload and a timestamp.
- *GPSSensorTimeoutMessage<Collection<SensorData>>:* Sent by server if sensor was supposed to send but did not send sensor data. Content is collection of sensor data, including the id of the sensor that timed out. This sensor is believed to have reached his target or have lost a signal and, as a result, will be removed from the map.

- *RemoveSensorsMessage<Collection<Integer>>:* Sent by server if sensor was removed from simulation. Content is a collection of sensor ids. These sensors will be removed from the map and deleted from the client side simulation.
- *RemoveVehiclesMessage<Collection<VehicleData>>:* Sent by server if vehicle was removed from simulation. Content is a collection of vehicles. These vehicles, including their sensors if still existent, will be removed from the simulation.
- *ErrorMessage<String>:* Sent by server if some error occurred. Content is error message as string.
- *GenericNotificationMessage<String>:* Sent by server to notify that some operation was successfully completed. Content is confirmation message as string.
- *SimulationStoppedMessage<Void>:* Sent by server if simulation was stopped. There is no content in this message.

The messages are displayed in the class diagram in Figure 39. Moreover Figure 41 shows the logical sequence of the messaging system in a sequence diagram.

5.2.4.1 Scheme

Server-side architecture:

The operation center does not provide an API, because it is completely exchangeable, same as control center. The following UML class diagrams show the server-side architecture of the operation center with its classes and packages and their interaction points. Only the important method signatures are shown. The parameters and return types are hidden.

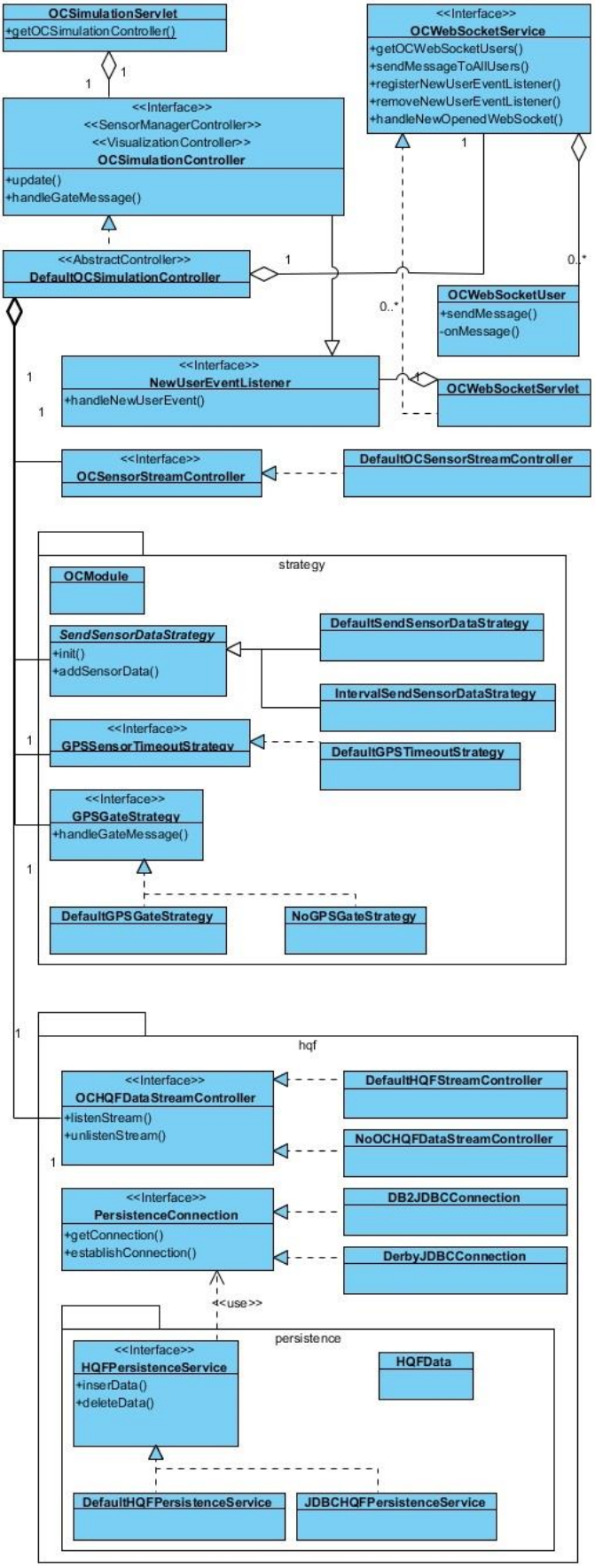


Figure 38: Operation Center class diagram

Figure 38 shows all the servlets, controllers and services of the operation center. They are descripted in the following listing.

- **OCSimulationServlet**: The OCSimulationServlet receives all the simulation information from the SimulationController. E.g. init, start, stop, update or create sensors. After receiving the information, it provides them to the *OCSimulationController*.
- **OCSimulationController**: The OCSimulationController handles all the received information and provides it to the user via *OCWebSocketService*. It can also receive and handle user information. New users can be observed with the observer pattern by registering at the *OCWebSocketService*.
- **DefaultOCSimulationController**: The default implementation of the *OCSimulationController* is the DefaultOCSimulationController. It uses several controllers and strategies to archive its goals. Used controllers are the *OCSensorStreamController* and the *OCHQFDataStreamController*. Used strategies are *SendSensorDataStrategy*, *GPSSensorTimeoutStrategy* and *GPSGateStrategy*. All the bindings can be defined in *OCModule*.
- **NewUserEventListener**: A NewUserEventListener can be registered at OCWebSocketService to observe new users. This is useful to allow users to join a running simulation.
- **OCWebSocketService**: OCWebSocketService holds all users. It can send messages to every user or give information about new users to registered observers.
- **OCWebSocketServlet**: The OCWebSocketServlet is the default implementation of *OCWebSocketService*. It can create the *OCWebSocketUsers*.
- **OCWebSocketUser:** Instances of OCWebSocketUser represent online users. They can send messages and receive messages. Possible messages are shown in Figure 39. They will be serialized by GSON to send them as JSON objects to the client. For this purposes the *sendMessage* method is used. It is also possible to receive and handle messages from the client. The *onMessage* method is used for this purpose. Received JSON messages will be deserialized and can be handled by the *OCSimulationController* then.
- **OCModule:** The class OCModule can be used to set all the bindings for *DefaultOCSimulationController*. The dependency injection in the operation center is done by google guice.
- **SendSensorDataStrategy:** A SendSensorDataStrategy decides when it is time to send the sensor data to the client. While the*DefaultSendSensorDataStrategy* sends after every single sensor data, the *IntervalSendSensorDataStrategy* sends the data after every interval.
- **GPSTimeoutStrategy:** The GPSTimeoutStrategy decides when a GPS sensor has reached its target. The DefaultGPSTimeStrategy marks a GPS sensor with a timeout, when a specified amount of update steps are missing.

- **GPSGateStrategy:** GPSGateStrategy can be used to change the settings of the data stream management system. It is the only setting that the user can do on runtime. The GPSGateStrategy will be used, if the user sends a *GateMessage*. In the *GateMessage* there is a percentage value for every known sensor type. It means the data steam management system shall be able to send only values for a specified amount of sensors. E.g. if the sensor type for car is set to 50%, it means the user wants to receive only information from 50% of every known car GPS sensor. The *NoGPSGateStrategy* ignores the *GateMessage*. *DefaultGPSGateStrategy* will send the wanted sensor IDs to the data stream management system. There needs to be one port to receive wanted sensor IDs and one port to receive unwanted sensor IDs in the data stream management system. Internally the DSMS will save the wanted sensor IDs and when this amount changes the unwanted sensor IDs will be send there and they will be deleted.
- **OCHQFDataStreamController:** The OCHQFDataStreamController is used to listen to a specific data stream and to forward the incoming data to *HQFPersistenceService*. There are two implementations: *DefaultHQFStreamController* reads data from an aggregated data stream and forwards it to the *DefaultHQFStreamController*. The other implementation ignores all incoming data.
- **HQFPersistenceService:** HQFPersistenceService saves the data into a data sink like a database table or a flat file. The *DefaultHQFStreamController* forwards data to the default implementation *DefaultHQFPersistenceService.*
- **PersistenceConnection:** The *PersistenceConnection* establishes the connection to the data sink and offers it to *HQFPersistenceService*. There are two implementations: *DB2JDBCConnection* establishing a connection to a DB2 database and *DerbyJDBCConnection*for an apache derby database.

The class diagram in Figure 39 shows all possible messages between the operation center's server- and client-side.

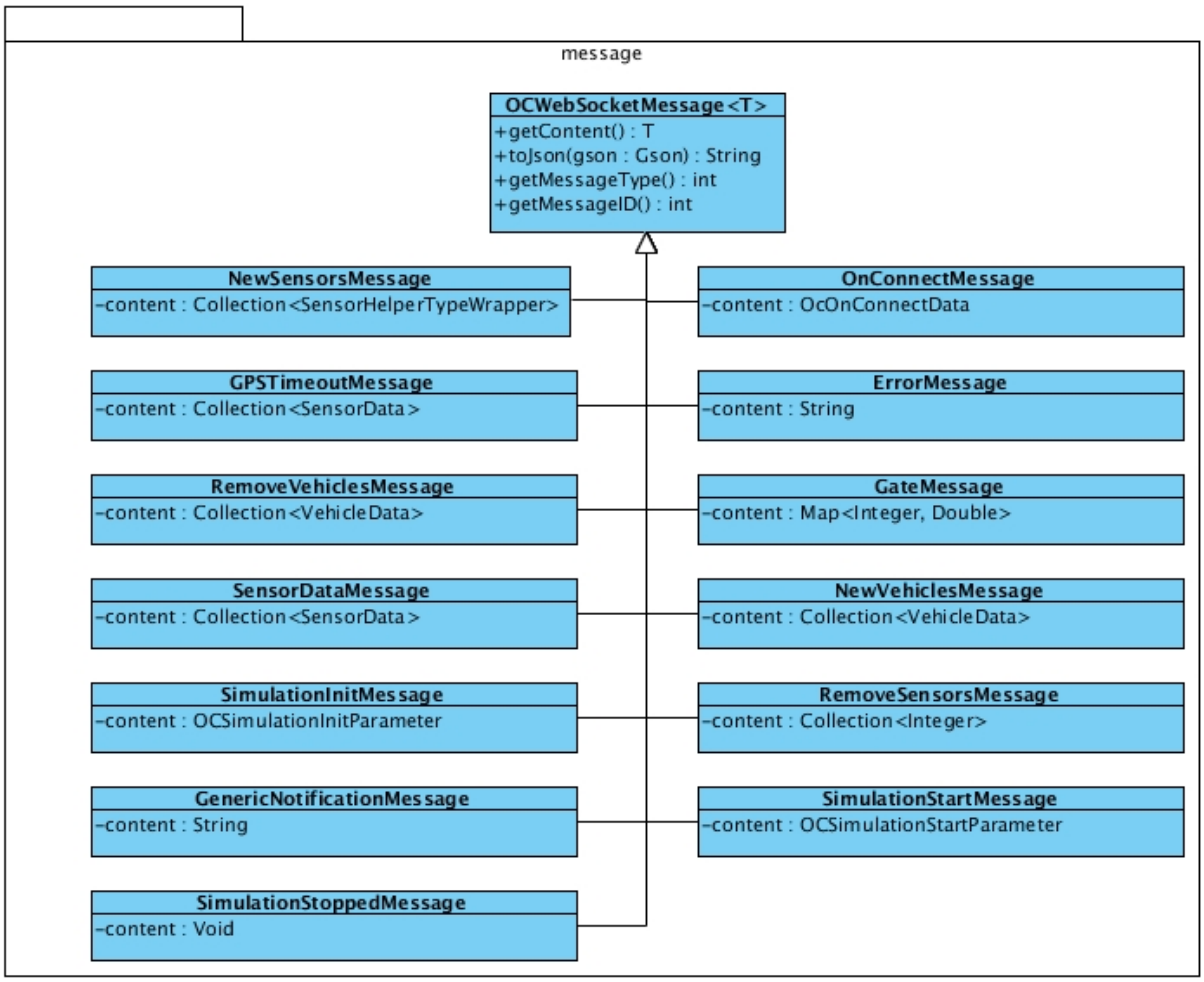


Figure 39: Control Center Messages

The used models of the operation center are displayed in Figure 40. The operation center has its own models to avoid unnecessary network traffic.

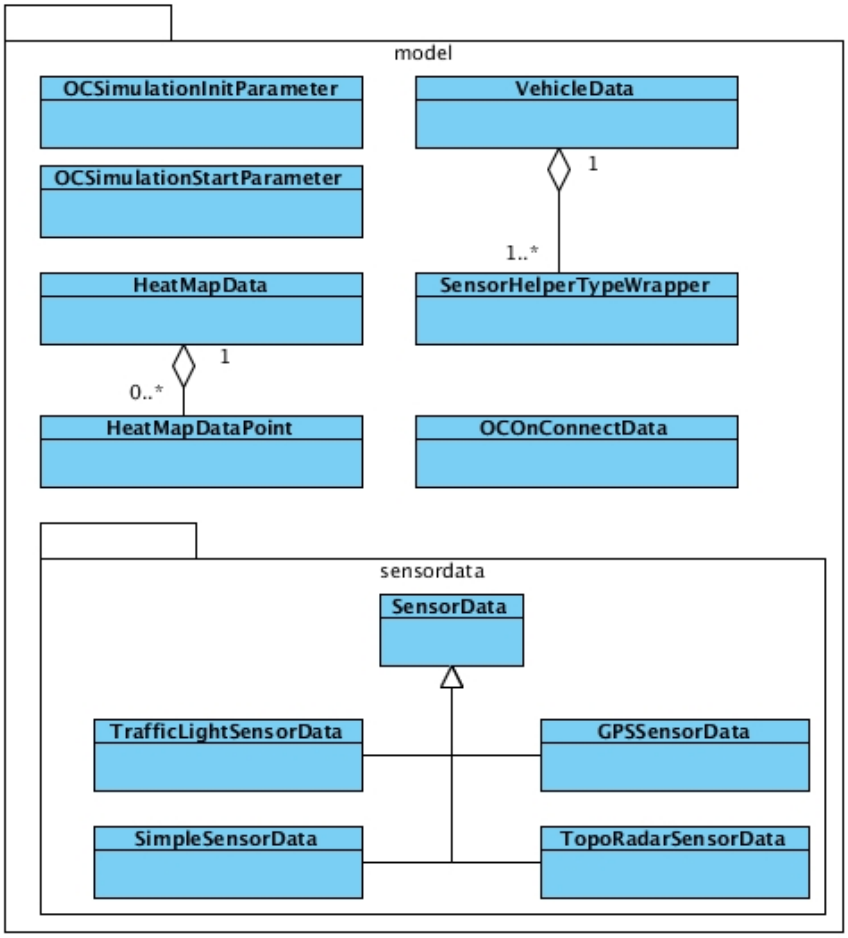


Figure 40: Control Center Models

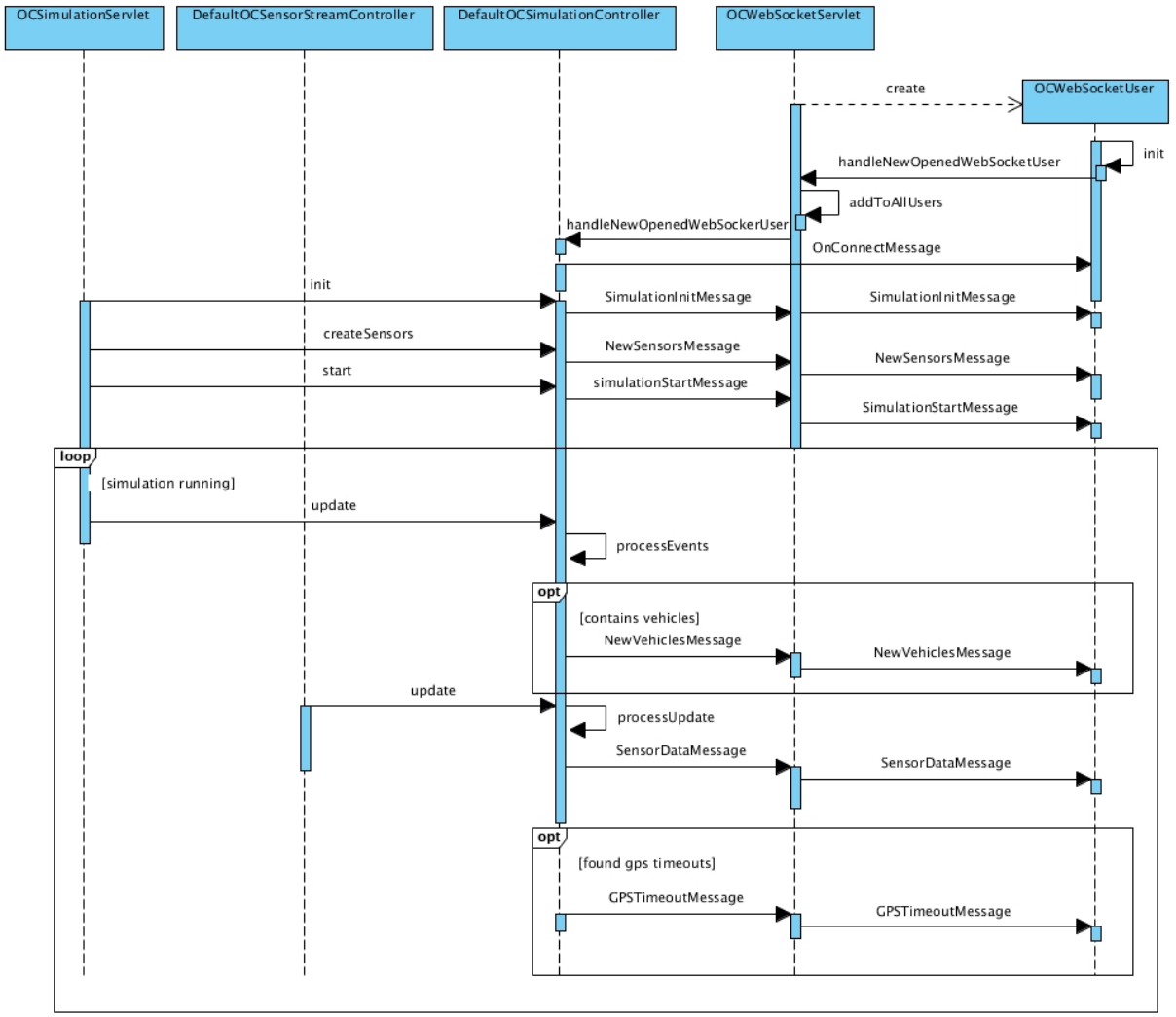


Figure 41: Control Center Sequence Diagram

The sequence diagram in Figure 41 shows the flow of the operation center.

- The *OCWebSocketServlet* creates the new user, add it to its list, and informs the *DefaultOCSimulationController* about it.
- In this case the simulation is not running. The user will receive the *OnConnectMessage*.
- The *OCSimulationServlet* receives an init call from the simulation and informs the *DefaultOCSimulationController* that informs every user with a *SimulationInitMessage*.
- The *OCSimulationServlet* receives a createSensors call from the simulation and informs the *DefaultOCSimulationController* that informs every user with a *NewSensorsMessage*.
- The *OCSimulationServlet* receives a start call from the simulation and informs the *DefaultOCSimulationController* that informs every user with a *SimulationStartMessage*.
- While the simulation is running, the OCSimulationServlet receives updates with events from the simulation. It informs the DefaultOCSimulationController, which processes the events and informs the users withNewVehiclesMessages, if the events contain new vehicles.

- While the simulation is running the *DefaultOCSensorStreamController* receives updates with sensor data from the data stream management system. It informs the *DefaultOCSimulationController* with the sensor data. Here the sensor data will be processed and send to the users will be informed via *SensorDataMessage*. After every update, the *DefaultOCSimulationController* checks for GPS timeouts. If there are any, the user will be informed with a *GPSTimeoutMessage*.

The simulation running loop is only for displaying the flow. In the real application it is done by more threads and the updates from the *OCSimulationServlet* and *DefaultOCSensorStreamController* will not arrive one after the other. The updates from the simulation are much faster, than the updates from the data stream management system. To avoid this skew, the *DefaultOCSimulationController* only sends the *SensorDataMessages* with timestamps and the clients will never know the real simulation time. Because of this skew, the only secure way to inform the client about the simulation stop is to wait for a new simulation init.

Client-side architecture:

The client side is made up of view components, controller components, model components and modules, including services.

The operation center is a single side web application, that is, there is only one main HTML file which is loaded at the beginning. Following interactions are WebSocket-based. The *body* tag of the *index.html* includes an *ng-app* attribute which binds the view to its controller components. *js/modules/modules.js* serves as the starting point of the application. All components are loaded with the HTML, since they are in the *head* tag of *index.html* but not initialized. When the *body* tag is processed, AngularJS initializes the main controller and then the other controllers.
The following class diagram depicts the view and controller components and their relationships.

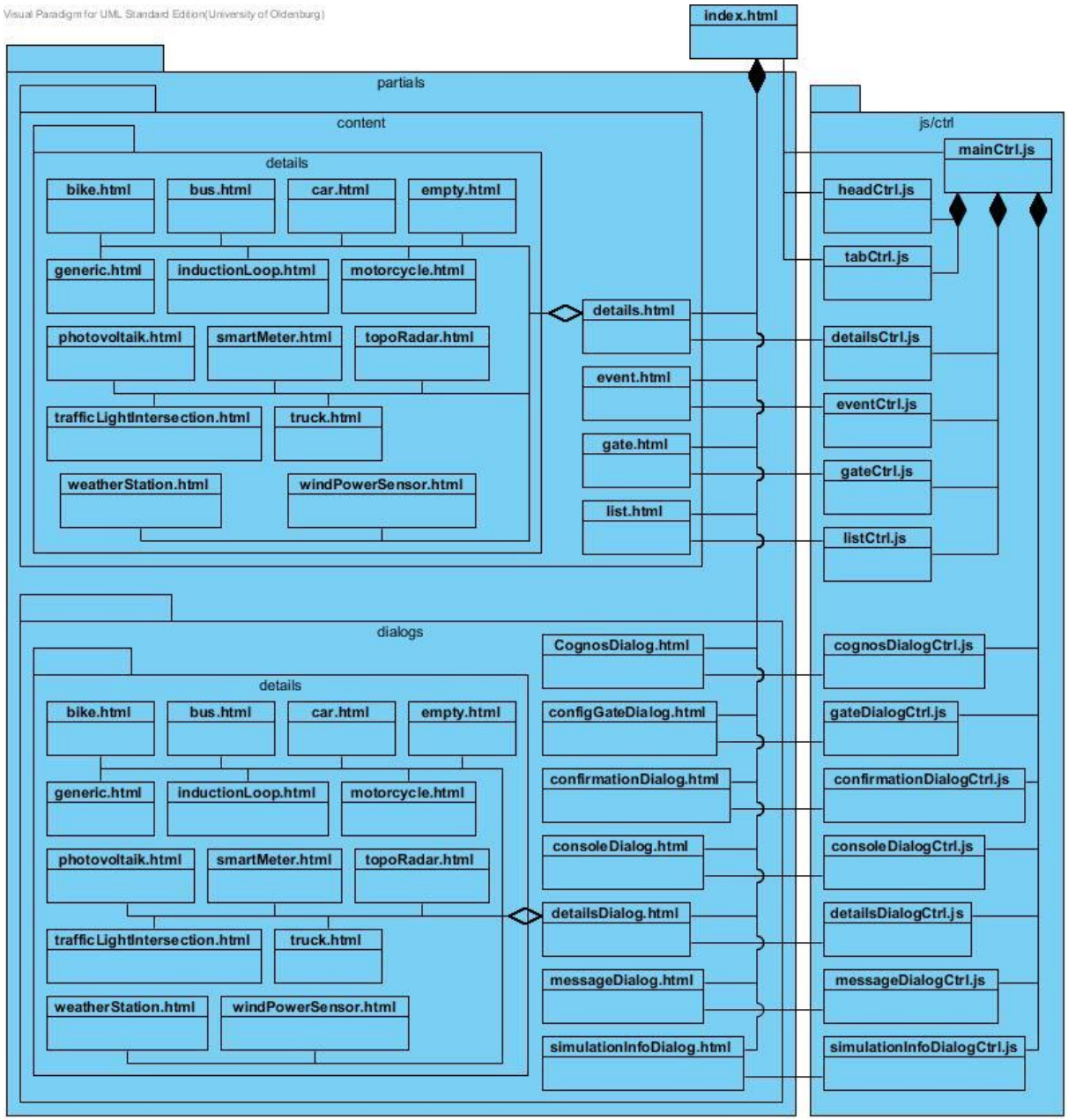


Figure 42: View and Controller Components of the Operation Center

View components:

View components are HTML files in *webapp/partials* directory. The *ng-model* tags represent the respective controller for the view element. Consequently, listCtrl.js contains the controller for the view list.html. Each controller has one scope for properties which can be jointly used by the controller and the view element, including all child elements (and their children and so forth).

- *content/list.html* is the view for the "List"-Tab on the left hand side of the UI.
- *content/gate.html* is the view for the "Gate"-Tab on the left hand side of the UI.
- *content/event.html* is the view for the "Event"-Tab on the left hand side of the UI.

- *content/details.html* is the view for the "Details"-Tab of the left hand side of the UI. This view includes an *ng:include*-Tag, which includes HTML-files dependent on the sensor which is presented by the view. These HTML-files are in *content/details* directory.
- *dialogs/CognosDialog.html* is the view for the IBMCognos dialog.
- *dialogs/configGateDialog.html* is the view for the gate configuration dialog.
- *dialogs/confirmationDialog.html* is the view for the confirmation dialog.
- *dialogs/consoleDialog.html* is the view for the console dialog.
- *dialogs/messageDialog.html* is the view for the message dialog.
- *dialogs/simulationInforDialog.html* is the view for the info dialog.
- *detailsDialog.html* is the view for the sensor detail dialog. Like for *details.html*, this view can include several HTML-files which are present in *details*.

Controller components:

Controller components are Java Script classes in *webapp/js/ctrl*. Each controller is represented in its own file. In the function parameters, the controllers scope-Object and several services can be included. For instance,

```
function MainCtrl($scope, SimulationService,
ServletCommunicationService, MapService, PopupService) { … }
```

includes the scope-Object and four services. The controllers are automatically instantiated by AngularJS as soon as the HTML is parsed, that is, if an HTML-chunk is put into the DOM, its respective controller will be instantiated. As a result, controllers are not ubiquous, since HTML can be removed from the DOM. As a result, controller life time is controlled by means of the DOM representation of the web application. Refer to the extensive in-code documentation for further details of how the controllers work.

- The controllers in *congnosDialogCtrl.js,consoleDialogCtrl.js, detailsDialogCtrl.js, gateDialogCtrl.js, messageDialogCtrl.js,* and *simulationInfoDialog.js* are the controllers for the respective dialogs (views).
- *detailsCtrl.js* and *detailsDialogCtrl.js* are the controllers for the detail view (and detail dialog view respectively). The controller watches for changes of the sensor data and, if this is the case, he updates its view. For this, the angular method $watch is used. For each sensor type, the controller provides the address of the sensor-detail-view which is to be included within the details view. Refer to the in-code documentation for further details.
- *gateCtrl.js, listCtrl.js* and *eventCtrl.js* are the controllers for the respective views.
- *tabCtrl.js* is the controller for the tab view component.
- *mainCtrl.js* is the main controller of the operation center. Every controller can use the main controllers scope. Consequently, every function of the operation center has its starting point in

the main controller, that is, communication or simulation services will call functions in the main controller, which then calls functions in the appropriate components.

Model components:

The model components encapsulate the models for the client side operation center in Java Script objects. The following class diagram depicts the model components:

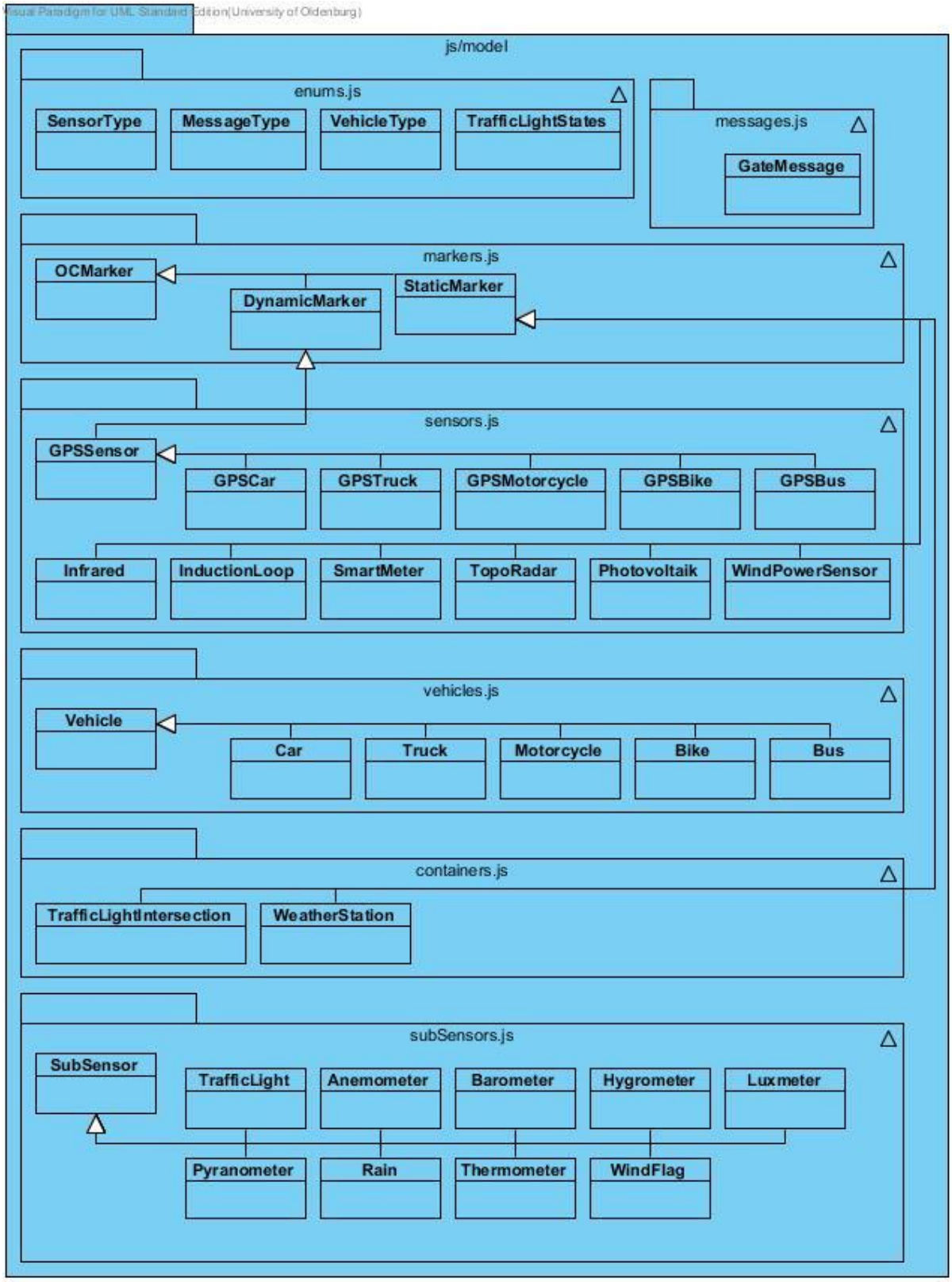


Figure 43: Model Components of the Operation Center

- *enums.js* includes all the enumerations of the operation center on the client side. Those are stored in the global variables *model.SensorType*, *model.MessageType*, *model.VehicleType* and *model.TrafficLightStates*.

- *messages.js* contains Java Script representations of the Java messages that will be sent to the server. As a result, these objects can be created, parsed into JSON, sent to the server and then parsed into Java Objects using GSON.
- *markers.js* contains the top level markers that are drawn onto the Open Layers map. *StaticMarker* and *DynamicMarker* extend *OCMarker*.
- *sensors.js* contains the sensors that are shown on the map. *GPSSensor* is the parent of special GPSSensor types for cars, trucks, bikes, busses and motorcycles. *GPSSensor* extends *DynamicMarker*. The other sensors extend *StaticMarker*.
- *vehicles.js* contains the vehicle object representations. These extend *DynamicMarker*.
- *containers.js* contains the Java Script objects *TrafficLightIntersection* and *WeatherStation*, which include sub sensors. Those are represented in *subSensors.js*. These sensors extend *SubSensor*.

Angular modules:

The angular models are provided in the *webapp/js/modules* directory. The following class diagram depicts the modules and their relationships:

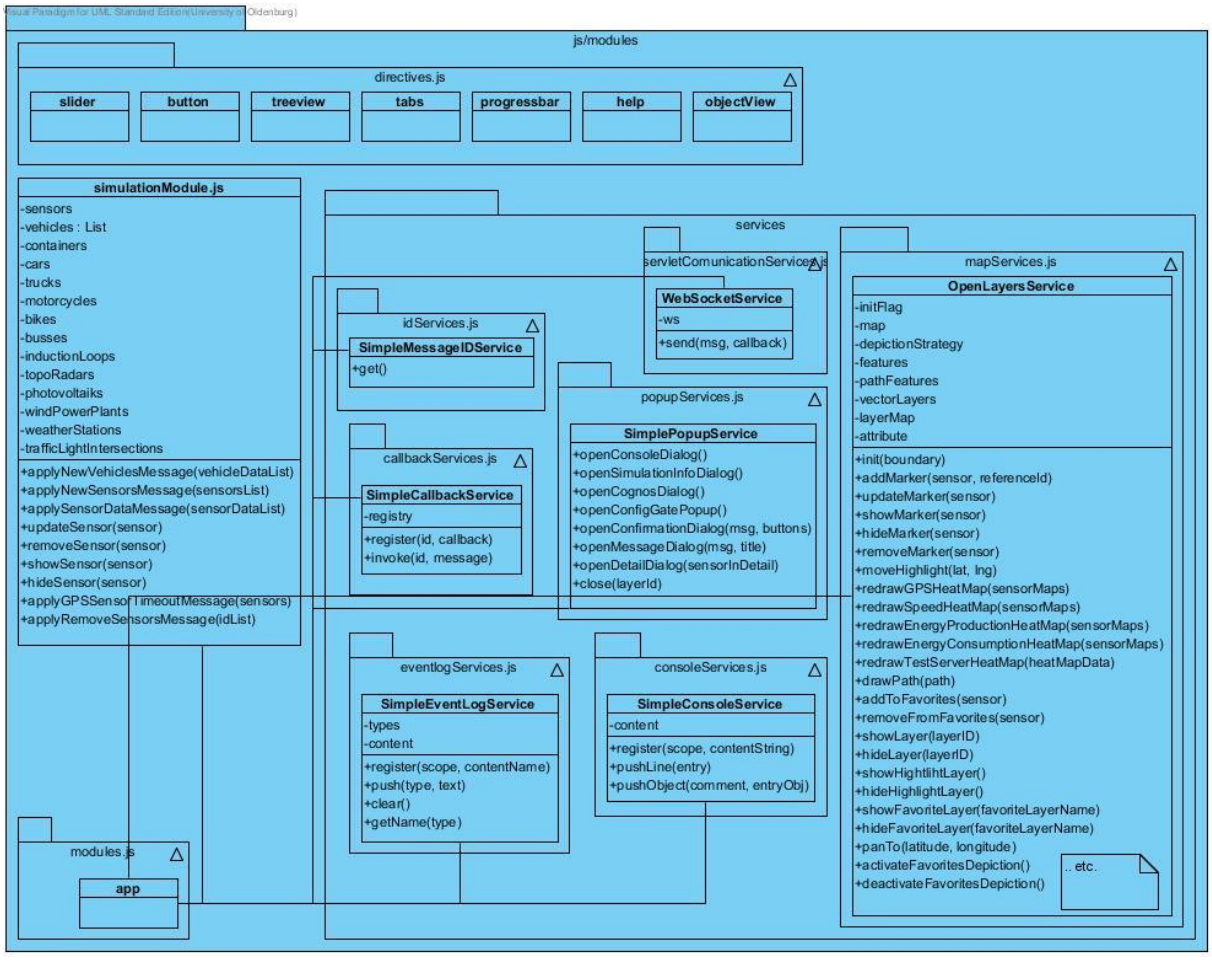


Figure 44: Module Components of the Operation Center

- *modules.js* is the Java Script starting point of the application. In *index.html*, there is an attribute *ng-app="app"*, which binds the body tag to the controller components. This is where the dependency injection is carried out, that is, injectable services like *MapService* are bound to a concrete implementation like *OpenLayersService*.
- *directives.js* contains some AngularJS directives, which are bound to HTML elements. These directives serve a constructor function which will be invokes when the respective HTML element is added to the DOM tree. Those constructor functions are used to call JQuery UI functions and for initialization of the component.
- *filters.js* can contain some custom filters but is empty for the operation center.
- *simulationModule.js* is the client side representation of the simulation, that is, all simulation objects are stored in this class. The simulation module is implemented as a service, so that is is singleton and available from everywhere in the application. Refer to the in-code documentation for further details.
- *services* directory contains all the service modules including the implementations for the services. Currently, there are just simple implementations for the services. However, more complex or alternative implementations can be added here. Dependency injection is then conducted in *modules.js*.
- *serices/callbackServices* contains a simple callback service, including a method for registering functions with message ids as callbacks. When a server side message with the same message id is received, the stored function will be called.
- *services/consoleServices* contains a simple console service, including methods for registering at the service and pushing individual lines or complex objects into the console. Components can register their scope and the name of an array object within this scope. As a result, when some content is pushed into the console, the consoleService redirects the content to all registered components and applies the respective views. When a complex object is pushed into the console, an intended string representation of the object will be printed to the console.
- *services/eventLogServices* contains functions similar to the console service. However, messages can be augmented with describing event types, such as *ERROR* or *NOTIFICATION*, which serves as possible ordering mechanism.
- *services/idServices* contains a simple implementation of an id service which returns an incrementing integer number.
- *services/mapServices* contains a Service, which serves as an API for handling the Open Layers map, e.g. for printing vector objects on the map or switching between depiction strategies.

Global Java Script objects:

The *js* folder itself contains three Java Script files:

- *global.js* contains the global variable of the application, including the packaged *ctrl* and *model*, as well as the global variable *log*, *mock* and *demo*. As a result, the log level (using *console.log()*), the mock mode (described in the next bullet point) and the demo mode (calling the mock method every *x* seconds) of the client side application can be enabled or disabled.
- *mock.js* contains a number of simple mock functions. For example, of the user clicks on the *ALISE* icon, random sensors will be pushed into the simulation.
- *Util.ls* contains a number of utility functions and extensions.

Frameworks used on the client side

Furthermore, the client component is supported by a number of JavaScript plugins and frameworks. The following plugins are used:

- *AngularJS*: Open source framework by Google which eases model view controller patterns and dependency injection.[17]
- *Open Layers*: Open source framework for depiction of geospatial data.[18]
- *Open Layers Heatmap*: Plugin for open layers for depiction of heat maps.[19]
- *JQuery*: Open source framework for ease of DOM operations and utility functions.[20]
- *JQuery UI*: Open source framework based on JQuery for common user interface modules.[21]
- *JQuery TreeView*: Open source framework based on JQuery for tree view.[22]
- *JQuery Sparkline*: Open source framework based on JQuery for simple graph depiction.[23]

5.2.4.2 Configuration

This section gives an overview about the configuration of the operation center.

- The bindings of the *DefaultOCSimulationController* can be set in *OCModule*.
- The "properties.props" file of the operation center provides several settings:
    - For *DefaultOCSensorStreamController* it is possible to set different IPs and ports for:

```
InfoSphereIPStaticSensors=134.106.56.58

InfoSphereIPDynamicSensors=134.106.56.58
```

[17] See http://angularjs.org/
[18] See http://openlayers.org/
[19] See http://www.patrick-wied.at/static/heatmapjs/
[20] See http://jquery.com/
[21] See http://jqueryui.com/
[22] See http://jquery.bassistance.de/treeview/demo/
[23] See http://omnipotent.net/jquery.sparkline/#s-about

```
InfoSphereIPTopoRadarSensors=134.106.56.58

InfoSphereIPTrafficLightSensors=134.106.56.58

InfoSherePortStaticSensor=12600

InfoSherePortDynamicSensor=12601

InfoSherePortTrafficLightSensor=12603

InfoSherePortTopoRadarSensor=12603
```

- The DefaultGPSGateStrategy uses:
  ```
  InfoSphereGateMessageIP=134.106.56.58
  InfoSphereGateMessageAddKeysPort=12345
  InfoSphereGateMessageRemoveKeysPort=12346
  ```
  To specify the IBM InfoSphere Streams IP and the ports for adding and removing sensor IDs.
- The DefaultOCSimulationController informs the DefaultGPSTimeoutStrategy about the possible number of missed GPS update steps, which is specified at:
  ```
  MissedGPSUpdateStepsBeforeTimeout=2
  ```
- The OCWebSocketServlet uses:

  ```
  MaxSensorDataInOneMessage=100

  MaxSensorHelperInOneMessage=100
  ```

  Tospecifying the maximal length of messages.

- The operation center can be mocked in the global “simulation.conf” file. With:

  ```
  simulation.configuration.server.occ.mock=true
  ```

- On the client side, see *webapp/js/global.js* for simple configuration issues.

5.2.4.3 Extension Points

This section gives an overview about possible extensions of the operation center.

- Every strategy, controller and service can be extended and the new bindings can be set in *OCModule*.
- New messages for communication with the user can be added by extending *OCWebSocketMessage*. A new *OCWebSocketMessage.MessageType* needs to be added. This also needs to be added on the client side as well in

webapp/js/model/enums.js.

- If a new sensor is added to the simulation and the output does not fit in one of the existing types in sensordata, a new extension of *SensorData* is needed and the *DefaultOCSensorStreamController* needs a new thread listen to the data stream management system's TCP/IP sink for the new sensor type. On the client side, the new sensor/vehicle can be added in *webapp/js/model/*. Furthermore, *webapp/js/module/simulationModule.js* should be extended by new sensor maps for storing the sensors of the new type. In either *applyNewSensorsMessage* or *applyNewVehiclesMessage*, a new *case* block must be added in the fashion of the existing sensors. Furthermore, for reacting on the new data type, the method *applySensorDataMessage* must be extended. The method *removeSensor* must be extended as well.
- For new services, there are some very easy possibilities for extension. For instance, all services are served via dependency injection. The modules in *webapp/js/modules/servives* can be extended by new implementations of the services and then bound in *webapp/js/modules/modules.js*.

5.2.4.4 Exception Handling

Every malicious exception in the simulation will be send to the user via an ErrorMessage. On the client side, these errors are then processed and, in most cases, presented to the user in a message dialog.

Java Script exceptions are either presented to the user in a message dialog or available in the browser’s development tool console.

5.2.5 Component: Parser

The project parser contains several needed parsers and their strategies.

5.2.5.1 Scheme

The following UML class diagram shows architecture of the parser project with its classes and packages and their interaction points. Only the important method signatures are shown. The parameters and return types are hidden. The description is divided by the packages.

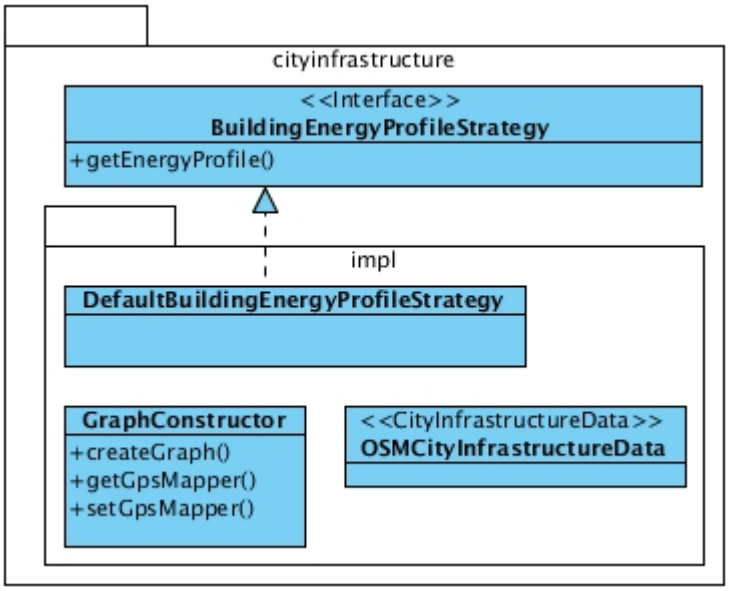


Figure 45: Parser Package Cityinfrastructure

The package *cityinfrastructure* is shown in Figure 45 and contains all classes to create a *Graph* for the simulation. The classes are described in the following listing:

- **BuildingEnergyProfileStrategy**: A BuildingEnergyProfileStrategy is used by *OSMCityInfrastructuteData* to determine the correct *EnergyProfile* for a building. It is an interface to make it exchangeable without changing *OSMCityInfrastructure*.
- **DefaultBuildingEnergyProfileStrategy**: The DefaultBuildingEnergyProfileStrategy is a default implementation of *BuildingEnergyProfileStrategy*. It determines the correct *EnergyProfile* of a building by the given tags, size and location of the building.
- **OSMCityInfrastructureData:** The OSMCityInfrastructureData is an implementation of the interface *CityInfrastructure* in the shared package. It uses given OpenStreetMap and bustop files the retrieve the needed information. The nearest node and nodes in boundary methods are implemented with a PR-tree to allow very fast searches. To allow different implementations of those methods, the *CityInfrastructureData* is an interface and not a class.
- **GraphConstructor:** The GraphConstructor can build a *Graph* by using an instance of *CityInfrastructureData*, e.g. *OSMCityInfrastructureData*.

The package GTFS.dataParser contains the *VWGGTFSCreator*, which helps to create the GTFS files for the busses of the VWG of Oldenburg. Because there are no GTFS files of Oldenburg yet we had to create them on our own. Because creating the GTFS files out of the timetables manually would be too time-consuming our first approach was to parse the PDF timetables automatically. Unfortunately this approach was to complex to create the needed files completely automatic because many bus routes differ irregular from the standard route. That's why we choose to take an abstract view of the timetables. Therefore the first trip of each bus line and the bus stops and bus stop times of this trip are the basis of our bus system. This data (stored as CSV files for each bus line and direction) were parsed by the *VWGGTFSCreator,* which creates (assuming that all busses drive every 15 minutes on workdays and every 30 minutes at the weekend) some of the GTFS files. The remaining GTFS files had to be created manually. Besides the *VWGGTFSCreator* the GTFS.dataParser package contains the *GTFSParser*. This component parses the GTFS files and stores them into a database.

The GTFS.service package contains an interface of the *BusService* and the *BusService* itself. The *BusService* provides a method that returns all existing bus routes, which is used for displaying the available routes in the Control Center. Another important service returns the bus line data for a specific bus line. These *BusTrip* objects contain information about the sensors (GPS and poss. infrared) of a bus, the bus stops to serve, the respective bus stop times, the route short name, the route long name etc.

5.2.5.2 Configuration

This section gives an overview about possible configurations in the parser project.

- The DefaultBuildingEnergyProfileStrategy uses the "properties.props" to determine the min square meters of a house and for an industry building.

  ```
  too-small-size-for-house=10

  min-size-for-industry=300
  ```

5.2.5.3 Extension Points

This section gives an overview about possible extension points in the parser project.

- Different versions of *BuildingEnergyProfileStrategy* can be implemented and given to the *OSMCityInfrastructureData*.

5.2.5.4 Exception Handling

The parser project does not need an exception handling, because it is only called from other projects.

5.2.6 Component: Sensor Framework

Sensor Framework defines both interfaces and base classes for application wide used components. Its designing goals are to establish a common structure for sensors, handling persistence and managing connections from a sensor to a host.

5.2.6.1 Scheme

The following UML class shows the structure of Sensor Framework. It is shortened to focus on the central aspects, but keeps all important information. In reality Sensor Framework is split up into an API and an implementation due to EJB usage. But to make the connections between interfaces and implementations more understandable it is summarized into one UML class diagram shown in Figure 46.

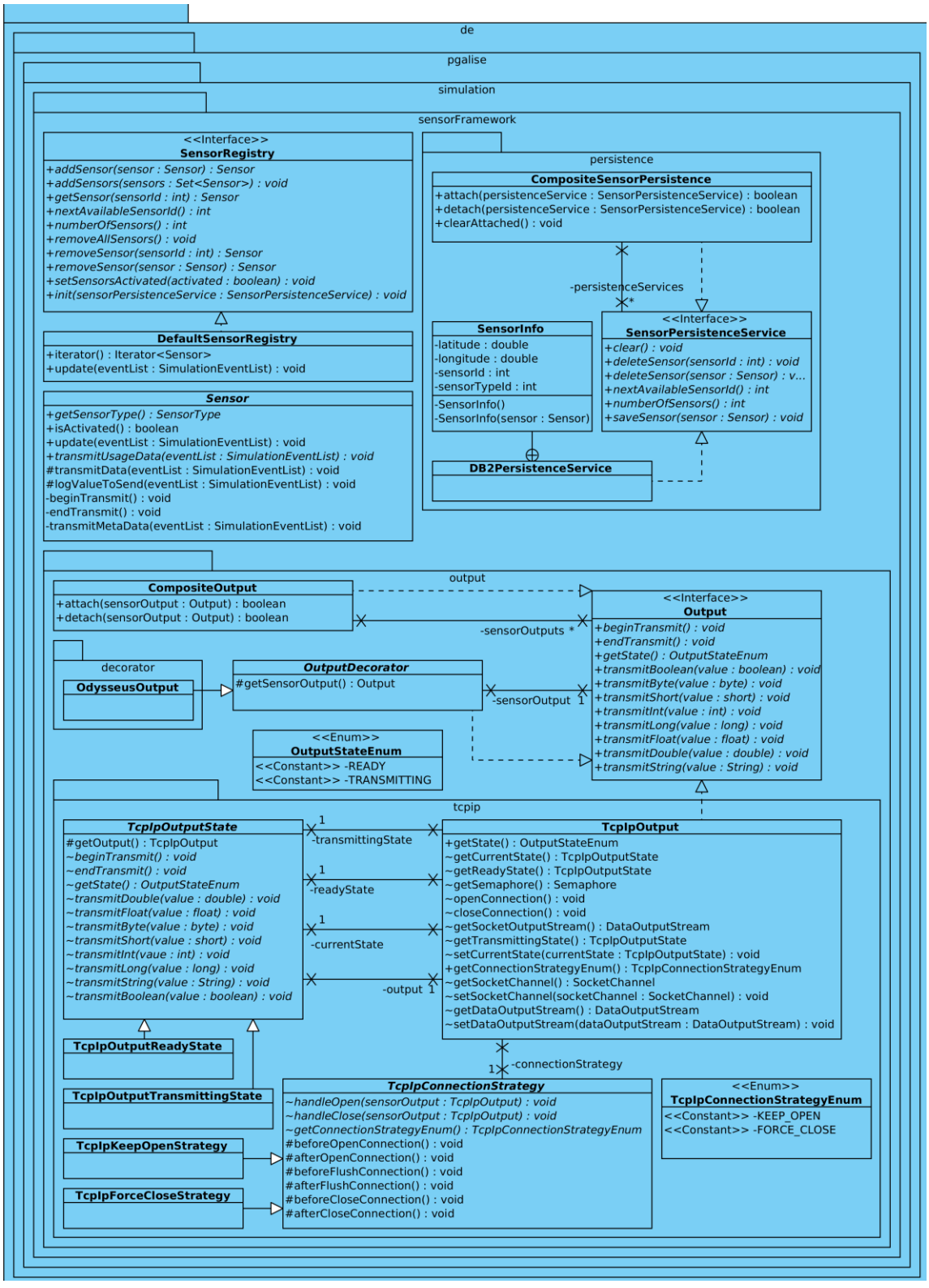


Figure 46: SensorFramework

Abstract class *Sensor* defines a base structure for all sensors used in the application. Each *Sensor* owns an *Output*. A *Sensor* uses its output for transmitting data to a predefined host (e.g. IBM InfoSphere Streams). The transmission sequence is defined by *Sensor* itself. Meta data and usage data are to be transmitted. *Output* as a low level component takes care about transmission. The concrete protocol is

ought to be specialized by its implementations. In the application *TcpIpOutput* is used to transmit data with TCP/IP protocol. *TcpIpOutput's* states are realized by a state pattern. In addition a *TcpIpConnectionStrategy* can be chosen to determine whether the *TcpIpOutput* is supposed to close connection after each tuple transmission or leave it opened. *Sensors* are stored in a *SensorRegistry*. *SensorRegistry's* job is to persist *Sensors* and offer methods to access *Sensors*. For persistence it can use an implementation of *SensorPersistenceService*. Sensors are currently saved into a DB2 database by *DB2PersistenceService* used by *DefaultSensorRegistry*.

5.2.6.2 Configuration

There is mainly one configuration that can be done by users in short time. Users can determine the *TcpIpOutput's* host by assigning the line *simulation.configuration.server.sensor_output* in *tomee/applib/simulation.conf.*

5.2.6.3 Extension Points

Many extension points exist in Sensor Framework. It is possible to reimplement each interface. If the user for instance wants to store sensors in a file instead of use a DB2 database he should write a new implementation of *SensorpersistenceService* where he puts in his own logic. He also could change the way data is transmitted by defining his own output. All in all SensorFramework is highly loose coupled. This means every component can be replaced or enhanced.

5.2.6.4 Exception Handling

Exceptions may occur at some problematic places. For instance, at sensor's data transmissions due to an unreachable host. At the moment the application will be stopped, because it would totally break applications logic when data can't be sent to servlet. There are some other cases where application won't be killed but where is given a warning to the user. Failed database commits in *SensorPersistenceService* are an example, since there is no further usage of persisted sensors in database at the moment.

5.2.7 Component: Processing

After simulating synthetic sensors it is mandatory to process this streaming data and export this data for the client or web application. The following figure shows the data stream processing in the context of the whole application.

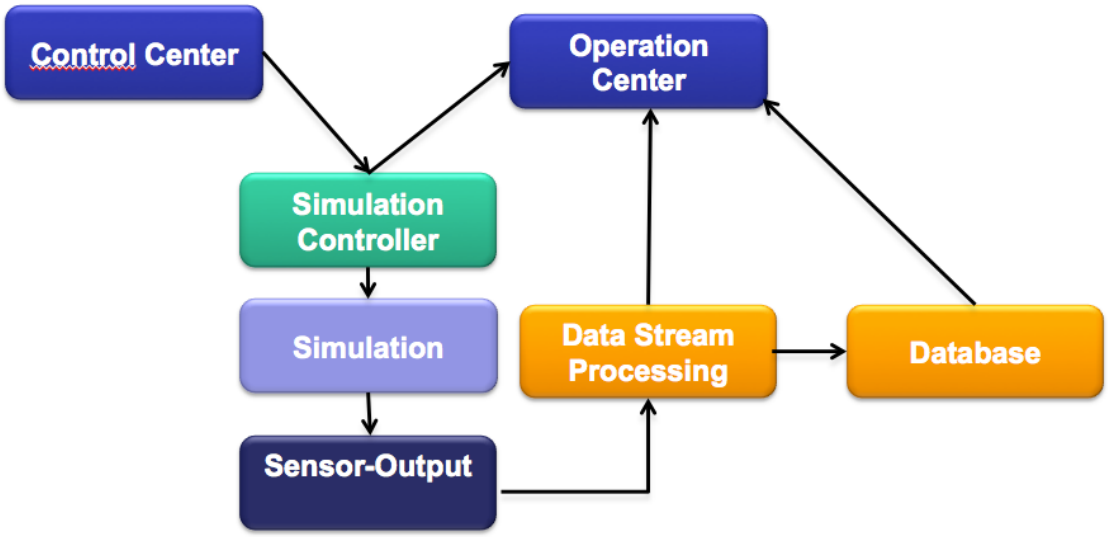


Figure 47: Architecture of the whole application

In our application we are using two different data stream management systems (DSMS), IBM InfoSphere Streams 3.0 and Odysseus [AG12]. General information about InfoSphere Streams can be found in the basic chapter. Odysseus is an open source data stream management framework and the result of a still ongoing research project of the Carl von Ossietzky Universität Oldenburg and the institute for information technology OFFIS. Odysseus can be used to create efficient applications that require data stream or event based processing. The main idea was to easily change the DSMS in the whole application.

The main focus was on implementing a data stream application with all its function, also higher functions, in IBM InfoSphere Streams3. In Odysseus the application was realized prototypically. In the next sections both applications will be explained in detail.

To establish the connection from the simulation to the DSMS, the simulation has to know the server address and an approved port of the specific server. That means the DSMS acts as the server and the simulation or the sensor output as the client.

For the output the DSMS will send the output tuple from defined ports. The web applications (Operation Center or the collaborating servlet) have to watch on the server address and ports to collect and interpret this data stream.

To link the DSMS to the simulation a specific interface has to be declared.

Therefore all the sensor data sent from the simulation have the same scheme, which fits for all sensors and delivers all relevant data for further processing. This scheme is shown later on.

To differentiate between different sensor data, each tuple contains a sensor type id.

The Sensor Type IDs are:

"0": "ANEMOMETER",
"1": "BAROMETER",
"2": "GPS_BIKE",
"3": "GPS_BUS",
"4": "GPS_CAR",
"5": "HYGROMETER",
"6": "INDUCTIONLOOP",
"7": "INFRARED",
"8": "LUXMETER",
"9": "PHOTOVOLTAIK",
"10": "PYRANOMETER",
"11": "RAIN",
"12": "SMARTMETER",
"13": "THERMOMETER",
"14": "TRAFFICLIGHT",
"15": "WINDFLAG",
"16": "WINDPOWERSENSOR",
"17": "WEATHER_STATION",
"18": "TOPORADAR",
"19": "GPS_TRUCK",
"20": "GPS_MOTORCYCLE"

5.2.7.1 Input Scheme

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

double *measureValue1* : measure value1 of a sensor

double *measureValue2* : measure value2 of a sensor

byte *axleCountOrStatus* : number of axles (topo radar) or the status of a traffic light

short *lengthOrIntersectionID* : length of a vehicle (topo radar)

short *axialDistance1* : distance between axle 1 and axle 2 (topo radar)

short *axialDistance2* : distance between axle 2 and axle 3 (topo radar)

Example usage of the input scheme:

Input GPS data stream

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

double *measureValue1* : latitude

double *measureValue2* : longitude

byte *axleCountOrStatus* : empty (0)

short *lengthOrIntersectionID* : empty (0)

short *axialDistance1* : empty (0)

short *axialDistance2* : empty (0)

Example GPS Tuple: 3122131010109, 12345, 4, 3.121, 8.2132, 0, 0, 0, 0

Input wind power sensor data stream

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

double *measureValue1* : measure value of the wind power plant

double *measureValue2* : empty (0)

byte *axleCountOrStatus* : empty (0)

short *lengthOrIntersectionID* : empty (0)

short *axialDistance1* : empty (0)

short *axialDistance2* : empty (0)

Example wind power sensor tuple: 3122131010109, 12346, 16, 30021, 0, 0, 0, 0, 0

Input topo radar

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

double *measureValue1* : empty (0)

double *measureValue2* : empty (0)

byte *axleCountOrStatus* : number of axles

short *lengthOrIntersectionID* : length of a vehicle

short *axialDistance1* : distance between axle 1 and axle 2

short *axialDistance2* : distance between axle 2 and axle 3

Example topo radar tuple: 3122131010109, 12347, 18, 0, 0, 3, 3134, 1862, 0

5.2.7.2 Output schemes

In this section the different output schemes, that were already mentioned, are declared explicitly.

GPS

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

double *measureValue1* : latitude

double *measureValue2* : longitude

double *distance* : driven distance in one update step

int *totalDistance* : total driven distance of this vehicle since start

byte *speed* : the speed of the vehicle

byte *avgSpeed* : the average speed of one type of vehicle

byte *direction* : the direction of a vehicle in degree

long *travelTime* : the total travel time of this vehicle since start

Traffic Light

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

double *measureValue1* : measure value 1

double *measureValue2* : measure value 2

byte *Status* : status of the traffic light

short *intersectionID* : the ID of the intersection

Topo Radar

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

byte *axleCountOrStatus* : axle count

short *length* : length of the vehicle

short *axialDistance1* : axial distance between axle 1 and axle 2

short *axialDistance2* : axial distance between axle 2 and axle 3

Static Sensors

long *currentMillis* : timestamp of the simulation

int *sensorId* : the unique ID of a sensor

byte *sensorTypeId* : the ID of the sensor type

double *measureValue1* : measure value 1

5.2.7.3 InfoSphere Streams 3.0

The application has six different Input operators, each for a server in our distributed architecture. Afterwards the different Input streams will be unionized and processed individually. This processing consists of sorting, deleting duplicates and higher geospatial functions. At the end the data streams will be output over a TCP-Output.

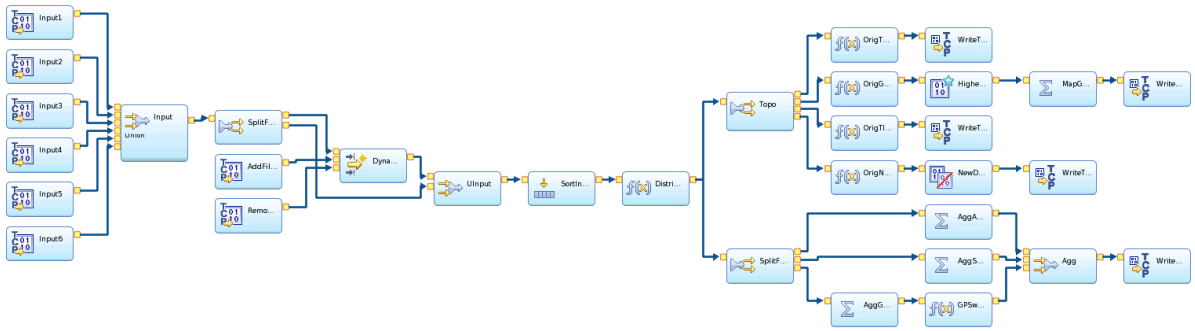


Figure 48: Overall streaming application in InfoSphere Streams 3

A high-res figure of this streaming application is in the appendix (25) of this document.

At first the data will be input over six *TCP-Source* operators. This operator allows each server in our distributed architecture to have its own input stream, so they will not block each other's connection. The connection-policy of the *TCP-Source* operator is set to "Infinite Retry". This means after a connection loss it is allowed to reconnect to the InfoSphere Streams server or to the *TCP-Source* operator. In the next step these input streams will be unionized with the *Union* operator to synchronize these streams. Independently of the amount of input streams they will be unionized correctly into one output stream. After this the stream will be split in two streams, one stream containing only tuple with GPS data and one stream for the other tuples. The GPS tuples will be input in the *DynamicFilter* operator. With this operator it is possible to change the amount of different vehicles in the Operation Center during the simulation. In this operator the amount of tuples, which will be output, can be changed on runtime. Therefore the *DynamicFilter* operator has three input ports. The first port is the stream of tuples to be filtered, in this case the tuples containing GPS data. The tuples on the first port will be passed through the *DynamicFilter* if their *sensorId* matches a valid *sensorId* within the operator. The second port is a stream of the allowed *sensorIds*, which will be added to the valid keys in the *DynamicFilter* operator. The third port is a stream of tuples that contain the *sensorIds* that will be removed from the valid keys. On default there are no allowed ids, this means if no *sensorIds* gets in, no tuples will pass through. The data streams containing the allowed and the ids that should be removed from the allowed list are sent from the Operation Center servlet. This servlet creates the list of ids depending on the actions in the Operation Center and send this data via a TCP data stream to InfoSphere Streams 3. After the *DynamicFilter* operator the data stream will be unionized again with the other data stream. The next operator is the *Sort* operator to order the tuples based on ordering expressions and window configurations. It is important to sort the streams for the visualization, because the output of the simulation may be a little out of order hence some part of the simulation take more time than others, e.g. the traffic simulation. In this operator the stream will be sorted ascending by the *currentMillis* in a sliding window of 50 tuples. A sliding window was chosen to guarantee a constant and non-hopping output, as it would be with a time based and tumbling window. The following operator (DistributeStream) duplicates the stream in two output streams, one for aggregating

the sensor data, saving historical data and using this in IBM Cognos to create reports, the other one is for processing this data and sending it to the Operation Center.

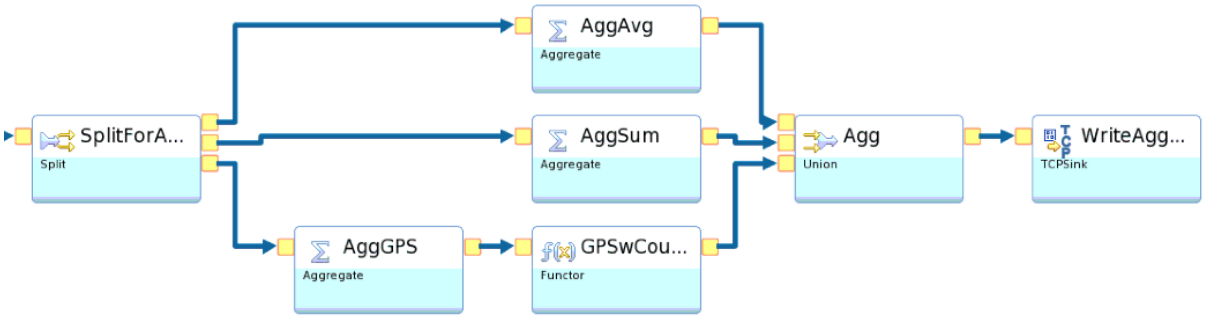


Figure 49: Aggregation stream

Following the aggregate stream, the first operator in this part is a split operator. This operator splits the stream into three output streams depending on the aggregation method for each *sensorTypeId* and if it make sense for this measureValue1. All *Aggregate* operators in this context are using a tumbling window on the delta of the simulation timestamp. This window has a size of ten minutes simulation time. This means every ten minutes simulation time an aggregated data stream will be output

The first aggregate operator calculates the average of *measureValue1* and takes the highest timestamp every ten minutes simulation time. This behavior is used for the barometer, anemometer, hygrometer, luxmeter, photovoltaic, pyranometer, thermometer, smart meter, wind flag and the wind power sensor.

The second aggregate calculates the sum of the *measureValue1* and takes the highest timestamp, too. This behavior is used for the induction loop, infrared and rain sensor. The third aggregate operator counts the vehicles or sensors of a GPS sensor type every ten minutes simulation time.

The following *Functor* operator maps the attribute *count* to the *measureValue1* attribute to keep the standard scheme. After this these three data streams will be unionized again into one data stream. Finally the *TCP Sink* operator will output this unionized data stream. This data is send to the servlet and will be saved by a java operation into a database (in this case the IBM DB2). The export scheme is the same as the source scheme.

On base of this data different IBM Cognos reports are created. The needed data will be loaded on runtime from the database if the user opens the reports in the Operation Center.

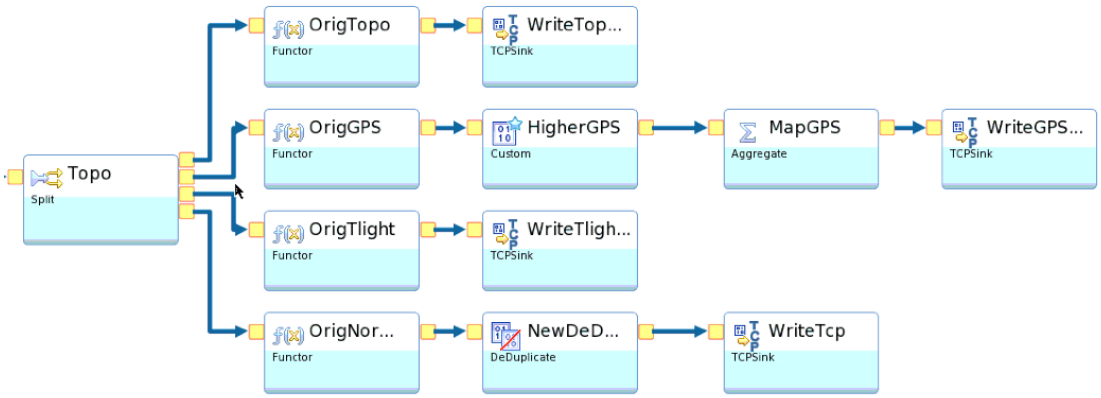


Figure 50: Processing Stream

Going back to the data stream (see the operator DistributeStream), which will be processed for the Operation Center, the next operator is a *Split* operator. This operator splits the stream in four different streams, again depending on the *sensorTypeId*. One data stream for topo radar data, one for traffic light data, one for the other static sensors and one for the GPS data. This separation is beneficial, because the current scheme is different by each data stream. It is now possible to drop irrelevant attributes and reduce the tuple size to enhance the performance. All schemes of the outgoing tuples can be found in Chapter 5.2.7.2.

The topo radar data stream scheme will be reduced in the next step by the *Functor* operator and afterwards the data stream will be output by a *TCP-Sink*.The same procedure goes for the traffic light data stream, however the scheme is different from the topo radar.

The scheme of the data stream containing the static sensors will be reduced by a *Functor* operator, too. After this, double measure values of a specific *sensorId* will be filtered out of the static sensor data stream by the *DeDuplicate* operator to enhance the performance and reduce the tuple output rate to the Operation Center. The *DeDuplicate* operator suppresses duplicate tuples that are seen within a given time period. In this operator the measure value of a sensor id will be checked within a period time, in this case 0.5 seconds, if the measure value has changed. If it has changed, the tuple will be passed through or else all duplicates will be filtered out. Finally this data stream will be output by a *TCP-Sink* operator.

Once the GPS data stream has been reduced to the relevant attributes by the *Functor* operator a *Custom* operator adds additional information to this data stream. This operator generates higher information out of the latitude and longitude with the geospatial toolkit of InfoSphere Streams 3. Within this operator the current speed, distance and total distance, the travel time and direction will be calculated. For this it is necessary to save the relevant data in a map, because in each operation it is mandatory that the old latitude and longitude, the first timestamp and the current travel time of a specific sensor id is know. With the old and new latitude and longitude two points can be created and

the geospatial functions can be called with this two points to calculate the distance between those points and the heading direction from the old point to the new point. Out of these results further information can be produced, like the speed and the total travel time.

In the following *Aggregate* operator the average speed per vehicle type will be calculated. Afterwards the data stream will be output via a *TCP-Sink* operator.

The exact implementation of this data stream application is in the appendix (26) of this document.

5.2.7.3.1 Perspective

In context of the whole application different aggregate mechanisms for specified *sensorTypeIDs* were tried out. The problem on aggregating the data before sending it to the operation center was to maintain a fluent simulation, for example no extremely hopping cars or the possibility to directly view the effect of an event. After a lot of testing the results were not satisfying, because either the reduction of data was too less or the aggregate window was too big. Additionally the servlet building the data objects for the Operation Center could not identify the lifetime of certain vehicles. This means some cars just remained on the map though they were not driving anymore. This whole context can be analyzed in perspective, if there is a way to establish a fitting aggregation type and window for the different sensor types in the processing stream. In this context aspects of load shedding, besides dropping duplicate, could be investigated for the application.

In addition to that further higher functionalities can be implemented. For this the consisting *Custom* operator in the GPS data stream processing can be extended or a complete new custom operator can be created with additional geospatial functions, like a shortest path algorithm considering the current traffic or the amount of other vehicles in a certain area around the sender. Besides of that the other sensor types can be evaluated if there are potentials for higher functionalities, too. In the case of a topo radar it is interesting if a vehicle can be recognized again by another topo radar and which time the vehicle needed between these measure points and what route they might took. Putting all the information the simulation delivers together it can be examined how these information can be used and processed in InfoSphere Streams to predict further happenings in the simulation, like traffic jams on certain bottlenecks.

Since it is possible to start the InfoSphere Streams application in the distributed mode, this allows live metrics of this application. With these metrics deeply analysis of the performance of the overall application and out of it performance enhancements can be done in future. Furthermore the Streams application can be distributed on additional server for a better performance. For this it is necessary to install InfoSphere Streams on these additional servers and add these servers to the main managing server as hosts for streaming applications. Within the application different operators have to be collected in a processing element (PE), because these PEs are the unit in which the application can be

distributed on different hosts or servers. In the ideal case the processing of individual sensors could take place on different hosts in parallel.

5.2.7.4 Odysseus

The requirement exists to create an interface for the DSMS Odysseus. Besides just delivering an interface a prototype was developed. This prototype can be the starting point for the further use of Odysseus in this Smarter City Application.

The developed prototype bases on the implementation in InfoSphere Streams 3. Basically all sensor data will be received in Odysseus and processed individually, to create the exact data stream output schemes as in InfoSphere Streams. Additionally the data streams can and will be directly saved in a database for the analysis of historical data (see the figure below).

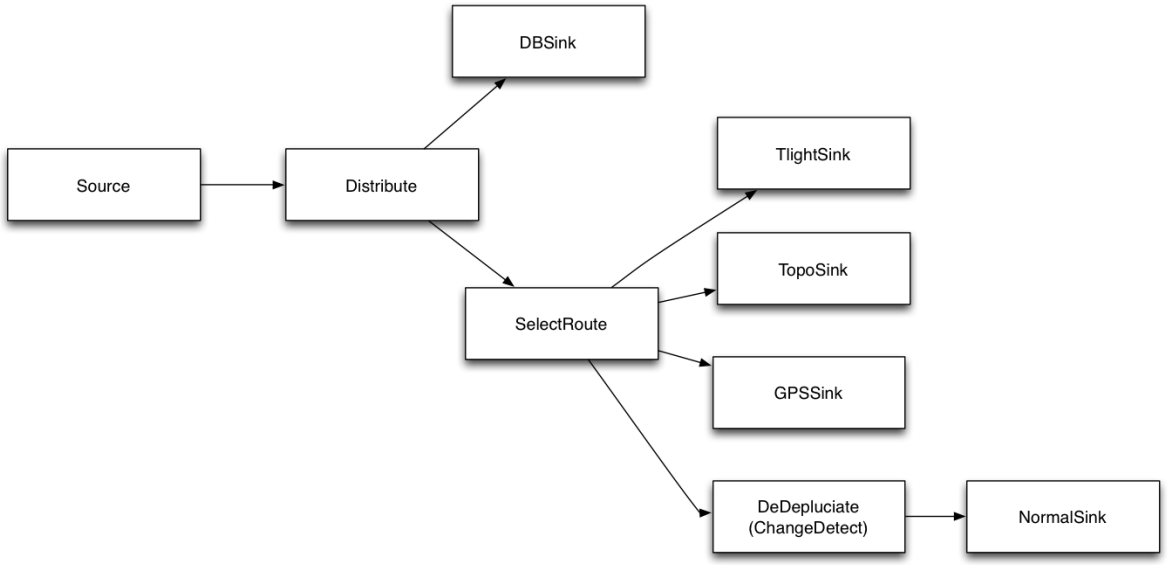


Figure 51: Graph of the Odysseus Application

The whole application is written both in CQL and in PQL. The first source operator defines Odysseus as the server in the same way as in InfoSphere Streams. In this operator it is mandatory to use the *MarkerByteBuffer* as the protocol handler. This means the input needs a marker at the beginning and at the end of a tuple. In this application -1 and -1 is used, so the input has to be extended in the Smart City Application. Because of the different schemes there exist two different outputs in the sensor framework in our application.

In the next step the data stream will be split via different *Select SQL-Statements.* Through these SQL-statements the scheme will already be reduced to the relevant attributes in the streams.The topo radar, traffic light and GPS stream will directly send out at specific ports each by a TCP-Sink operator, called in PQL the *Sender* operator. The data stream containing the tuples of the static sensors will be

cleared of its duplicates before sending it out to, to reduce the data rate. For this the *Changedetect* operator was used. In this operator, similar to the equivalent operator in InfoSphere Streams, the *measureValue1*of a concrete *sensorId* will be compared in a given window, if there are duplicates. The *tolerance* attribute of this operator was set to zero, so just exact duplicates will be filtered out. Finally this reduced data stream will be output via TCP with the *Sender* operator.

The exact implementation of this data stream application is in the appendix (27) of this document.

5.2.7.4.1 Extendibility of the prototype

The different operations declared above are strictly divided into different Odysseus script files, so it is very easy to add new operators or new files to the whole data stream application.
One function that should and can be added is the support of multiple TCP inputs. So far these inputs are already defined but it is mandatory to create an *Assureheartbeat* operator for the case that not on all input operators, tuples are coming in. This heartbeat is used by the union operator to recognize after a given time period that all data (or no data) arrived for the union and the streams will be unionized.

Another function might be some different aggregation behaviors before saving the tuples to the database, like the ones in the application in InfoSphere Streams. Imaginable are the sum, average and the count of vehicles in a given period of time. Since Odysseus has also geospatial toolkit some additional information can be created out of the GPS data stream.

These extensions are the base to gain the features of the InfoSphere Streams application. Besides of that more information can be added by functions/operationson individual sensor types since these different types are generally processed individually.

5.2.8 Component: Services

This component provides several services that are used throughout the whole application and are necessary to fulfill important requirements. For instance there is a service called *RandomSeedService* to produce deterministic seeds for random number generators. Most services are annotated as local Enterprise Java Beans (EJB) and can be therefore automatically injected to needing components by the EJB container.

5.2.8.1 Scheme

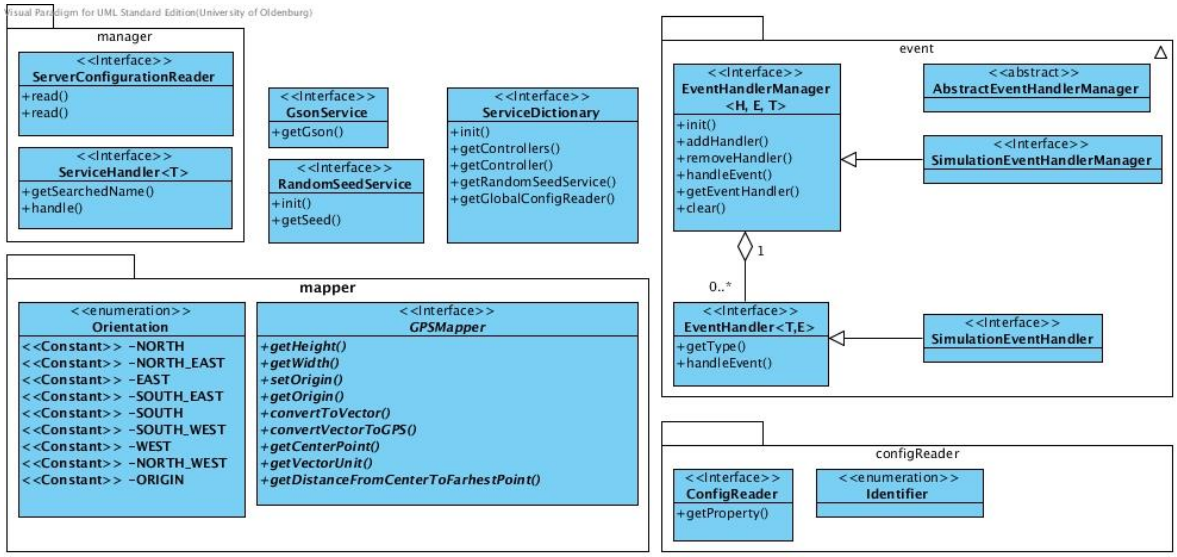


Figure 52: Services used throughout the whole application

Figure 52 shows an overview of the provided services. Only the important method signatures are shown. The parameter and return types are hidden. In the following a short description of the services is given.

- **GsonService**: Provides gson instances with type adapters, to allow serialization and deserialization of subclasses.
- **RandomSeedService**: A service producing deterministic seeds for random number generators.
- **GPSMapper:** A vector based coordinate system is internal used by the simulation to position the simulation entities (e.g. sensors, vehicles) in the city, but for the visualization and sensor output GPS coordinates are needed. The *GPSMapper*takes care to map between these two coordinates.
- **Orientation**: Enum of directions (e.g. EAST, NORTH) used by the *GPSMapper*.
- **ConfigReader**: Enables the user to read properties found in the configuration of the simulation. It depends on the implementations how the configuration is technically backed upand accessed (e.g. table in a database, accessed through JDBC).
- **Identifier**: Enumeration of properties used by the *ConfigReader* to configure the simulation.

- **ServiceDictionary**: A dictionary to retrieve as singleton implemented services that may reside on different servers. None of the services listed here are supposed to be referenced directly. Instead a component that depends on one of these services has to use this dictionary to get a reference.
- **ServerConfigurationReader**: Looks for services the user is interested in on the servers that are listed in the *ServerConfiguration*. Whenever a service could be found a callback function (*ServiceHandler*) is executed in order to inform the user and pass him the resolved reference to the service.
- **ServiceHandler**: Has two jobs: Firstly, to tell the ServerConfigurationReader which service the user is looking for; secondly, to provide a callback function which will be invoked with the reference to the service whenever it could be found.
- **EventHandlerManager**: Manages several *EventHandler* loaded on initialization. Each *EventHandler* listens for a particular type of event.When the *EventHandlerManager* is told to process an event it looks for the appropriate *EventHandler* and informs it.
- **EventHandler**: Is responsible for processing a particular event. An event itself is recognized by its event type.
- **AbstractEventHandlerManager**: A generic base class that can be used to implement an *EventHandlerManager*. All functions and methods of the interface are already implemented. However the implementer has to implement an abstract that determines which handler has to be called to process an incoming event.
- **SimulationEventHandlerManager**: Manages *SimulationEventHandler*.
- **SimulationEventHandler**: An *EventHandler* that is responsible for processing simulation events.

5.2.8.2 Configuration

The implementation of the *ConfigReader* (DefaultConfigReader) reads the configuration parameter of the simulation through an input file (simulation.conf) located in root directory of the EJB container in the directory *applib*. For further details see the User Guides.

5.2.8.3 Extension Points

Every service can be changed by implementing the interface and set it to the current implementation with EJB annotations.

To build a new event handler mechanism a new class that extends the AbstractEventHandlerManager has to be created, implementing, if necessary, an interface derived from the EventHandlerMangager interface. Furthermore corresponding EventHandler has to be defined.

5.2.8.4 Exception Handling

The service component provides no additional exception handling.

5.2.9 Component: Shared

The package shared contains several models and controller interfaces for the other projects.

5.2.9.1 Scheme

The following UML class diagrams show the architecture of the different packages in shared. Only the important method signatures are shown and the parameter and return types are hidden. The description is divided by the packages in shared.

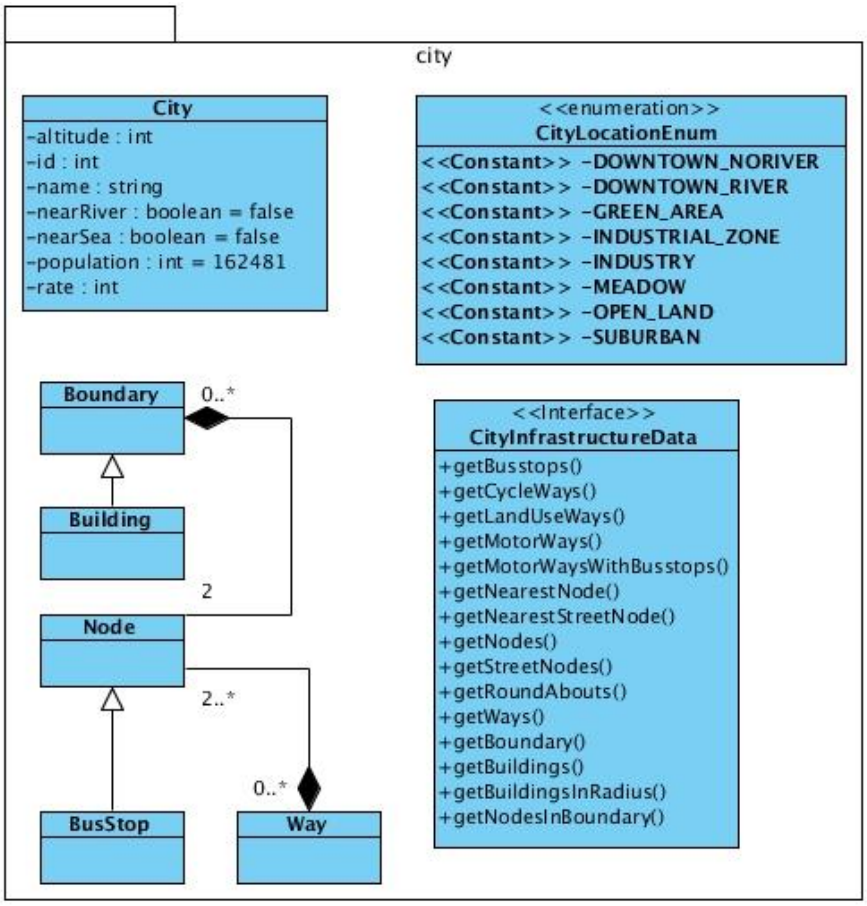


Figure 53: Shared Package City

The package city is shown in Figure 53 and provides useful information about the city.

- **City**: The class City gives information about the city. It is used to determine the perfect simulation weather.
- **CiyLocationEnum**: In the CityLocationEnum are possible constants for different city locations.
- **CityInfrastructurData**: An Instance of CityInfrastructureData gives information about the infrastructure of the city. This instance can be used to create a graph of the traffic system of

the city for the traffic project. It is an interface to allow different ways to retrieve the information and different algorithms to provide it. E.g. methods like *getNearestStreetNode()* can be implemented in different ways with different complexity.

- **Boundary**: A boundary provides the boundary of a city or a building. It is only a rectangle and provides the locations for north-east and south-west.
- **Building**: Buildings are special types of *Boundary* with information about the area inside the boundary.
- **Node**: The class node has a *GeoLocation* point and an ID. The ID is the same as in the *Graph*.
- **Way**: A way is a list of Nodes with several information, like highway, cycle way or speed limit.
- **BusStop:** A BusStop extends a *Node* by a bus stop name.

The package controller is shown in Figure 54 and provides the interfaces for every controller in the prototype.

- **Controller**: A Controller can be initialized, reseted, started, stopped and return the name and current status.
- **AbstractController**: The AbstractController is an abstract implementation of the *Controller*. It provides the correct state transitions. The transitions are shown in Figure 55.
    - The first transition is the instantiation. The controller switches in the INIT state.
    - In the INIT state it can be initialized and switches into the INITIALIZED state.
    - In the INITIALIZED state it can switch back to INIT by reset or switch into the STARTED state by start.
    - In the STARTED state the controller can receive updates, if it also implements the SimulationComponentManager interface and is known by the *EventInitiator* of the *SimulationController*. The other possibility in STARTED is to stop. Then the controller switches into the STOPPED state.
    - In the STOPPED state it can be restarted with start or resetet by reset.
- **InitParameter**: The InitParameter provide information to init the controllers
- **StartParameter**: For the start of every controller, the information in StartParameter is needed.

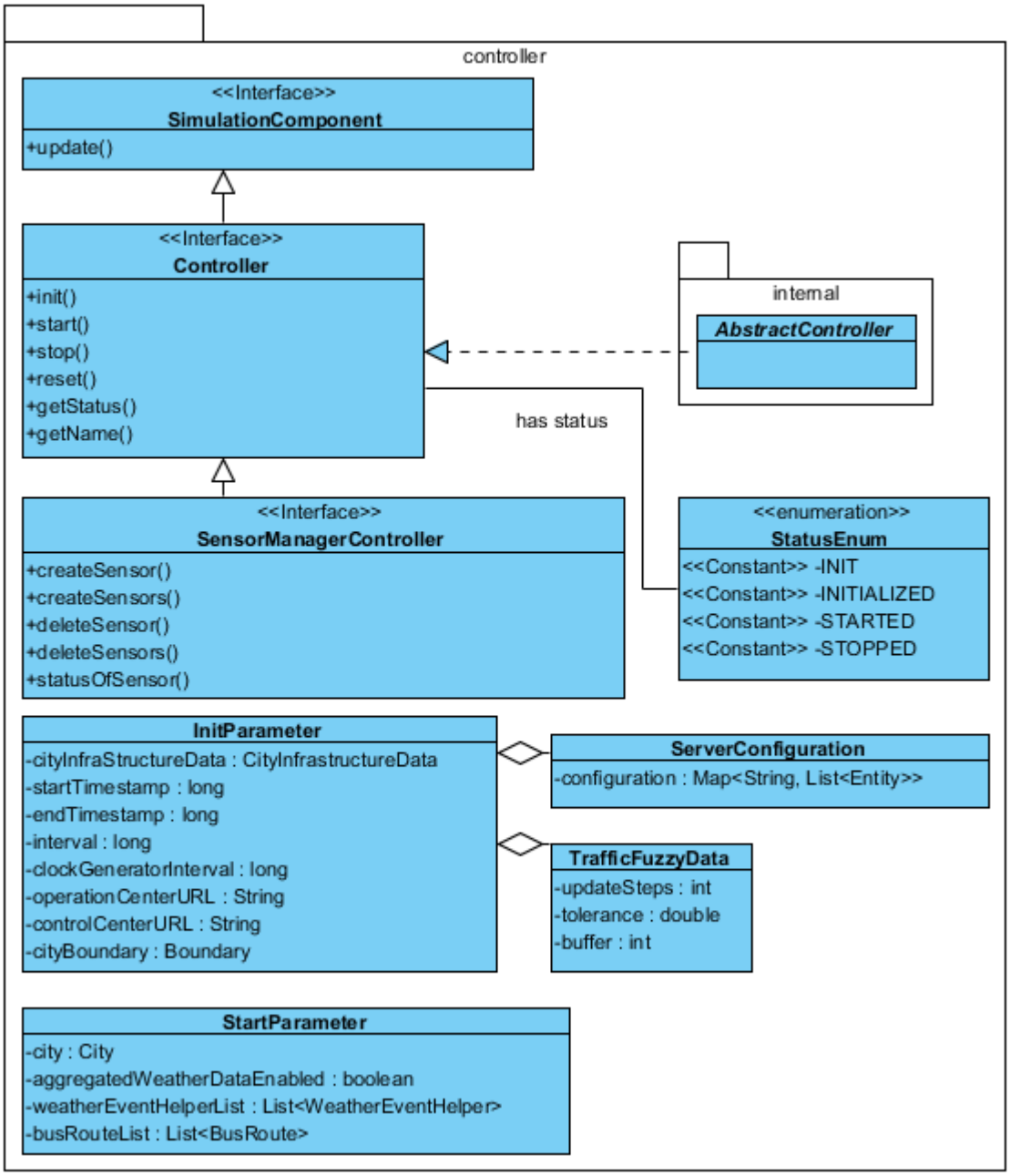


Figure 54: Shared Controller Package

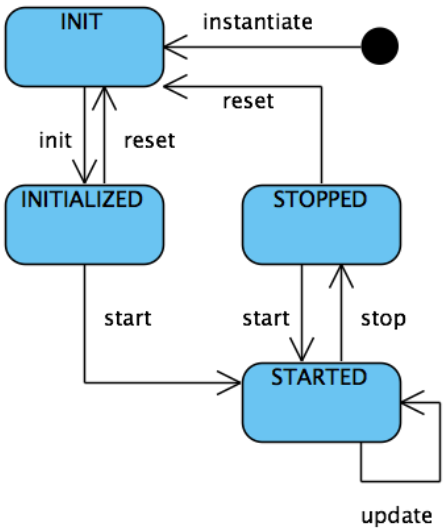


Figure 55: State transitions of AbstractController

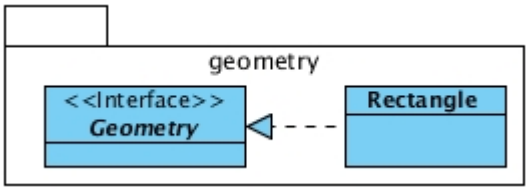


Figure 56: Package Geometry

The package geometry is shown in Figure 56 and provides models for geometry.

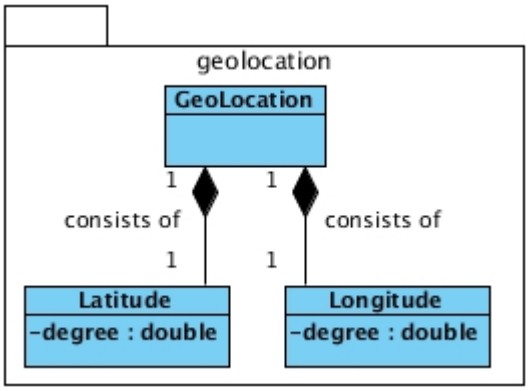


Figure 57: Package GeoLocation

The package GeoLocation provides all models for spatial data. It is shown in Figure 57.

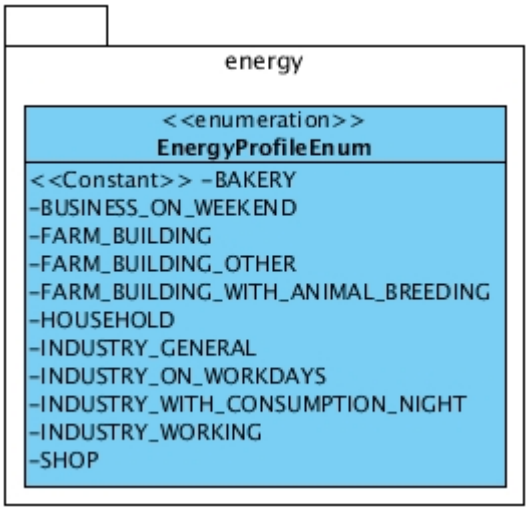


Figure 58: Package Energy

The package *energy* is shown in Figure 58 and contains only an *EnergyProfileEnum*. This enum has constants for every possible energy profile.

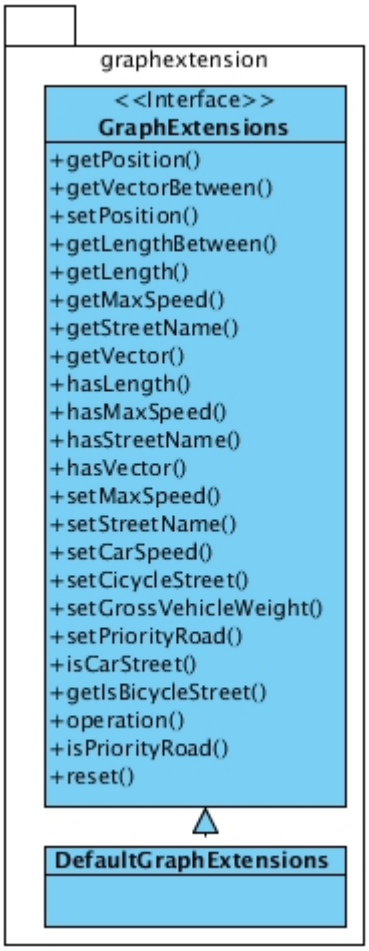


Figure 59: Package GraphExtension

The package *graphextension*, displayed in Figure 59, contains an interface and also an implementation with extensions for the used *Graph*.

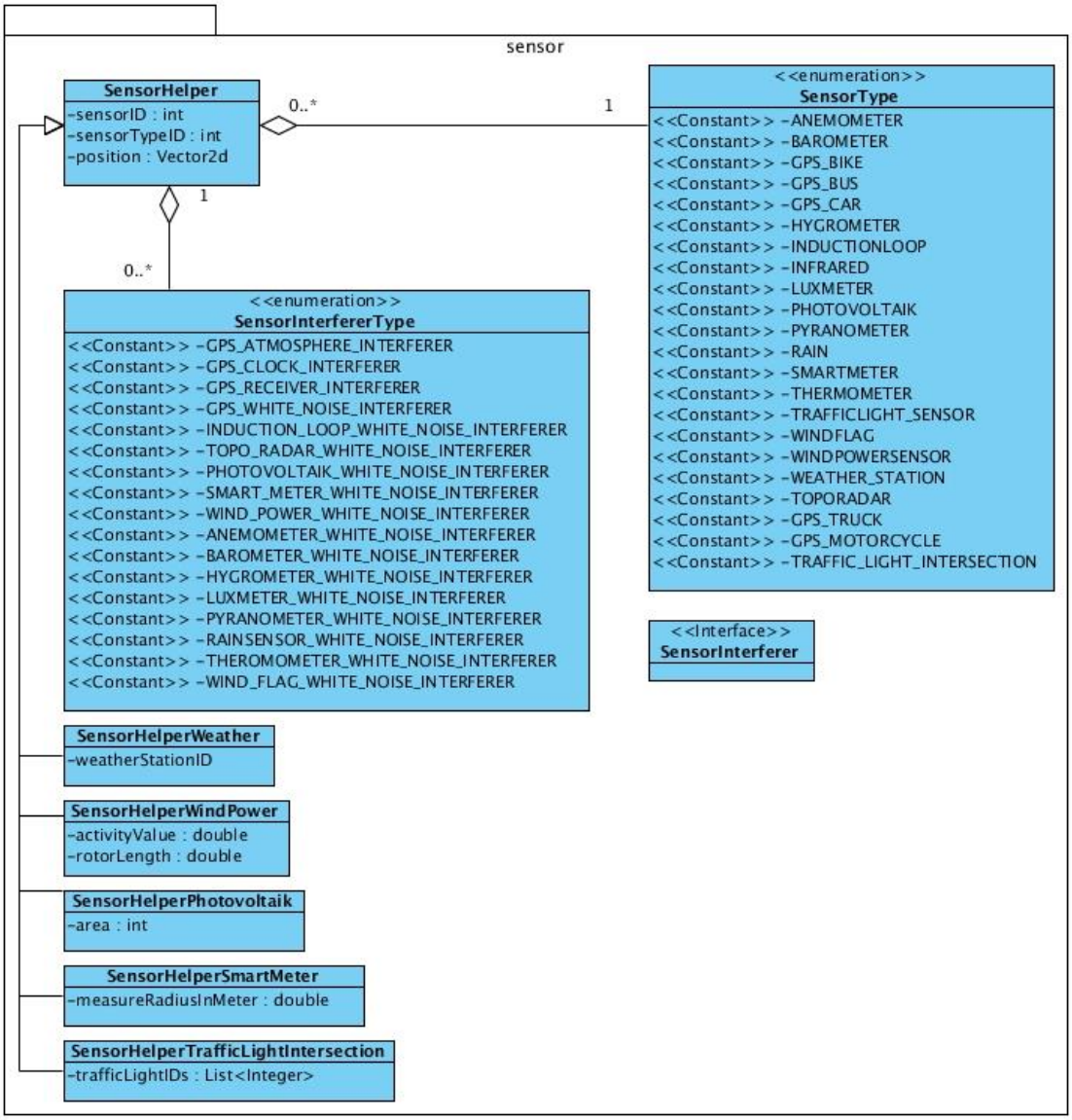


Figure 60: Shared Package Sensor

The package *sensor* is displayed in Figure 60 and has all needed models for sensors.

- **SensorHelper**: A SensorHelper is used to create a concrete sensor in the simulation. The SimulationController will send it to every *SensorManagerController* until one can create it without a *SensorException*. Every sensor has a unique ID, position, a type and a list of interferer. The type values are in the *SensorType* enum and represent the concrete type of the sensor. The interferers are set in a list of *SensorInterfererType*.
- **SensorType**: SensorType is an enum with an entry for every sensortype available in the simulation. It also contains possible interferer for the sensor types and the concrete classes of *SensorHelper*. E.g. a smart meter can only be created with a *SensorHelperSmartMeter*.

- **SensorInterfererType**: Every sensor can have different interferer. These interferers are listed in the SensorInterfererType enum. The *SensorType* enum says if a SensorInterfererType is suitable for a sensor.
- **SensorHelperWeather**: For weather sensors the SensorHelperWeather must be used, because a weather station ID is needed.
- **SensorHelperWindPower**: The SensorHelperWindPower extends the normal *SensorHelper* by values for activity and rotor length.
- **SensorHelperPhotovoltaik**: A SensorHelperPhotovoltaik needs an area in meter.
- **SensorHelperSmartMeter**: SensorHelperSmartMeter needs a radius in meter, which will be observed.
- **SensorHelperTrafficLightIntersection:** To create traffic lights, the SensorHelperTrafficLightIntersection proves a list with IDs.

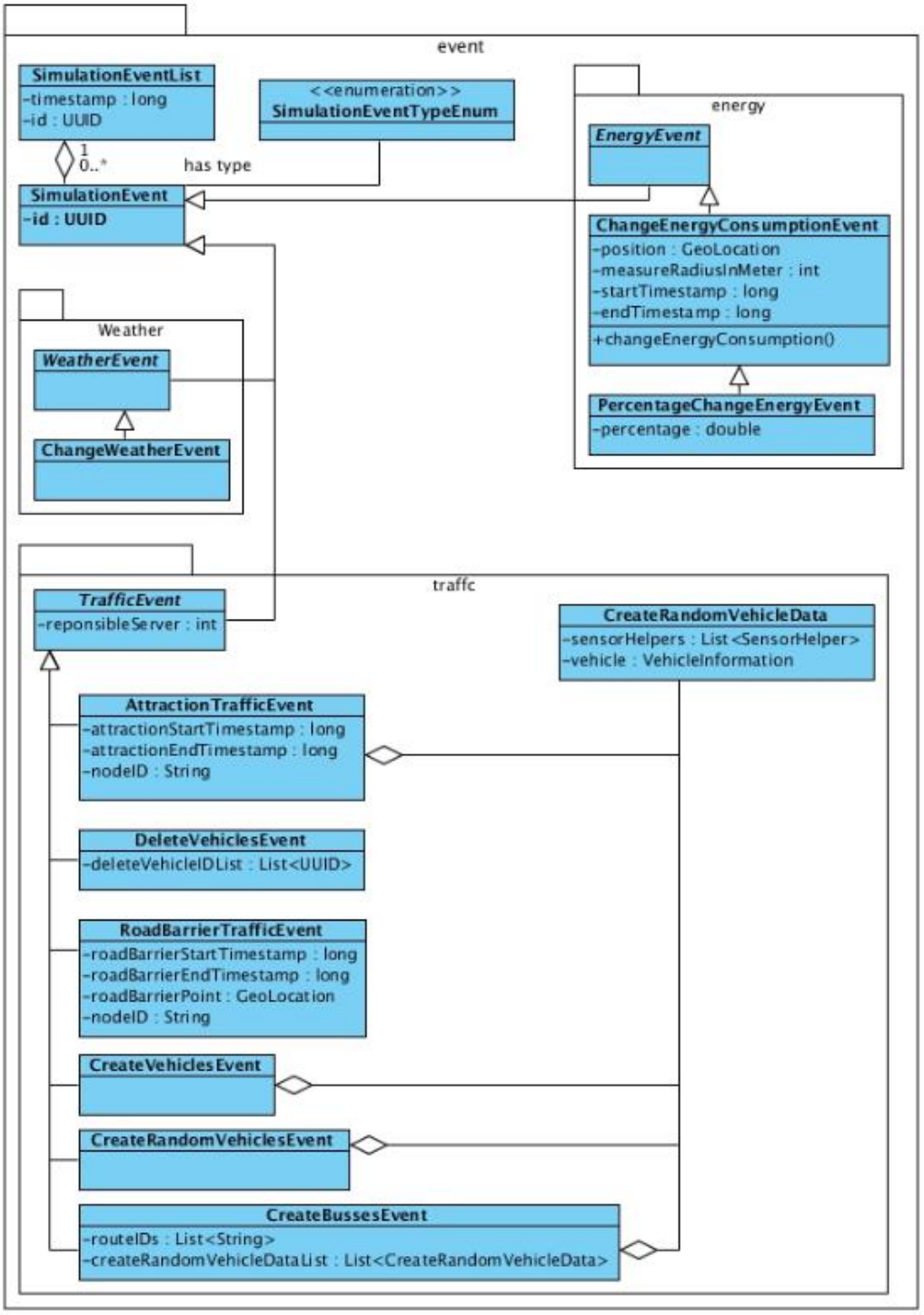


Figure 61: Shared Package Event

The package event, displayed in Figure 61, proved the needed event models.

- **SimulationEvent:** The SimulationEvent is the base class for very event in the simulation. It has a unique ID and a type of *SimulationEventTypeEnum* to allow serialization with GSON and also faster event detection.
- **SimulationEventList:** A *SimulationEventList* wraps several *SimulationEvents* with a timestamp. This class is used to avoid unnecessary network traffic.

- **SimulationEventTypeEnum:** The SimulationEventTypeEnum has a field for every possible event and allows GSON to serialize and deterialize the events.
- **EnergyEvent:** The EnergyEvent is the superclass for every energy event.
- **ChangeEnergyConsumptionEvent:** ChangeEnergyConsumptionEvent is the superclass of every energy event, which will change the current energy consumption value by a specified method.
- **PercentageChangeEnergyEvent:** The PercentageChangeEnergyEvent will change the energy consumption in a specified radius by a specified percentage value. E.g. a concert can increase the current energy consumption by 200%.
- **WeatherEvent:** The WeatherEvent is the superclass for every WeatherEvent.
- **ChangeWeatherEvent:** The ChangeWeatherEvent provides information about how to change the weather.
- **TrafficEvent:** TrafficEvent is the superclass for every traffic event. It extends the *SimulationEvent* by a traffic server ID.
- **CreateRandomVehicleData:** A random vehicle can only be created with information about the used *SensorHelper* and *VehicleInformation*.
- **AttractionEvent**: An AttractionEvent will create an attraction on a given point, in a specific time window and with given vehicles.
- **DeleteVehicleEvent**: The DeleteVehicleEvent is used to remove vehicles.
- **CreateVehiclesEvent:** If the trip of a vehicle is known, the CreateVehicleEvent can be used.
- **RoadBarrierTrafficEvent:** A RoadBarrierTrafficEvent will block a street node during a specific time window.
- **CreateRandomVehicleEvent**: If the trips of the vehicles are unknown, the CreateRandomVehicleEvent can be used.
- **CreateBussesEvent**: The CreateBussesEvent will create busses for specific routes.

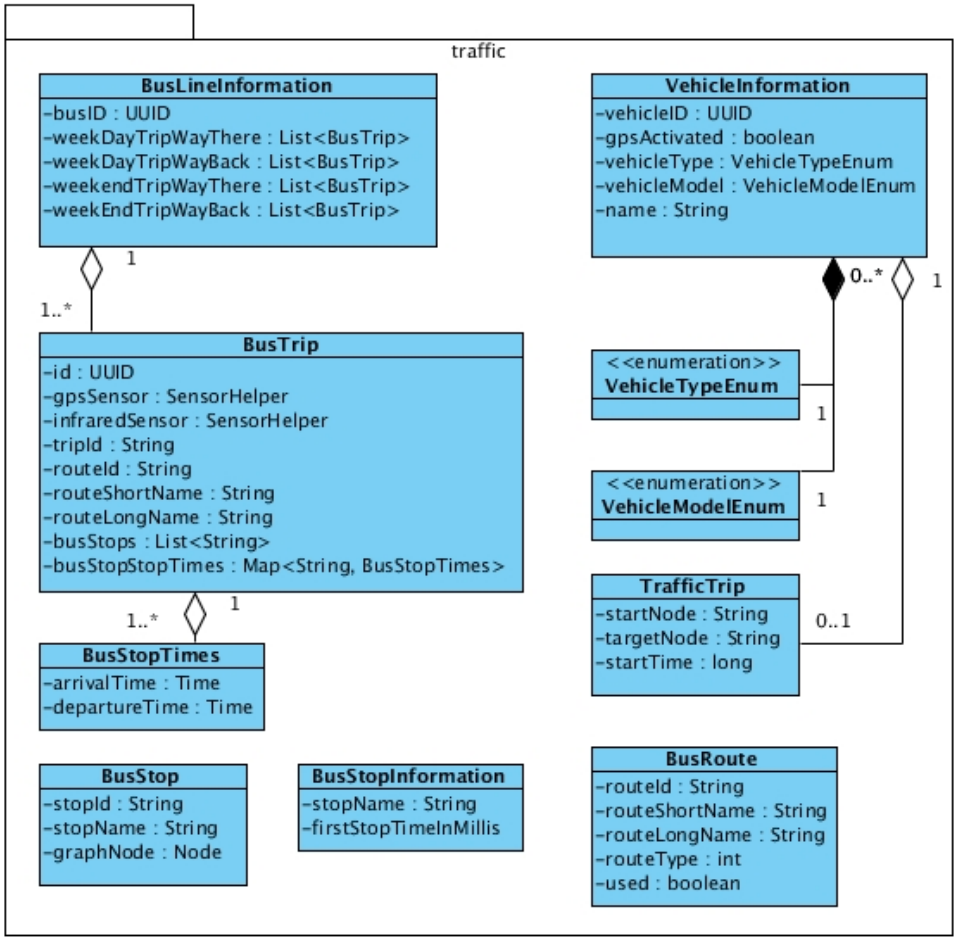


Figure 62: Shared Package Traffic

The package traffic is shown in Figure 62 and gives an overview about the traffic models.

- **VehicleInformation**: The class VehicleInformation gives needed information to create vehicles.
- **VehicleTypeEnum**: The VehicleTypeEnum holds every possible vehicle in the simulation.
- **VehicleModelEnum**: The VehicleModelEnum has several fields for every *VehicleTypeEnum*. It gives concrete models for the vehicle types. E.g. the vehicle type can be car and the model can be VW Golf.
- **TrafficTrip**: A traffic trip has a start node, a target node and also a start timestamp.
- **BusLineInformation**: The BusLineInformation class holds every *BusTrip* for a bus line.
- **BusTrip**: A BusTrip is a concrete bus route for one bus with sensors.
- **BusStopTimes**: The class BusStopTimes wraps the times of a *BusTrip*.
- **BusStop:** A BusStop has a name, an ID and a node in the *Graph*.
- **BusStopInformation:** Instances of BusStopInformation provide the stop name and the first stop in miliseconds.

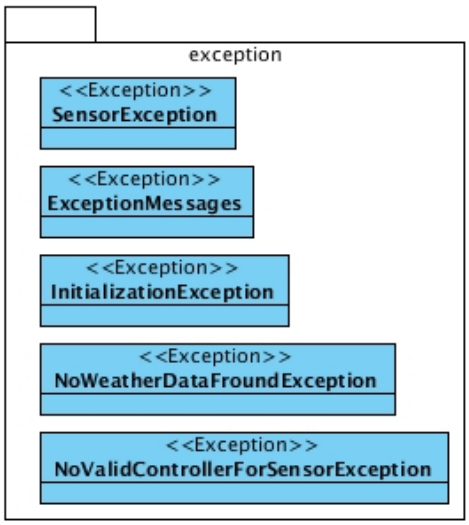


Figure 63: Package Shared Exception

The package exception, shown in Figure 63, contains every nonstandard exception of the prototype.

5.2.9.2 Configuration

There are no configurationally parts in the shared package.

5.2.9.3 Extension Points

This section gives an overview about extension points of the shared package.

- New sensors can be added by adding a new *SensorType*. If the *SensorHelper* class is not suitable for the new sensor, it can be extended. The *DefaultSimulationController* will call the *createSensors* method for every *SensorManagerController*, until one creates it without an exception.
- New *SimulationEvents* can be added by adding a type in *SimulationTypeEnum* and extending the *TrafficEvent*, *WeatherEvent* or *TrafficEvent*.The sensors for vehicles can only be created by firing a *TrafficEvent* with *VehicleInformation*, because these sensors belong to vehicles and vehicles can also exists without sensors or deactivated sensors.
- New vehicles can be added in *VehicleTypeEnum* and possible models for the new type can be added in *VehicleModelEnum*.

5.2.9.4 Exception Handling

There is no exception handling needed in the shared package.

5.2.10 Component: SimulationController

The Simulation Controller is superior to all other controllers in the architecture. Every other controller can be controlled by using the Simulation Controller. Because of that, it is the only touch point of the *Simulation Control Center* and it is one of the two touch points of the *Operation Center*, the other is the *data stream management system*.

5.2.10.1 Scheme

The following UML class diagrams show the architecture of the Simulation Controller. Only the important method signatures are shown and the parameter and return types are hidden. The description is divided into API and implementation. The API of the Simulation Controller is shown in Figure 64.

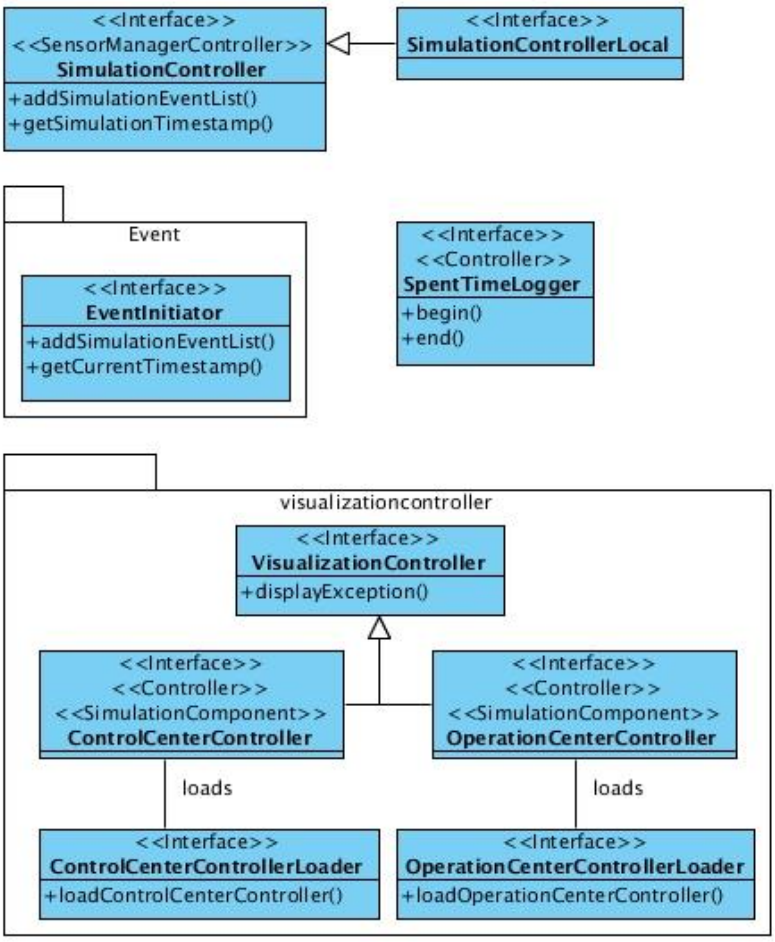


Figure 64: Simulation Controller API

- **SimulationController**: The SimulationController is the highest controller in the architecture. Its methods will be called by the *Simulation Control Cente*r and it has to distribute the calls to the lower level controllers. After the start, it needs update every other controller in every simulation step with the current timestamp and *SimulationEvents*.
- **SimulationControllerLocal**: SimulationControllerLocal is the local view of the *SimulationController*. It is needed by EJB to distinguish between local and remote calls.

- **EventInitiator:** The EventInitiator can be used by the *SimulationController* to handle the simulation updates. It can receive new events via *addSimulationEventList()* and will provide the current timestamp and needed *SimulationEvents* to the other controllers in every simulation step.
- **SpentTimeLogger:** To measure the needed computing time of any component, the SpentTimeLogger can be used. In this prototype it is used for the scalability test. For this purpose it measured the computing time of every update iteration in the DefaultEventInitiator.
- **VisualitionController**: The VisualizationController is the interface for the *ControlCenterController* and *OperationCenterController*. It allows sending exceptions to the *Simulation Control Center* and *Operation Center*.
- **ControlCenterController**: A ControlCenterController gives important simulation information to the *Simulation Control Center*. E.g. the current timestamp to allow *SimulationEvents* during runtime.
- **ControlCenterControllerLoader**: The ControlCenterControllerLoader can load an instance of *ControlCenterController*. This can be used to allow different bindings. E.g. if the *Simulation Control Center* is not needed.
- **OperationCenterController**: The *Operation Center* can receive simulation information via an OperationCenterController. This is needed to inform about init, start, stop or new sensors.
- **OperationCenterControllerLoader**. The OperationCenterControllerLoader can load an instance of *OperationCenterController*. Similar to the *ControlCenterControllerLoader*, this can be used to ignore the *Operation Center*.

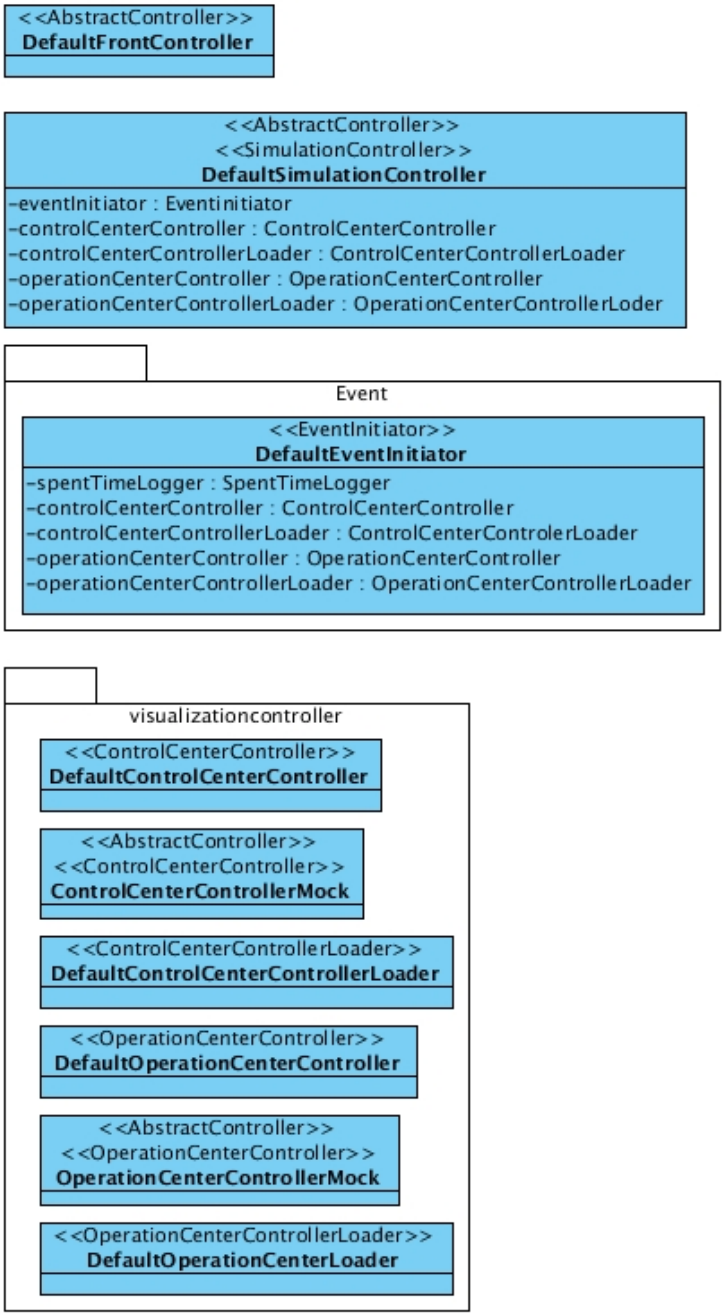


Figure 65: Simulation Controller Implementations

The Figure 65 gives an overview about the implementations of the Simulation Controller project.

- **DefaultFrontContoller**: The DefaultFrontController exists on every server. It inits the *SensorRegistry* and the *ServiceDictionary*.
- **DefaultSimulationController**: The default implementation of the *SimulationController* is the DefaultSimulationController. It inits, starts and updates the lower level controllers. The DefaultSimulationController uses an instance of *EventInitiator* to outsource the update part. To create new sensors it will iterate over the list of known *SensorManagerControllers* and call the *createSensor* functions until one does not response with a *SensorException*. This allows an easy extension of the sensors without modifying the high level controllers.

- **DefaultEventInitiator**: When the DefaultEventInitiator is started, it will start a thread and updates every controller in every simulation step with *SimulationEventLists*. It will only send the needed *SimulationEvents*, to avoid unnecessary network traffic. E.g. the *TrafficController* will never receive an *EnergyEvent*.
- **DefaultControlCenterController**: The DefaultControlCenterController is a link to the *Simulation Control Center*. It provides information about start, stop, updates or errors via HTTP.
- **ControlCenterControllerMock**: ControlCenterControllerMock is a *ControlCenterController* without any behavior. It can be used to ignore the *Simulation Control Center*.
- **ControlCenterControllerLoader**: The ControlCenterControllerLoader can load the correct *ControlCenterController*. If the *Simulation Control Center* is mocked, it will provide an implementation of *CentrolCenterControllerMock*, and if not it will provide an instance of *ControlCenterController*. It uses EJB to dissolve the dependencies. Other implementations can be set without changing the class.
- **DefaultOperationCenterController**: The DefaultOperationCenterController is a link to the *Operation Center*. It provides information about start, stop, updates or errors via HTTP.
- **OperationCenterControllerMock**: OperationCenterControllerMock is an*OperationCenterController* without any behavior. It can be used to ignore the *Operation Center*.
- **OperationCenterControllerLoader**: The OperationCenterControllerLoader can load the correct *OperationCenterController*. If the *Operation Center* is mocked, it will provide an implementation of *OperationCenterControllerMock*, and if not it will provide an instance of *OperationCenterController*. It uses EJB to dissolve the dependencies. Other implementations can be set without changing the class.

5.2.10.2 Configuration

This section gives an overview about the configurations of the Simulation Controller.

- The connection to the Simulation Control Center can be mocked by setting the following line in the global "*simulation.conf*":
  `simulation.configuration.server.ccc.mock=false`

- The connection to the Operation Center can be mocked by setting the following line in the global "*simulation.conf*":
  `simulation.configuration.server.occ.mock=false`

5.2.10.3 Extension Points

The Simulation Controller does not allow many extensions, because its implementation is very generic. But it is possible to use other implementations of every component and set the dependencies via EJB-annotations.

5.2.10.4 Exception Handling

Non-fatal exceptions of other controllers are handled by the Simulation Controller. Fatal-exceptions will be shipped to the *Simulation Control Center* and *Operation Center* and will stop the EJB Container.

5.2.11 Component: StaticSensor Controller

The *StaticSensorController* Component manages all static sensors, which comprise the energy and the weather sensors of the simulation. Like the other controllers this component has the same life cycle: It can be started, paused, stopped and resumed. Specific sensors and associated interferers can be created to manage the energy and weather simulation.

5.2.11.1 Scheme

The following UML class diagram shows the API architecture of this component with his classes and packages and their interaction points. The parameter and return types are hidden.

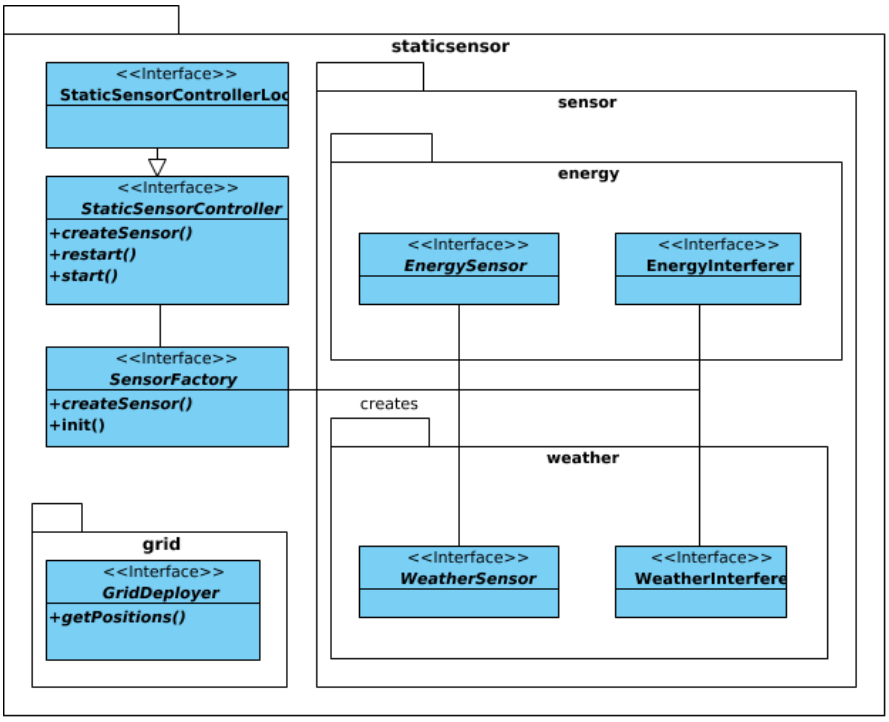


Figure 66: API scheme of the component Static Sensor Controller

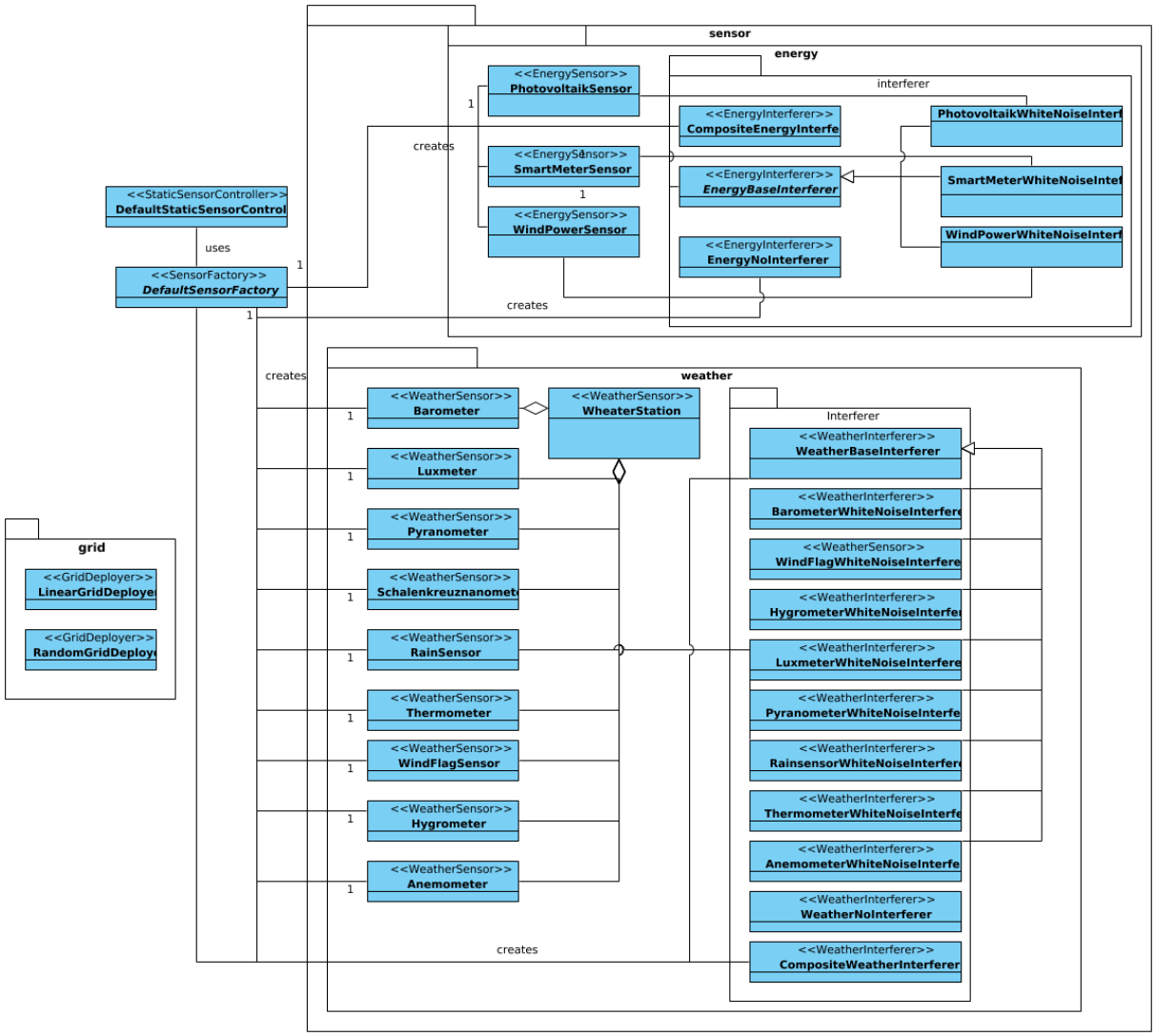


Figure 67: Implementation scheme of the component Static Sensor Controller

The UML class diagram above illustrates the implementation of the API. The interfaces belonging to the classes are shown as stereotypes. Like the previous UML diagram the method signatures are abridged.

The main interaction points of the component *StaticSensorController* are on one hand the implementation of the corresponding interface named *DefaultStaticSensorController*and on the other hand the implementation of the *SensorFactory* named *DefaultSensorFactory*. The interface *StaticSensorController* extends the *AbstractController* and thereby implements the basic controller functions. This includes the methods init, start, stop, reset and update. It also implements the *StaticSensorControllerLocal* which is necessary for the EJB. Furthermore the implementation creates and deletes sensors through the *DefaultSensorFactory*.

The *DefaultSensorFactory* implements the *SensorFactory*. The *DefaultSensorFactory* creates all sensors of the three basic sections: energy, weather and traffic. It also creates all kind of interferers, which generates realistic sensor data for instance by adding environmental influences. The weather

and energy interferers are the *WeatherBaseInterferer*, the *CompositeWeatherInterfer*, the *EngeryBaseInterferer* and the *CompositeEngeryInterferer*. The base interferers have several implementations, which typify different kind of errors for specific sensors such as the *PhotovoltaikWhiteNoiseInterferer*. This interferer generates generally low errors upon photovoltaic sensor output.

Referring to the types of sensors the energy sensors consist of photovoltaic, wind power and smart meter sensors. However the weather sensors consist of anemometers, barometers, hygrometers, luxmeters, pyranometers, rain sensors, thermometers and wind flag sensors. A weather station combines all of the weather sensors to represent a measurement station.

Other parts of the *StaticSensorController* component are the *LinearGridDeployer* and the *RandomGridDeployer* that implements the *GridDeployer*. The main purpose of these classes is to assist in deploying the sensors upon the graph by returning the positions of those. The basic approach for both implementations is mainly the same. First of all the graph will be disassembled into different sections. Afterwards the positions for the sensors will be calculated through different algorithms. The distribution takes place in a linear and in a random way, specified in the corresponding class.

5.2.11.2 Configuration

The standard interferer configurations can be changed with the aid of the properties files, which are located in the resource folder. All files begin with the expression *interferer_* and end with the file extension *.properties*. For instance, the file *interferer_anemometer_whitenoise.properties* configures the default values for the maximum change amplitude and the probability if the interferer gets active. The reading of the files takes place in the abstract **BaseInterferer* classes. The *EnergyBaseInterferer* class is such an example.

5.2.11.3 Extension Points

A new interferer can be added to the component with the following steps:

1. Implement the new class and realize the specific interface or expand the abstract class with the name **BaseInterferer*.
2. Create an individual properties file in the resource folder for reasons of maintenance.
3. Add the new interferer to the specific sensor in the method *createSensor(…)* which is located in the implementation class *AbstractStaticSensorFactory* of the interface *SensorFactory*.

5.2.11.4 Exception Handling

The main exceptions being thrown are *IllegalArgumentExceptions* which are used to intercept incorrect arguments such as parameters being null.

While creating a sensor the *DefaultStaticSensorController* catches exceptions which are thrown if the process gets interrupted or the process gets aborted by throwing an exception while attempting to

retrieve the result. The deletion of a sensor will also be checked. Errors occurring while the process is being executed will be caught and treated.

The controller acts like all controllers. For that reason exceptions appear if the controller cannot change to an illegal state. The state transitions are illustrated by Figure 55 on page 106.

5.2.12 Component: Traffic

The traffic aspect of the simulation (creating traffic sensors as well as produce traffic sensor data and process incoming traffic events) is handled by the traffic component. Its API is formed by the TrafficController, an extension of the Controller interface.

5.2.12.1 Scheme

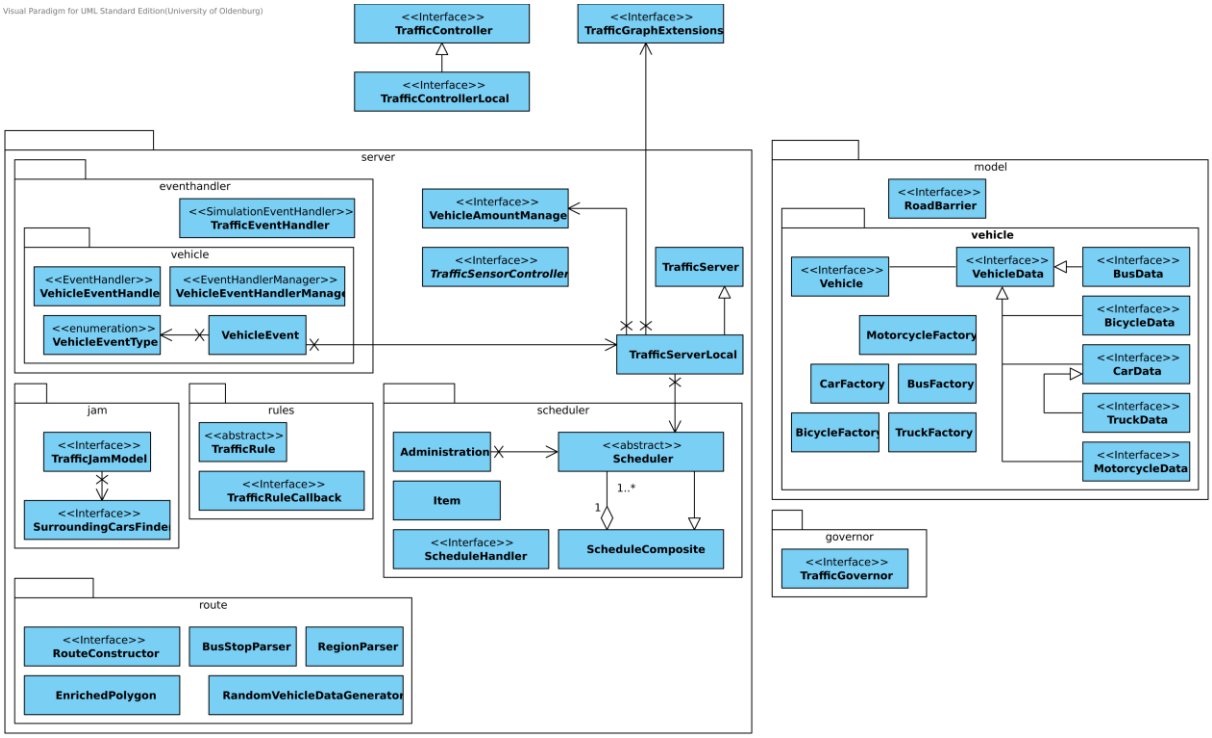


Figure 68: Overview of the traffic component

Figure 68 shows an overview of the traffic package in which the *TrafficController* resides. The traffic package is divided into 3 major sub packages model, server and governor. Further on a description of the classes and packages is given.

Traffic Package

The root package exposes the API of the traffic component of the simulation, the *TrafficController* (its remote view as well as remote view).

To access all extensions of the graph nodes and edges at one point, a class called *GraphExtensions* is defined. Here are all methods put together concerning for instance the lengths of edges and positions of nodes.

Model Package

The model package consists mainly of entities modeling moveable traffic objects, for example vehicles like cars and bicycles. However there are also helper classes that are used to store certain information regarding to the traffic graph. For instance the *RoadBarrier* knows which edges in the

traffic graph building a road are blocked and can't be used anymore to calculate a path between two nodes.

Governor Package

This package is used to determine a suitable number of vehicles that are active at a given timestamp for each vehicle type. The concrete implementation FuzzyTrafficGovernor internally uses fuzzy logic to calculate this number.

Server Package

In order to exploit the benefits of a distributed architecture it is necessary to divide the simulated city in distinct areas when it comes to the traffic aspect of the simulation. Therefore the *TrafficController* is used to manage multiple so called *TrafficServer* each taking care one of these distinct areas. Everything that is needed to simulate or generate traffic data is located in this package.

Server.EventHandlerPackage

This package consists of classes that extend the *EventHandlerManager* and *EventHandler* as described in chapter 5.2.8 (Component: Services). They are used by the implementation of the *TrafficServer* (*DefaultTrafficServer*) to process incoming *TrafficEvents* or raise own *VehicleEvents*. *VehicleEvents* are only used internal and are fired during the update process of a vehicle, e.g. when a vehicle has passed a node. These handlers can be used to add functionalities easily to the *TrafficServer* that concern a single vehicle. For instance induction loops are placed on nodes in the traffic graph. Whenever a vehicle passes a node the appropriate handler for this event is called. If an induction loop is found by the handler it is told to increment the amount of the passed vehicles by one.

Server.Jam Package

The implementation of the *TrafficServer* uses a model to simulate a traffic jam. To exchange this model easily an API has been formed, the *TrafficJamModel*, which uses the *SurroundingCarsFinder* to find relevant vehicles surrounding a specific one.

Server.Scheduler Package

In this package resides the Scheduler used by the implementation of the *TrafficServer* to schedule vehicles whose departure times lie in the future. A schedule plan consists of two lists containing expired Items and not expired Items. On every update step the *TrafficServer* iterates through the expired items to update the vehicles that are already on the road.

A *TrafficServer* may use a Scheduler for each type of vehicle. The *SchedulerComposite* can be used to manage these Schedulersas one unit.

The *ScheduleHandler* can be used to inform components about scheduling events (scheduling of a new item or removal of an item).

To prohibit other components to manipulate the schedule plan during the iteration through the expired items the *Administration* class along with read and write permissions was introduced. The owner of the *Administration* object belonging to a *Scheduler* instance gets privileged access and is therefore the only one who can change its access type (READ or WRITE).

Server.Rules Package

In this package the TrafficRule as well as the TrafficRuleCallback classes are located. The base functionality of a *TrafficRule* is that a *Vehicle* registers on a node's corresponding *TrafficRule* when it passes the same node. *TrafficRule* then takes responsibility to let that vehicle pass through the node (in and out), by invoking methods *onEnter* and *onExit* of a passed *TrafficRuleCallback*.

Server.RoutePackage

The Server.Route package contains all necessary classes for generating the routes of busses and all other vehicle types. The routes of the busses are generated on the basis of the given timetables parsed from the GTFS-files. For this purpose the *BusStopParser* parses the bus stops and adds them to the graph at first. Following this, calculating the intermediate routes between the bus stops and combining them to a whole route will compute the route.

Generating the routes for the other vehicle types is more complex. First the *RandomVehicleDataGenerator* generates for each vehicle all required data for creating an individual route and returns a *TrafficTrip* object. This object contains the *startTime*, the *startNode* and the *targetNode*. The *RandomVehicleDataGenerator* can use different parameters for creating a *TrafficTrip* for different types of vehicles. For example: For creating a car trip on workdays with a probability of 60% the trip is a work trip and otherwise a free time trip. The *startTime* of a free time trip can be at any time of a day while the *startTime* of a work trip is within the traffic peak hours. In addition to that the *startNode* and *targetNode* are chosen from corresponding subsets of all possible nodes. For this purpose the *RegionParser* parses the entire graph and extracts all nodes within residential areas for storing them as *homeNodes*. Furthermore all nodes within commercial or industrial areas are stored as *workNodes*. The route of a work trip is calculated between a home node and a work node (or vice versa) while the route of a free time trip is calculated between two home nodes.

Implementation Details

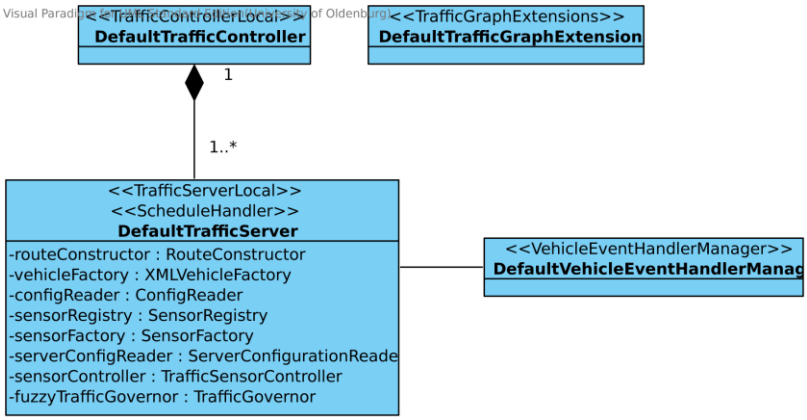


Figure 69: Implementation of the traffic component

Figure 69 shows the implementation of the *TrafficController* and *TrafficServer*. It can be seen that the *DefaultTrafficController* consists of one or more *DefaultTrafficServer* which depends on multiple components, e.g. the *SensorFactory* to create traffic sensors or the *ServerConfigReader* to find other *TrafficServer* on remote hosts. The *DefaultVehicleEventHandlerManager* is the implementation of the *VehicleEventHandlerManager* and handles, as described above, internal events concerning a single vehicle.

The implementation of *GraphExtensions* is *DefaultGraphExtensions*. In Addition to that the interface *TrafficGraphExtensions* extends *GraphExtensions* by comprising traffic specific data like traffic rules or sensors. The implementation of *TrafficGraphExtensions* used in the application is *DefaultTrafficGraphExtensions*. Figure 70 shows the architecture of *GraphExtensions*. Only a few methods are shown in order to demonstrate how things work.

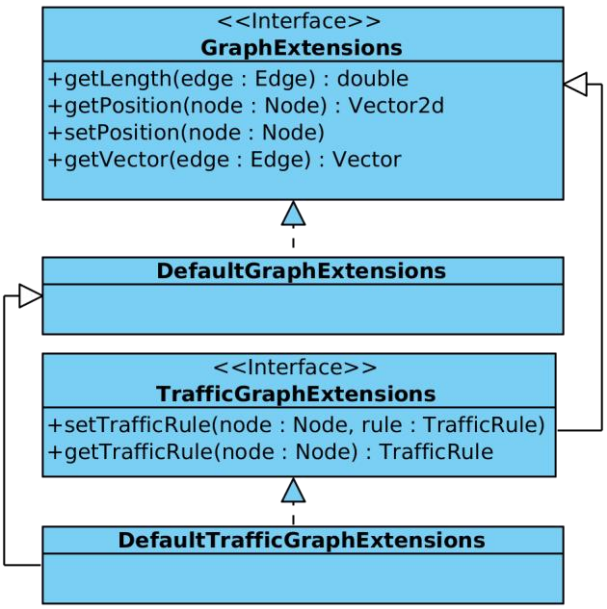


Figure 70: GraphExtensions

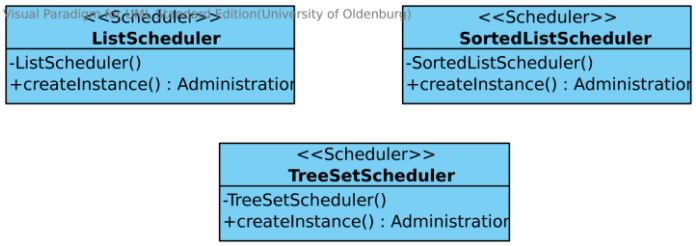


Figure 71: Various implementations of the Scheduler

Figure 71 represents various implementations of the *Scheduler*. In Chapter 5.3 (Implementation Progress) is an overview with the performance impact of each implementation.

The scheduling of vehicles can be reduced into two sub problems. One the one hand the amount of active vehicles has to be determined. On the other hand the vehicles must be scheduled. The calculation how many vehicles are driving at a given time is the task of *TrafficGovernor*. To calculate a realistic number its implementation *FuzzyTrafficGovernor* uses Fuzzy Logic which input variables include weather, time of day, time of week and time of year. A Governor is used by a *VehicleAmountManager*. Its implementation *DefaultVehicleAmountManager* takes a number from the Governor and schedules the vehicles based on this number. Figure 72 shows the relations between the components in short.

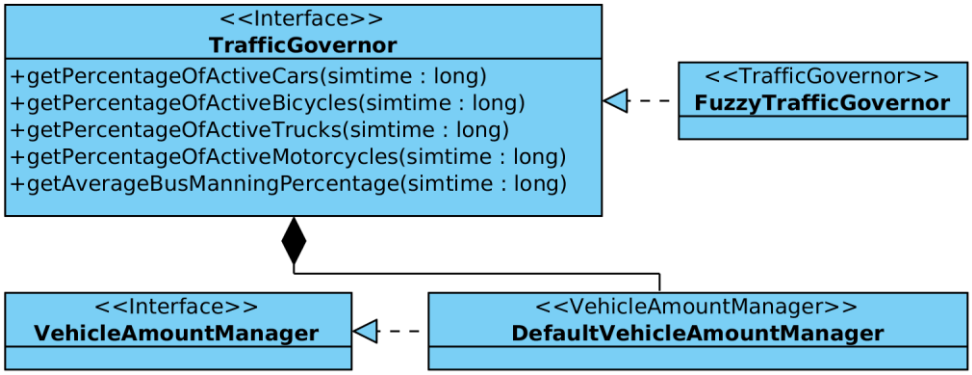


Figure 72: Scheduling of Vehicles with fuzzy logic

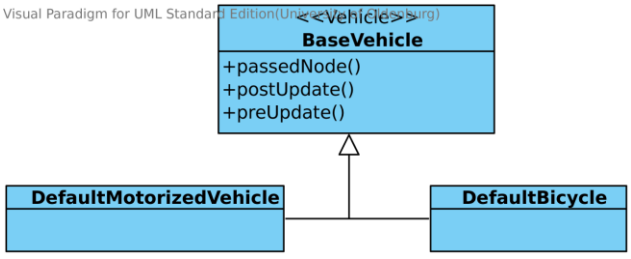


Figure 73: Implementation of the Vehicle interface

Figure 73 shows the implementation of the *Vehicle* interface. *BaseVehicle* is the default implementation and is responsible for the correct positioning of a vehicle on the traffic graph. In

addition it provides useful methods to be overridden by derived subclasses like *DefaultMotorizedVehicle* and *DefaultBicycle*. These methods are automatically invoked whenever the vehicle passes a node (passedNode), has been updated (postUpdate) or will be updated (preUpdate).

As the name suggests *DefaultMotorizedVehicle* is the implementation of the Vehicle interface for motorized vehicles like cars, trucks and motor bicycles. *DefaultBicycle* on the contrary is the implementation for bicycles that follow other rules than the motorized vehicles.

Vehicles are created through the various vehicle factories. Implementations of these factories, which realize the factory pattern[24], are the XML-based factories. The following table shows the six XML-based factories with their implemented interfaces:

Implementation	Used Interface	Tag
XMLBicycleFactory	BicycleFactory	Bicycle
XMLBusFactory	BusFactory	Bus
XMLCarFactory	CarFactory	Car
XMLMotorcycleFactory	MotorcycleFactory	Motorcycle
XMLTruckFactory	TruckFactory	Truck
XMLVehicleFactory	BicycleFactory, BusFactory, CarFactory, MotorcycleFactory, TruckFactory	

Table 5: XML-based factories of the Traffic component

Every mentioned XML-based factory reads a XML file and creates for every tag element with the listed name in the tag column of the Table 5 an individual vehicle type. The following xml code, used by the *XMLCarFactory*, creates a vehicle type for a car named BMW 120i:

```
<?xml version="1.0"?>
<vehicles>
      <cars>
            <car>
                  <typeid>car001</typeid>
                  <wheelDistanceWidth>1535</wheelDistanceWidth>
                  <wheelbase1>2690</wheelbase1>
                  <wheelbase2>0</wheelbase2>
                  <length>4324</length>
                  <width>1765</width>
                  <height>1421</height>
                  <weight>1360</weight>
                  <power>75</power>
                  <maxSpeed>195</maxSpeed>
                  <axleCount>2</axleCount>
                  <description>BMW 120i</description>
            </car>
      </cars>
</vehicles>
```

[24] http://en.wikipedia.org/wiki/Factory_method_pattern

The *XMLVehicleFactory* can be shown as a super class of all the five other XML-based factories because this factory contains all of the others and forwarded only the XML file to them. The advantage is that the XML file needs to be read by the factory only one time. More information about the configuration and structure of the XML files are listed in the next subsection. According to the XML factories, there are *ExtendedXML*Factories* like the *ExtendedXMLCarFactory* that add to the normal XML factory traffic rule behavior. Hence the vehicles can react on the various implemented traffic rules, which are described in the following.

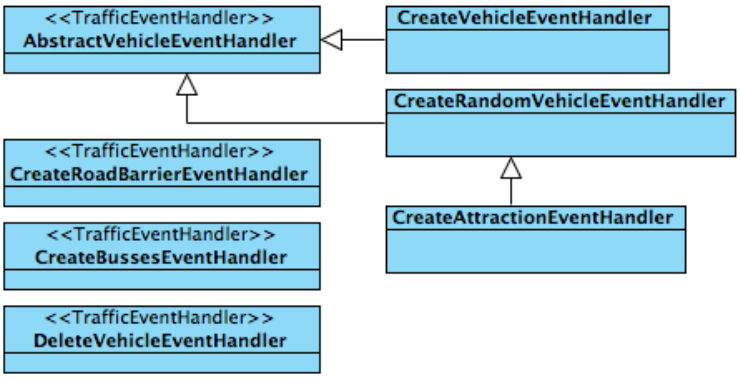


Figure 74: The implemented TrafficEventHandlers

There are two different types of event handlers. The first type of event handlers is the *TrafficEventHandler* (see Figure 74). These handlers will be executed by events that affect the whole TrafficServer and not only one vehicle.

The following Table 6 describes the six implemented *TrafficEventHandlers* with the corresponding traffic event and a brief explanation. The Simulation Control Center throws the traffic events and it will be transferred to the *TrafficController*. The controller passes the traffic event to all the *TrafficServers*. The *TrafficServer* has instantiated automatically all used *TrafficEventHandlers*. The *TrafficEventHandler* corresponding to the traffic event handles the new event.

Traffic event handler	Description	Traffic event
CreateVehicleEventHandler	It creates new vehicles with a given start position and destination.	CREATE_VEHICLES_EVENT
CreateRandomVehicleEventHandler	It creates new vehicles with random start positions and destinations.	CREATE_RANDOM_VEHICLES_EVENT
CreateAttractionEventHandler	It creates to a defined time new vehicles with a given destination. After a defined duration, it creates new vehicles with the given destination as start position and random target positions.	ATTRACTION_TRAFFIC_EVENT
CreateRoadBarrierEventHandler	It creates a time limited road barrier for a given position. All driving vehicles that will pass this position get a new route.	ROAD_BARRIER_TRAFFIC_EVENT
CreateBussesEventHandler	It creates the given busses with their routes.	CREATE_BUSSES_EVENT
DeleteVehicleEventHandler	It deletes the given vehicles on the traffic server.	DELETE_VEHICLES_EVENT

Table 6: Description of the TrafficEventHandlers

The other type of event handlers is the *VehicleEventHandler* (see Figure 75), which observes the process of one individual vehicle.

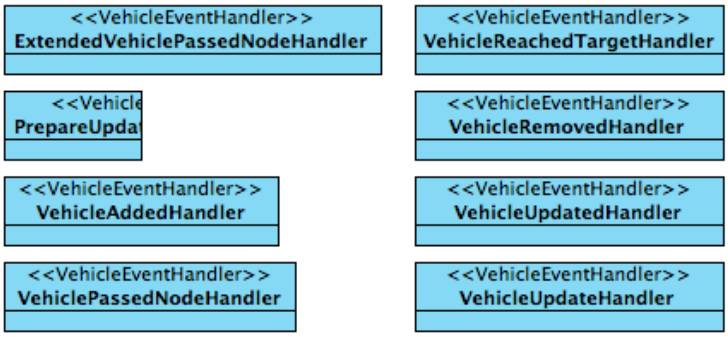


Figure 75: The implemented VehicleEventHandlers

Each handler acts to a specific event as described in the following table. Note many of the handlers do nothing. There were created to provide extension points for new features.

Vehicle event handler	Description	Traffic event
VehicleUpdateHandler	Event is fired to perform an update on a vehicle. The handler is responsible for the update process.	VEHICLE_UPDATE
VehicleUpdatedHandler	Event is fired after all vehicles have been updated.	VEHICLE_UPDATED
VehicleRemovedHandler	Event is fired when a vehicle has been removed from the local TrafficServer.	VEHICLE_REMOVED
VehicleReachedTargetHandler	Event is fired when a vehicle has reached its target.	VEHICLE_REACHED_TARGET
VehiclePassedNodeHandler	Event is fired when a vehicle has passed a node in the traffic graph. This implementation of the handler is used to count vehicles that passed a induction loop and let passengers enter or leave the vehicle if it is a bus.	VEHICLE_PASSED_NODE
ExtendedVehiclePassedNodeHandler	An extension to the prior event handler. It introduces traffic rules to the traffic like run abounds.	VEHICLE_PASSED_NODE
VehicleAddedHandler	Event is fired when a vehicle has been added.	VEHICLE_ADDED
PrepareUpdatingVehiclesHandler	Event is fired before the vehicles in the simulation get updated one by one.	PREPARE_UPDATING_VEHICLES

Table 7: Description of the VehicleEventHandlers

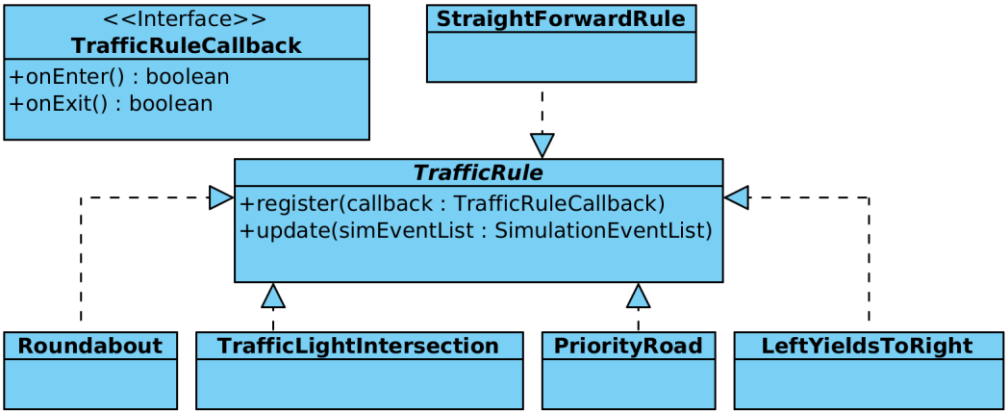


Figure 76: TrafficRules

Figure 76 shows all implemented traffic rules. As described above the base functionality of a *TrafficRule* is that a *Vehicle* registers on a node's corresponding *TrafficRule* when it passes the same node but how a *TrafficRule* acts depends on the respective implementation. A *Roundabout* for instance lets only a defined number of vehicles in so that some have to wait in a queue. When vehicles are let out others may let in. A StraightForwardRule in contrast would let vehicles through immediately.

5.2.12.2 Configuration

The number of how many vehicles should be driving at a given timestamp is currently calculated by *FuzzyTrafficGovernor*. It uses fuzzy rules specified in *src/main/resources/fuzzy.fcl*. The easiest way to customize the calculation is to change the rules in the several rule blocks. It has to be borne in mind to satisfy FCL's syntax. It is also possible to create a new FCL-file in this directory. To use this file the static string variable *FILE_PATH* in *FuzzyTrafficGovernor* has to be adjusted.

All used *TrafficEventHandlers* are listed in file *eventhandler.conf*, which is located in the resource folder. To load the *TrafficEventHandler* automatically, the complete class path with the package description had to be noted.

All used *VehicleEventHandlers* are listed in file *updatehandler.conf*, which is located in the resource folder. To load the *VehicleEventHandler* automatically, the complete class path with the package description had to be noted.

The standard interferer configurations can be changed with the aid of the properties files, which are located in the resource folder. All files begin with the expression *interferer_* and end with the file extension *.properties*. For instance, the file *interferer_gps_atmosphere.properties* configures the default values for the maximum change amplitude and the probability if the interferer gets active. The reading of the files takes place in the abstract **BaseInterferer* classes. The *GpsBaseInterferer* class is such an example.

The structure of the XML file that is read by the XML-based factory has to be like the following scheme:

```
<?xml version="1.0"?>
<vehicles>
      <cars>
            <car>
                  <typeid>...</typeid>
                  ...
            </car>
            ...
      </cars>
</vehicles>
```

The individual attributes of the vehicle type are located under the specific vehicle type tag[25]. The tag named *typeid* has to be unique because it is the identifier of the vehicle type. Every vehicle type can contain other attributes. The standard attributes of every vehicle type can be found in the XML file *defaultvehicles.xml* that is located in the resource folder.

5.2.12.3 Extension Points

A new *TrafficEventHandler* can be added to the component with the following steps:

1. Implement the new class and realize the interface *TrafficEventHandler*.
2. Append the new *TrafficEventHandler* to the file *eventhandler.conf* in the resource folder. Remove, if necessary, the handler that was responsible for this type of events before.
3. Throw the new *TrafficEvent* by the Simulation Control Center. If there is no corresponding *TrafficEvent*, add it to the enum *SimulationEventTypeEnum*.

A new VehicleEventHandler can be added to the component with the following steps:

1. Implement the new class and realize the interface *VehicleEventHandler*.
2. Append the new *VehicleEventHandler* to the file *updatehandler.conf* in the resource folder. Remove, if necessary, the handler that was responsible for this type of events before.
3. Let the DefaultTrafficServer raise the event if it already exists. If notthe corresponding code has to be included in the DefaultTrafficServer. In addtion the new *VehicleEvent* has to be added to enum *VehicleEventType*.

A new interferer can be added to the component with the following steps:

1. Implement the new class and realize the specific interface or expand the abstract class with the name **BaseInterferer*.
2. Create an individual properties file in the resource folder for reasons of maintenance.

[25]The tag names are illustrated by the Table 5 on page 63.

3. Add the new interferer to the specific sensor in the method *createSensor(…)* which is located in the implementation class *AbstractTrafficSensorFactory* of the interface *SensorFactory*.

Due to the fact that the XML-based vehicle factories read a XML file, it is very easily to add new vehicle types to the simulation. There is no need to change to source code. The factories offers the new vehicle types automatically if a new element with the particular tag name[25] was appended to the root element. Detailed information about the correct structure is described in the previous subsection.

A new TrafficGovernor can be added to the component with the following steps:

1. Implement a new class realizing the interface TrafficGovernor.

2. To use the new implementation copy the annotation of FuzzyTrafficGovernor to the new implementation and remove it from FuzzyTrafficGovernor.

5.2.12.4 Exception Handling

The traffic component provides no additional exception handling.

5.2.13 Component: Weather

The *Weather* component is the interaction point of the simulation that handles all weather issues. As well as the other controllers this component also has the same life cycle: It can be started, paused, stopped, and resumed. Specific weather events can update the component and the loaded data. If the component starts, it reads the needed weather information from the deposited database and after that it can respond all weather requests from the other components. The modification of the data from the database plays an important role. Therefor, a lot of weather modifiers, for instance the *CityClimateModifier*, are implemented.

5.2.13.1 Scheme

The following UML class diagram shows the API architecture of this component with his classes and packages and their interaction points. Only the important method signatures are shown. The parameter and return types are hidden.

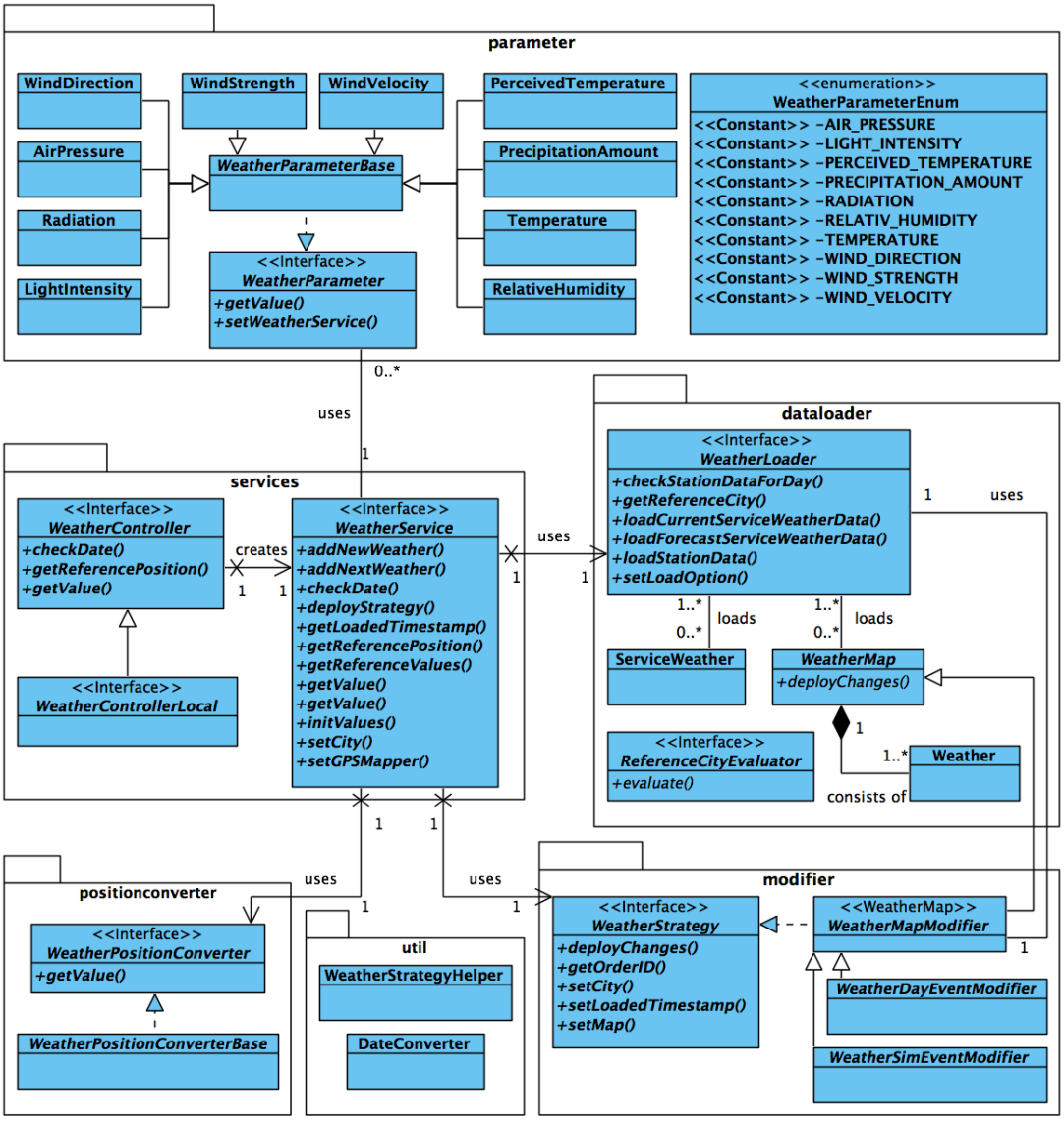


Figure 77: API scheme of the component Weather

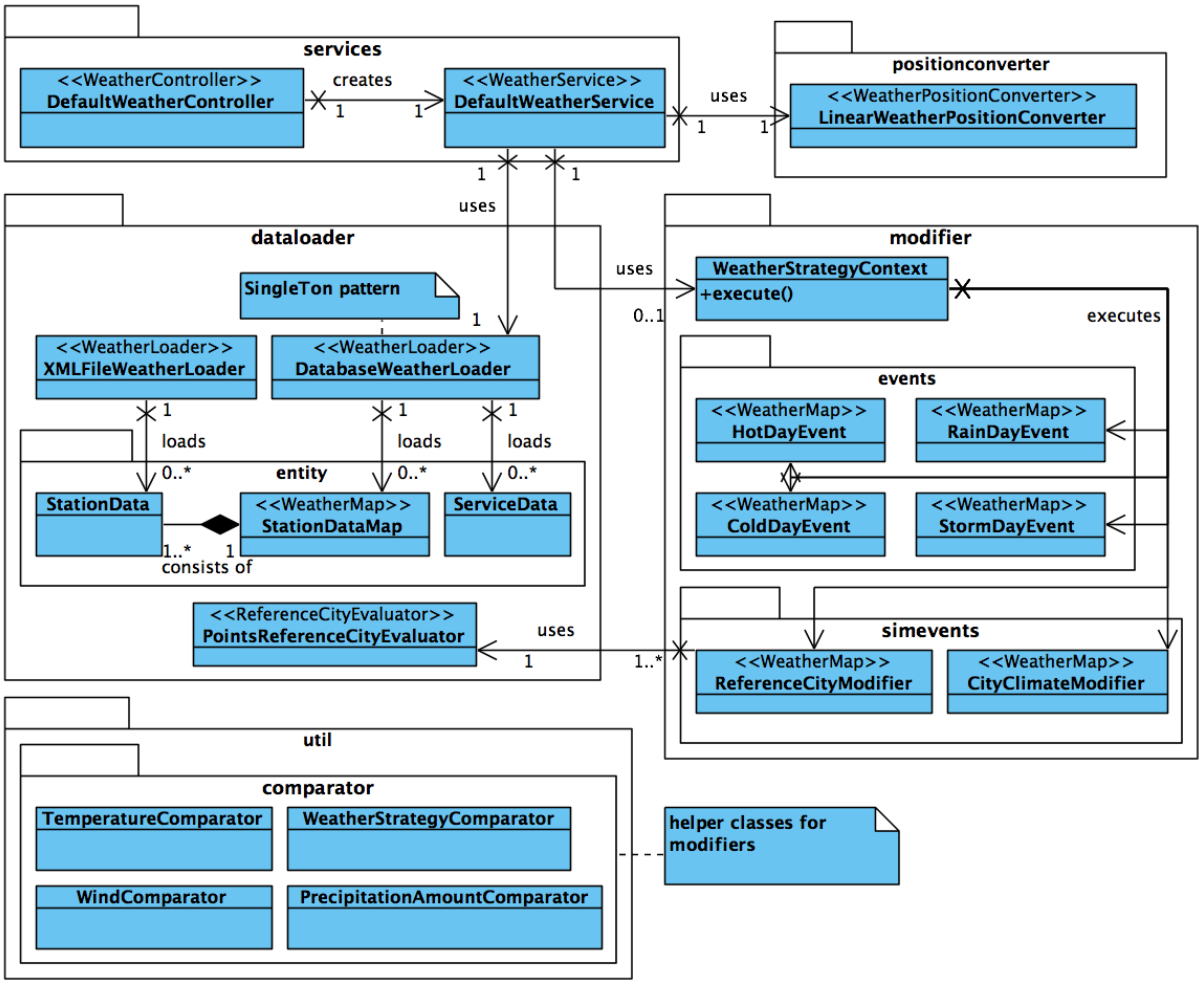


Figure 78: Implementation scheme of the component Weather

The UML class diagram above illustrates the implementation of the API. The interfaces belonging to the classes are shown as stereotypes. Like the previous UML diagram the method signatures are abridged.

The main interaction point of the component Weather is the interface *WeatherController* that represents the weather environment controller of the simulation. Its public methods can be called by other components. Therefore, the controller is implemented as an EJB. For one thing the controller sends all requests to the interface *WeatherService* and the responses back to the other components, for another thing it inherits from the generic *Controller* interface and implements its state transitions[26]. So this controller acts as intermediary between components of the simulation and the *WeatherService*. The interface *WeatherControllerLocal* can be used for local communication.

The *WeatherService* coordinates the weather parameter requests and provides access with the help of the *WeatherLoader* to the needed weather information. An implementation of that interface is the *DefaultWeatherService*. This implementation realizes a synchronization concept between the loading

[26]The state transistions are illustrated by Figure 55 on page 59.

of new data and responding to value requests. Due to the fact that the component should not allocate too much space during runtime, it provides only a specific amount of modified weather information. If there is a request for weather parameter values, which cannot be calculated from the provided data, the *WeatherService* try to get the needed data automatically. Only one thread can get the new information from the database. At that moment the other requests have to wait till the new data are available. Many threads can request specific weather parameter values simultaneously.

At the time of the *WeatherService* creation, the class loads the weather parameter by the help of the enum *WeatherParameter* automatically. There are eleven weather parameters available. Some of them are calculated by other weather parameters. These parameters will be cached by the *WeatherService* for another request to reduce the CPU processing. An example is the parameter *WindStrength*.

The aim and the purpose of the interface *WeatherLoader* is to get weather information from a previously described source during runtime. It is also implemented as an EJB. The *XMLFileWeatherLoader* and the *DatabaseWeatherLoader* are the two implementations, which are available. The configurations of the classes are explained in the next subsection. Their tasks are to load weather station values that are represented by the class *StationData* and to load weather service values as well as reference cities that are used by the *ReferenceCityModifier*.

The class *Weather* represents every single weather information entry to a particular timestamp. This class combines all available parameter values and is stored in a *WeatherMap*. The *WeatherMap* serves as the root for the weather modifiers that are all derived from the class *WeatherMapModifier*. For that reason the decorator pattern[27] is realized. Furthermore, the decorators can also function as strategy because the strategy pattern is implemented[28]. The different weather modifiers are separated into two groups: Some are derived from the class *WeatherDayEventModifier* and others expand the class *WeatherSimulationEventModifier*. The event modifiers, which expand the *WeatherDayEventModifier*, can be activated more than one time for each day during a simulation runtime. For instance, a *HotDayEvent* changes the weather values in such a manner that the simulation can request weather values of a warm day. The simulation event modifiers, which expand the *WeatherSimulationEventModifier*, can be activated only once and are always active for the complete simulation runtime. The *ReferenceCityModifier*or the *CityClimateModifier*are examples of this modifier type.

The *WeatherService* provides only weather information for one reference point of the simulation city. Therefore, the interface *WeatherPostionConverter* makes methods available that convert the reference value to a modified value on another position of the used graph. The implementation *LinearWeatherPositionConverter* calculates the values on a linear base away from the reference point.

[27] http://en.wikipedia.org/wiki/Decorator_pattern
[28] http://en.wikipedia.org/wiki/Strategy_pattern

5.2.13.2 Configuration

The main configuration points of the component are the three files in the *resource* folder. The file *jndi.properties* describes the configuration to the database. In particular, the file includes the settings of the Java Persistence API (JPA) and the database authorization. To change the database, one has to change the database driver in this file. Notice that the XML file *persistence.xml* in the *META-INF* folder configures the JPA process of this component.

The file *weatherloader.properties* explains the configuration of the JPA in more detail. It indicates the used persistence unit, the default file path of the *XMLFileWeatherLoader* and the Java classes for the JPA entities concerning the *StationData*. Notice that the database tables must be created before the *Weather Service* has started for the first time, otherwise an exception will occur. Moreover, the database tables must have the schemes like illustrated in Figure 17 and Figure 18. If there are changes in the scheme, one has to change also the JPA entities in the package *dataloader.entity* or the properties in the file *weatherloader.properties* regarding the classes for *StationData*.

The file *weather_decorators.properties* describes the default properties for the implemented modifier. If no parameters are given in the constructor of an implemented modifier, the standard properties of the file will be used.

5.2.13.3 Extension Points

Based on the using of many interfaces and abstract classes, the component has a lot of extension points. At first, creating a new controller that realizes the interface *WeatherController* can change the current implementation of the *WeatherController*. Besides, the current implementation of the *WeatherService* can be changed by creating a new service that realizes the interface *WeatherService*.

The interface *WeatherLoader* gives the opportunity to implement other classes, which provide the weather information. For instances, the class *XMLFileWeatherLoader* is an alternative implementation to the standard *DatabaseWeatherLoader* class. The reference to the loader can be found in the constructor of the *DefaultWeatherService*.

One has to realize the implement the interface *WeatherParameter* or inherit from the abstract class *WeatherParameterBase* to add new weather parameters. The abstract class has methods to communicate with a given *WeatherService*. Besides, the enum *WeatherParameterEnum* must have a new entry to the new class. The *DefaultWeatherService* will add the new parameter automatically.

The *LinearWeatherPositionConverter* is one implementation of the interface *WeatherPositionConverter*. To add new converter, one has to realize this interface or inherit from the abstract class *WeatherPositionConverterBase* that makes general methods regarding the used *GPSMapper* available. The reference to the converter can be found in the constructor of the *DefaultWeatherService*.

As already described, the modifier for weather data play an important role. So, it is very easy to add a new modifier:

1. First of all, the new class has to inherit from the abstract classes *WeatherDayEventModifier* or *WeatherSimulationEventModifier*.
2. The enum *WeatherEventEnum* has to be appended by a new entry.
3. The method *createStrategyFromEnum(...)* in the *DefaultWeatherController* has to be modified.

The *ReferenceCityModifier* uses the interface *ReferenceCityEvaluator* to find the best city for the simulated city during runtime. To change this evaluator, one has to realize the mentioned interface and change the reference in the modifier.

5.2.13.4 Exception Handling

If the *WeatherController* tries to start, it checks the availability of weather information regarding the start date of the simulation. An exception will occur if the component cannot connect to the database or found any information. It is mandatory that there is weather data to the first simulation day.

Due to the fact that the component should not allocate too much space during runtime, it provides only a specific amount of modified weather information. If there is a request for weather parameter values, which cannot be calculated from the provided data, the *WeatherService*try to get the needed data automatically. Requests for dates that not bear on the simulation date interval be ignored by the component and an exception occurs during runtime.

The controller acts like all controllers. For that reason exceptions appear if the controller cannot change to an illegal state. The state transitions are illustrated by Figure 55 on page 71.

5.2.14 Component: Weather Collector

The *Weather Collector* component serves as a helper for the *Weather* component. It collects weather information from particular weather stations and weather services, preprocesses the collected information by harmonizing the data, and saves the reviewed weather information to a database. The simulation, especially the Weather component, can use the weather data thence.

5.2.14.1 Scheme

The following UML class diagram shows the general architecture of the component with its classes and packages and its interaction points. Only the important method signatures are shown. The parameter and return types are hidden.

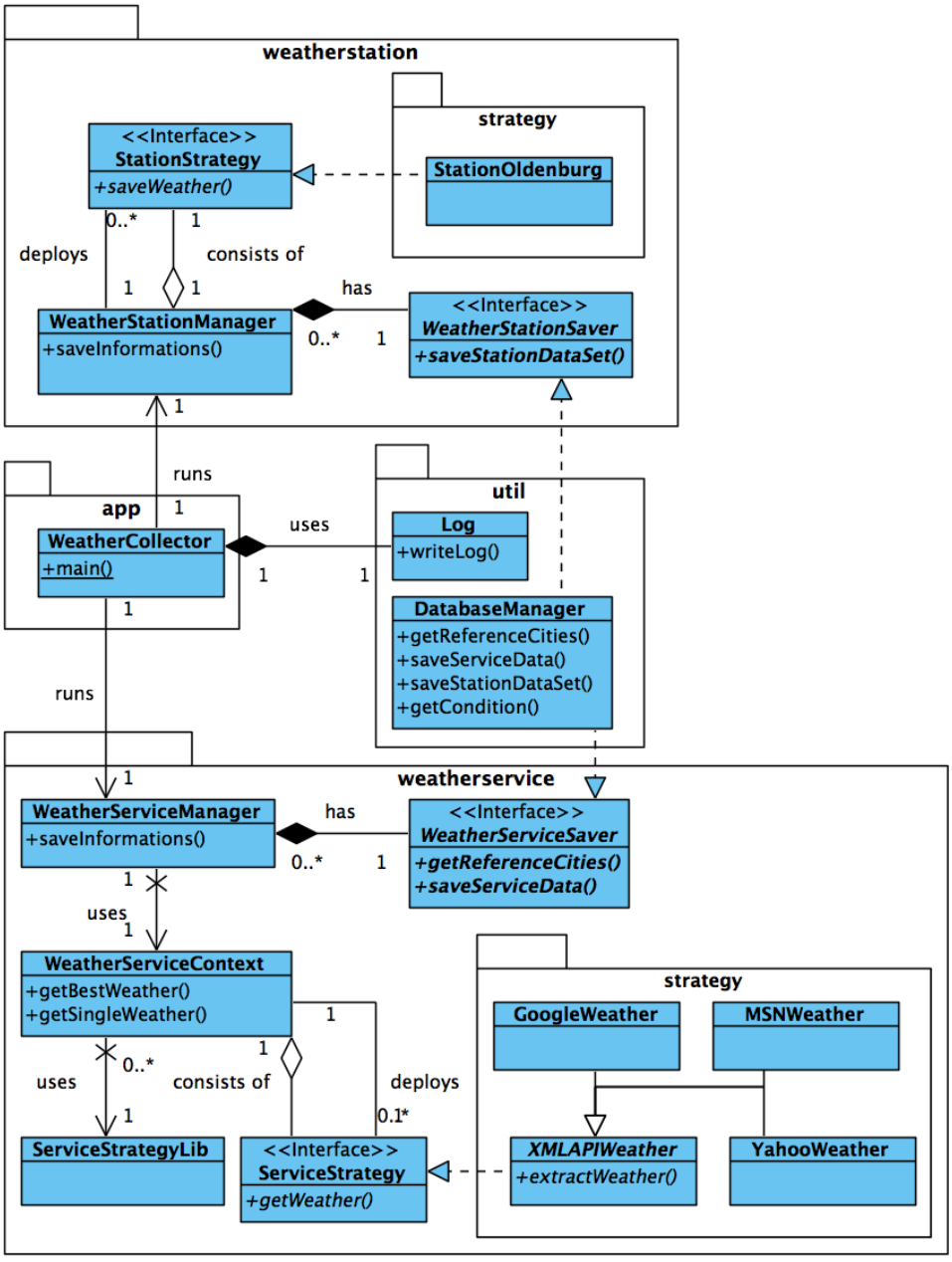


Figure 79: Scheme of the component Weather Collector

The starting point of the Weather Collector can be found in the package *app*. The class *WeatherCollector* implements the main method of the project and instantiates the two process managers *WeatherServiceManager* and *WeatherStationManager*, which execute the individual strategies. Furthermore, the main class creates a log file with the help of the class *Log* depositing in the package *util*.

The class *WeatherStationManager* instantiates the individual weather station strategies. In the first instance, it reads the XML file *weatherstations.xml* and creates the weather station strategies. These strategies have to implement the interface *StationStrategy*. The strategy pattern is implemented[29]. The class *StationOldenburg* realizes this interface and reads weather information from a weather station of the Carl von Ossietzky University Oldenburg[30]. The next step of the process manager is to execute all weather station strategies step by step. After an execution of a strategy the specified instance of the interface *WeatherStationSaver* will be called to persist the new data. The class *DataManager* in the package *util* realizes this interface and saves the information to the database, which is described in the file *database.properties*.

The database scheme to persist the weather station data is described in section 4.8.2.1 on page 40. The *StationData (normal)* saves the exact weather information collected by the weather station strategies. A database trigger calculates the aggregate weather data, represented by the entity *StationData (aggregate)*. Two implementations of the described trigger for a MySQL database and for an IBM DB2 are listed in the appendix. The trigger starts after a new tupel was inserted into *StationData (normal)*. It generates average values for the specific day and time of the previous years and creates a new tupel in *StationData (aggregate)*.

The class *WeatherServiceManager* has the same scheme like the *WeatherStationManager*. However, this class processed all weather service strategies that are listed in the XML file *weatherservices.xml*. Moreover, with the help of the classes *WeatherServiceContext* and *ServiceStrategyLib* the process manager can combine all results of the individual weather service strategies to one complete result. By doing this the absence of certain weather parameters and the last updateof the service data play an important role. Every weather service strategy will be executed for every reference city, which are achieved from the specified realization of the interface *WeatherServiceSaver*. Before the new information to a reference city is stored in the database, old data to the collected dates and cities will be removed. Three weather service strategies are implemented: The class *GoogleWeather* that reads data from the Google Weather API, the class *MSNWeather* that uses the MSN Weather API, and the class *YahooWeather* that connects to the Yahoo Weather API.

[29] http://en.wikipedia.org/wiki/Strategy_pattern
[30] More information is described in the chapter 4.8.2.1 "Weather Data“ on page 42.

5.2.14.2 Configuration

The main configuration points of the component are the three files in the *resource* folder. The file *database.properties* describes the configuration to the database. In particular, the file includes the settings of the Java Persistence API (JPA) and the database authorization. To change the database, one has to change the database driver in this file. Notice that the database tables must be created before the *Weather Collector* has started for the first time, otherwise an exception will occur. Moreover, the database tables must have the schemes like illustrated in Figure 17 and Figure 18. If there are changes in the scheme, one has to change also the JPA entities in the package *model*.

The XML file *weatherservices.xml* contains the available weather service strategies with their complete class signature. The XML file *weatherstations.xml* contains the available weather station strategies with their complete class signature.

5.2.14.3 Extension Points

There are four destined extension points. Two of these are designed to add new storing classes for the weather stations and weather services. To add new storing classes for weather stations one has to implement the interface *WeatherStationSaver* and change the reference in the class *WeatherStationManager*. To add new storing classes for weather services one has to implement the interface *WeatherServiceSaver* and change the reference in the class *WeatherServicesManager*.

Another way to easily extend the Weather Collector is to add new weather stations strategies to the XML file *weatherstations.xml*. The scheme of this XML file is like the following:

```
<?xml version="1.0"?>
<strategies>
      <strategy>de.pgalise.weathercollector.weatherstation.strategy.S
tationOldenburg</strategy>
</strategies>
```

To append new strategies one has to put a new <strategy> tag between the <strategies> tag. In the text content of the new tag it contains the whole class signature (with class name). Moreover, the class signature has to be known by the class path. The Weather Collector will instantiate the new strategies automatically during runtime. Notice that data from another weather station has to be stored separately in the database. A possible solution is to extend the weather station table of database scheme, which is illustrated in the design document in Figure 17 on page 41, with a new attribute. This attribute can be a strategy or city identifier to identify each weather station strategy.

The last destined extension point is the possibility to add new weather service strategies. It can be done with the help of the XML file *weatherservices.xml* in the same way like the XML file for weather stations. The scheme of the XML file is the same.

5.2.14.4 Exception Handling

First of all, the component will not start if it cannot write a log file. The software creates a new text file called *weatherLog.txt* for logging. Another critical issue is the connection to the database and the persistence mechanism. If the component cannot read data from the database (e.g. weather service information) or save new data into the database, it will not start.

The processing of the individual strategies does not influence the overall processing of the component. If one strategy cannot finish his work successfully (for example an exception occurs), the Weather Collector will continue with the next strategy.

5.2.15 Component: IBM Cognos

The *IBM Cognos* component is the Business Intelligence platform to display the data generated by the simulation. Using a dashboard all dimensions, energy, weather und traffic, are visualized.

5.2.15.1 Scheme

The foundation for IBM Cognos was in the relational database IBM DB2. Using the High Quality Function interface of the SimulationController, data is saved into the table "pgalise.hqf_data". The interface receives a data stream from the data stream management system and inserts it directly into the table. In IBM Cognos an ODBC Connection to this database is used to select data from the running simulation. Therefore IBM Cognos includes a package "ALISE", containing the database connection and a query for the specified table.

```
Select
      HQF_DATA.ID,
      HQF_DATA.SIMULATIONTIMESTAMP,
      HQF_DATA.SENSORTYPEID,
      HQF_DATA.AMOUNT
From
      [IBM DB2 DS].HQF_DATA as HQF_DATA
```

The package is used in IBM Cognos Report Studio to design a dashboard containing information of the three dimensions energy, traffic and weather. The following figure shows the dashboard in design time.

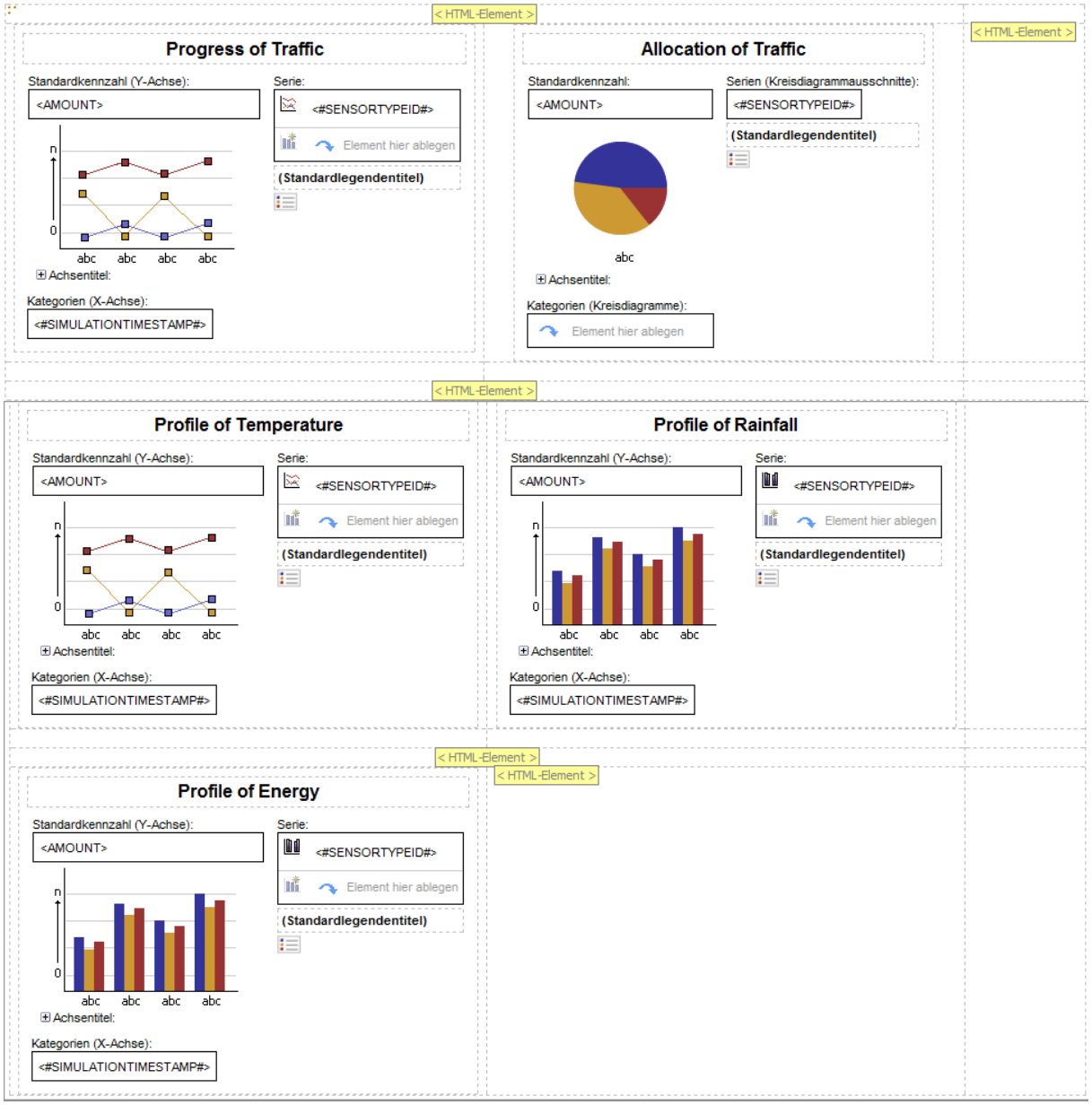


Figure 80: IBM Cognos Report in Report Studio

5.2.15.2 Configuration

There is no direct configuration in this component.

5.2.15.3 Extension Points

There are two main extension points: the first one is in the package “ALISE”. The user can add a data source like a flat file or another database to enrich the analysis. Furthermore there could be additional queries to other tables to extend the detail level of the diagrams. Secondary further reports can be defined to satisfy the users eager for knowledge. All extensions in this component require more data in databases or other persistent data sinks. Thus the simulation and the data stream component need to, on the one hand create and on the other hand pass through the desired data for analyzing purposes.

5.2.15.4 Exception Handling

IBM Cognos has its own Exception Handling. Every part is validated by its creation or change.

5.3 Implementation Progress

As already described in section 3.1, the project group uses the software development process model SCRUM. This decision enabled a flexible and agile implementation process, in which the software expanded sprint by sprint. The continuous integration as well as the test-driven development with the help of a high number of JUnit tests assures a high quality of software implementation. Besides, a continuous grown test protocol recorded all test results. The following curve chart illustrates the JUnit test development by indicating the number of JUnit classes and the related date. The illustrated dates refer to the dates of the test protocol. The test protocol is described in the appendix on page 199.

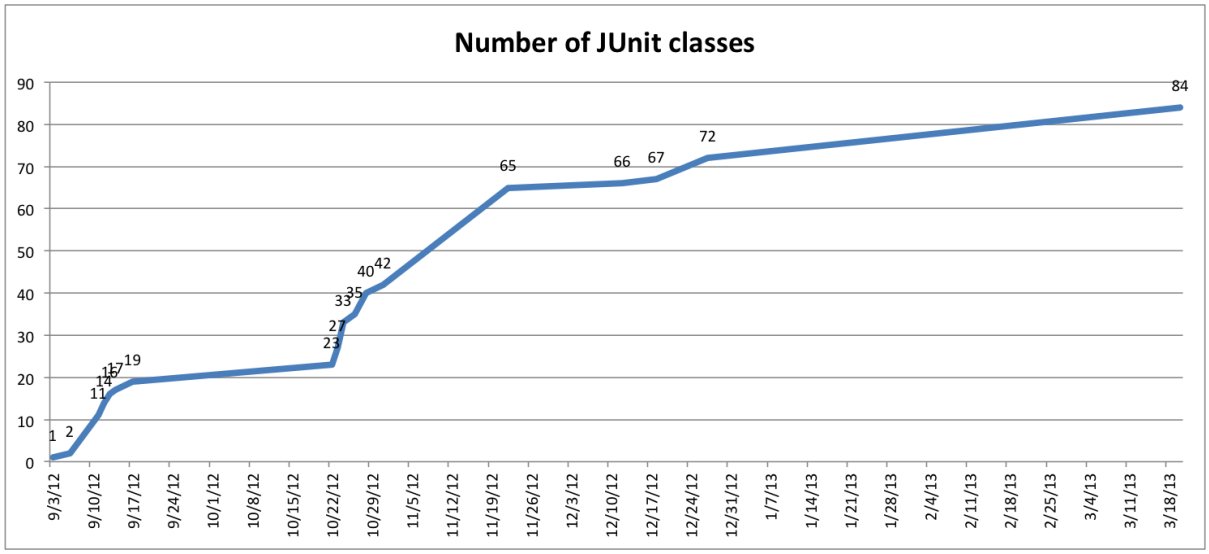


Figure 81: Development of the number of JUnit classes

Another quality tool of the implementation process was Jenkins, "an extendable open source continuous integration server" [Jenkins]. This tool observed the building process of all Java projects and executed daily all JUnit test classes. As a result it was very easy to identify main issues as well as bugs of the software application. Furthermore, unsuccessful test results were published every day by e-mail to all of the group members. Based on that information the related programmer could find and correct the errors faster. Installed plugins for static source code analysis helped the project group to fix bugs and other discrepancies. The following static analysis tools were used:

1. FindBugs plugin[31]
2. Checkstyle plugin[32]
3. PMD plugin[33]

[31] https://wiki.jenkins-ci.org/display/JENKINS/FindBugs+Plugin
[32] https://wiki.jenkins-ci.org/display/JENKINS/Checkstyle+Plugin
[33] https://wiki.jenkins-ci.org/display/JENKINS/PMD+Plugin

6 Description of Views

In this chapter the different views of the product “Die schlaue Stadt” are described starting with the Simulation Control Center followed by the Operation Center.

6.1 Simulation Control Center

The next few chapters describe the different views of the Simulation Control Center. The Simulation Control Center in general is used to adjust and start the simulation.

6.1.1 Initial screen

When a new simulation is drawn up, the first view is the initial screen. As shown in Figure 82 the general options for the simulation, like name, map and bus system can be set. The map and the bus system have to be loaded from files. There also options for loading simulation from XML or loading simulations previously created. After confirming all options set there are other views to manage the simulation.

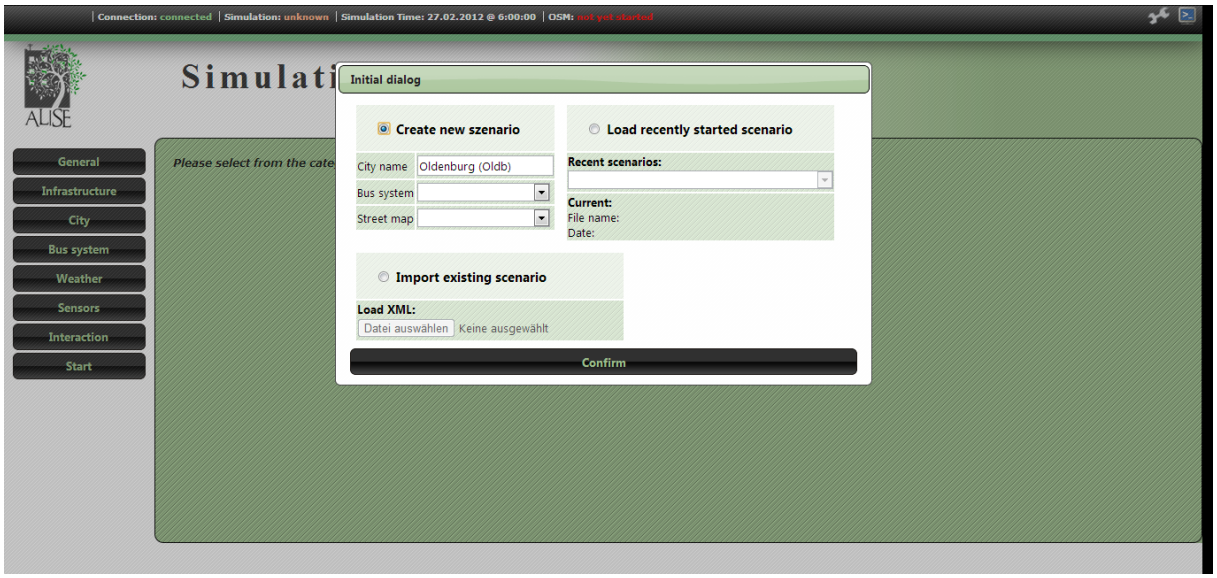


Figure 82: Initial screen

6.1.2 General view

In the general view (see Figure 83) options like the time that shall be simulated, the speed of the simulation and the default update steps per sensor can be set. There is also an option to toggle the sensor interferer.

Figure 83: General view

6.1.3 Infrastructure view

The infrastructure view holds some options to manage the servers needed for the simulation. Server (IP-) addresses can be given and traffic server can be added. As shown in Figure 84 the addresses for the user interfaces (OC and CC) can be set.

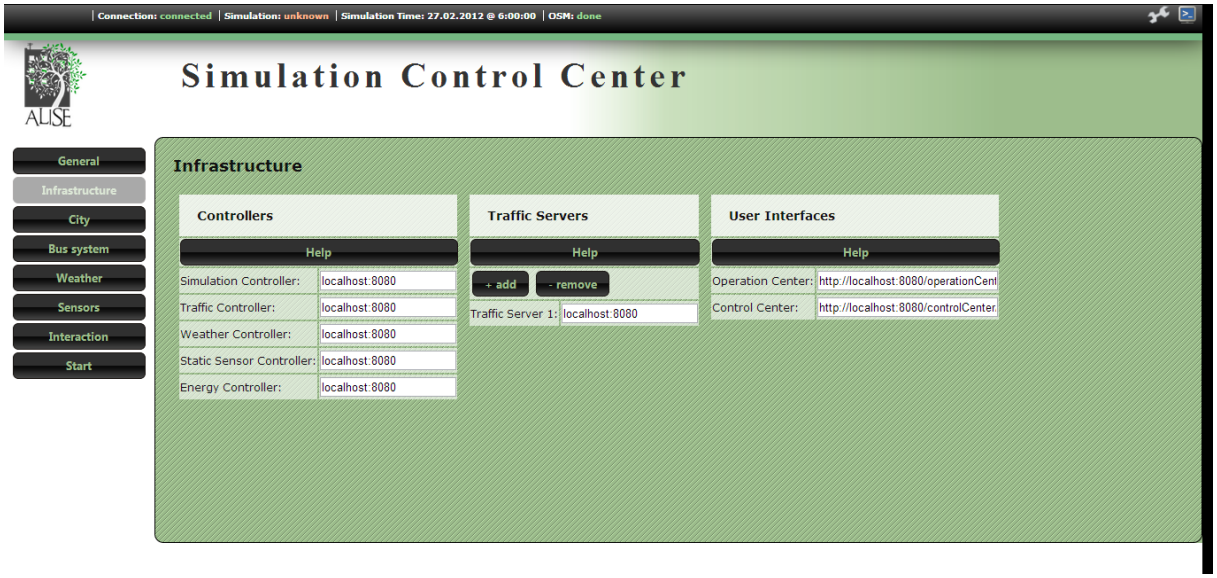


Figure 84: Infrastructure view

6.1.4 City view

City parameters like height or if the city is near the sea or a river, which are used for the weather simulation, can be set in the city view. As illustrated in Figure 85 the population of the simulated city along with the amount of mobile sensors like cars, bikes, trucks or motorcycles and there GPS ration can also be set in the city view.

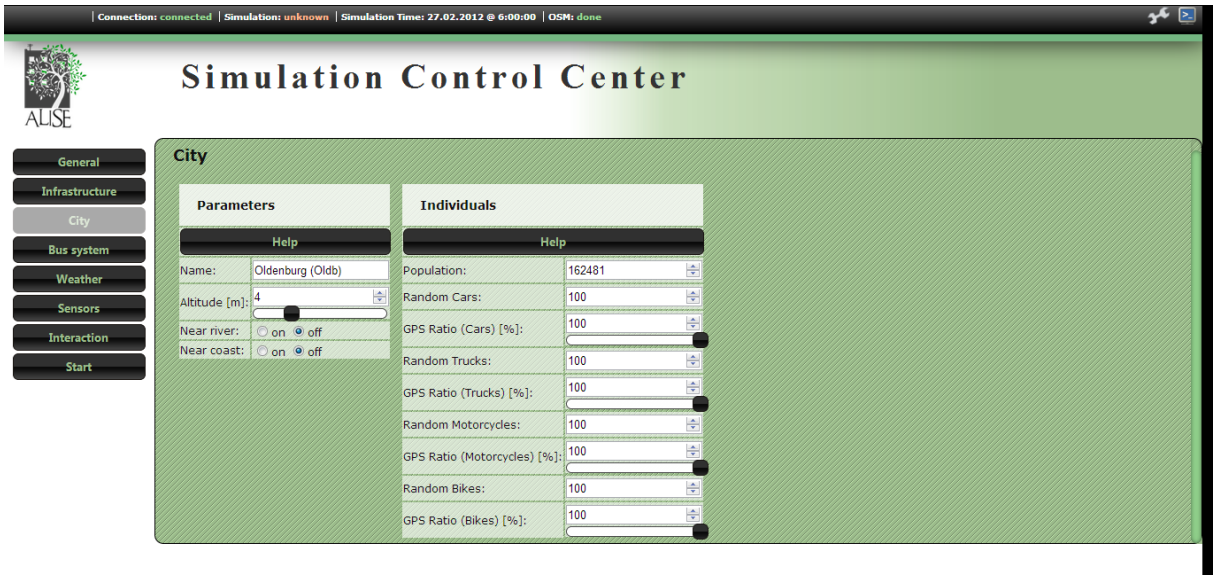


Figure 85: City view

6.1.5 Bus system view

If a GTFS file for the bus system is loaded on start, the bus system view provides the option to toggle single bus lines. As seen in Figure 86 the view also shows all kind of information about the bus lines in a clear table.

Figure 86: Bus system view

6.1.6 Weather view

In Figure 87 the weather view is shown. It provides the option to toggle aggregated weather data and add weather events like a hot or a rainy day. The weather events can be set up for different days and with special values. The events will affect the simulation on the given time.

Figure 87: Weather view

6.1.7 Sensors view

The Sensor view Figure 88 provides a map of the selected city and the possibility to add various static sensors like traffic lights, smart meters, induction loops and others, and simulation events like an event, a road blocking or others. Adding a static sensor or an event opens a dialog window where several options for the sensor or event can be set.

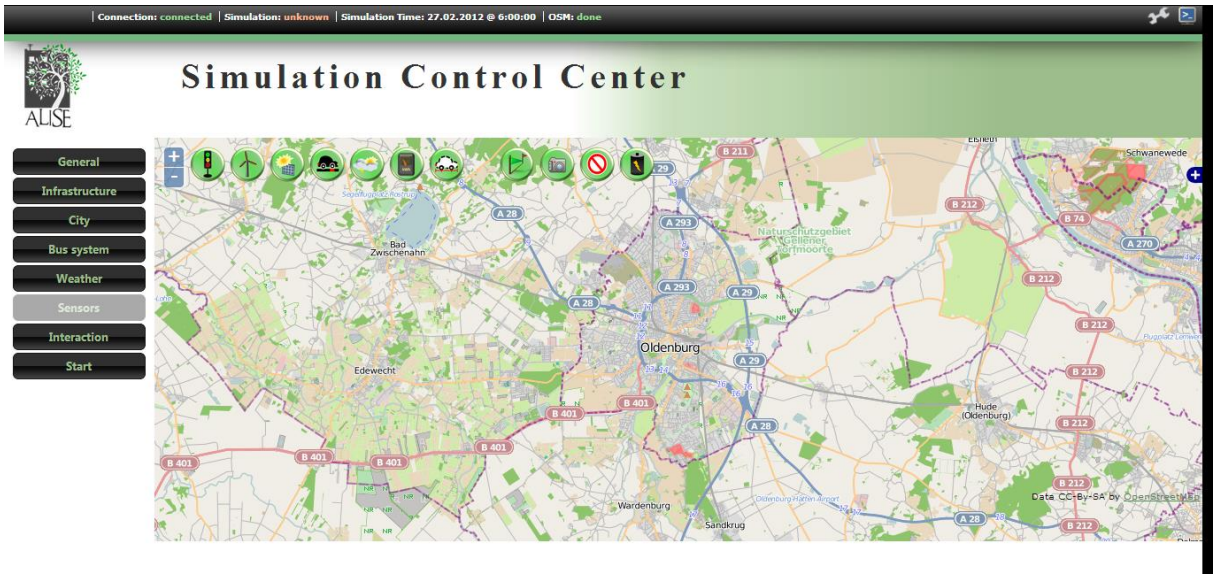


Figure 88: Sensors view

6.1.8 Interaction view

General options concerning the schedule logic can be adjusted in the Interaction view shown in Figure 89. The general update steps for the simulation and the tolerances in percent for the fuzzy logic can be set to a specific value.

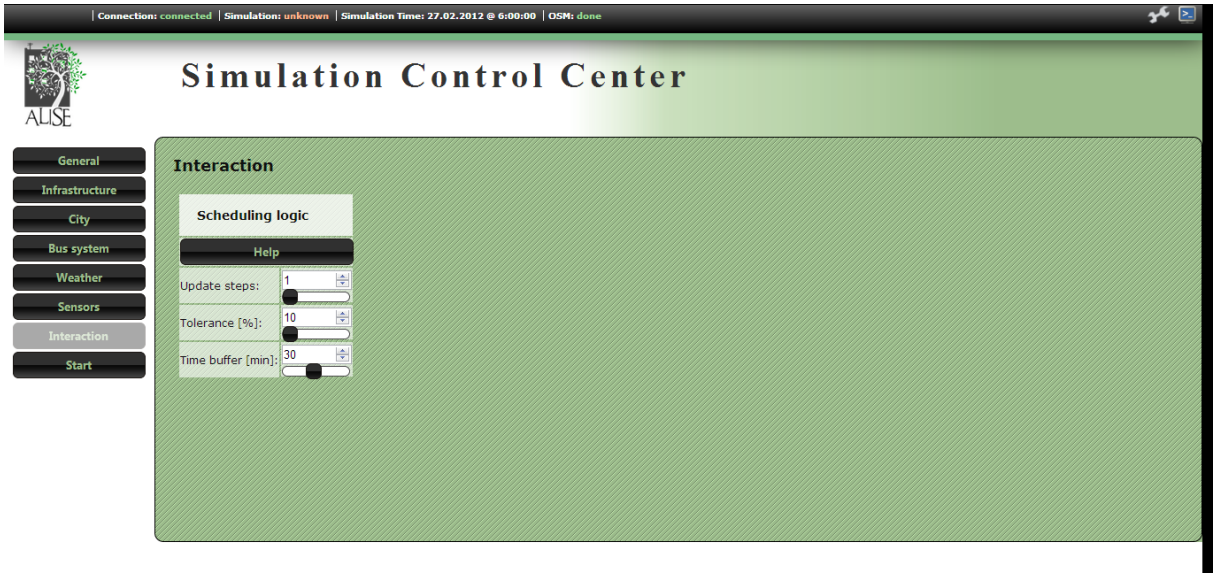


Figure 89: Interaction view

6.1.9 Start view

The start view provides a summary of all options set in the other views. This summary gets verified and notifies if anything is inconsistent. As shown in Figure 90 this view also makes it possible to export the simulation with all its starting parameters. The simulation can be started from this view.

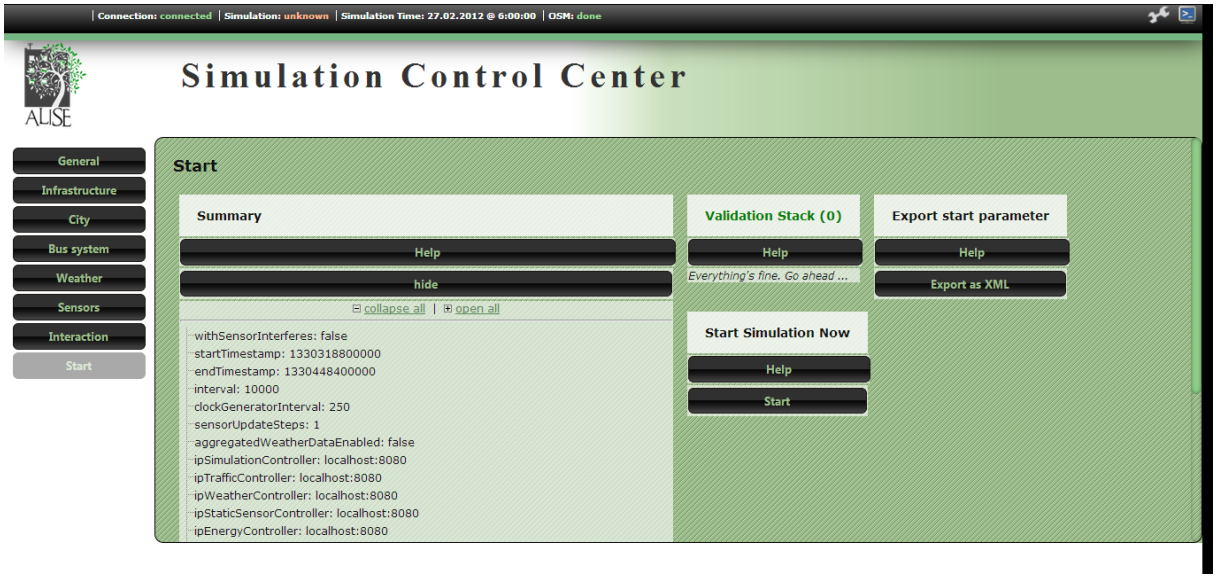


Figure 90: Start view

6.2 Operation Center

The Operation Center in general shows a map from the simulated city on the right hand side. In the upper part of the screen is a time ray with the simulation date and time and a progress bar. In the upper part of the map there are several options to change the view of the map. There is a GPS and a Speed Heat map, which shows the current traffic or the speed as a heat map. There are also views for the favorite sensors or the energy production/consumption. On the left hand side are various views, described by the following chapters.

6.2.1 Gate view

As shown in Figure 91 the amount of mobile sensors and the current GPS ratio can be seen. This view also provides a Configure button, to set the GPS ratio in percent.

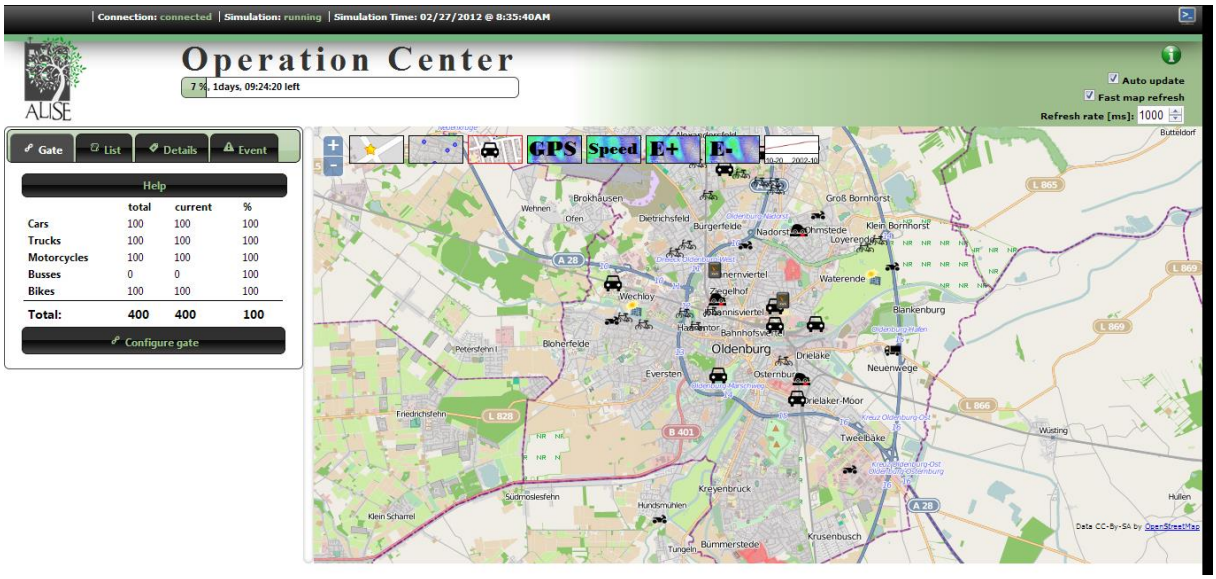


Figure 91: Gate view

6.2.2 List view

The List view (Figure 92) provides a list of all favorite, traffic, GPS and energy sensors as well as a search field to look up a specific sensor. Also a single sensor can be favored in this view.

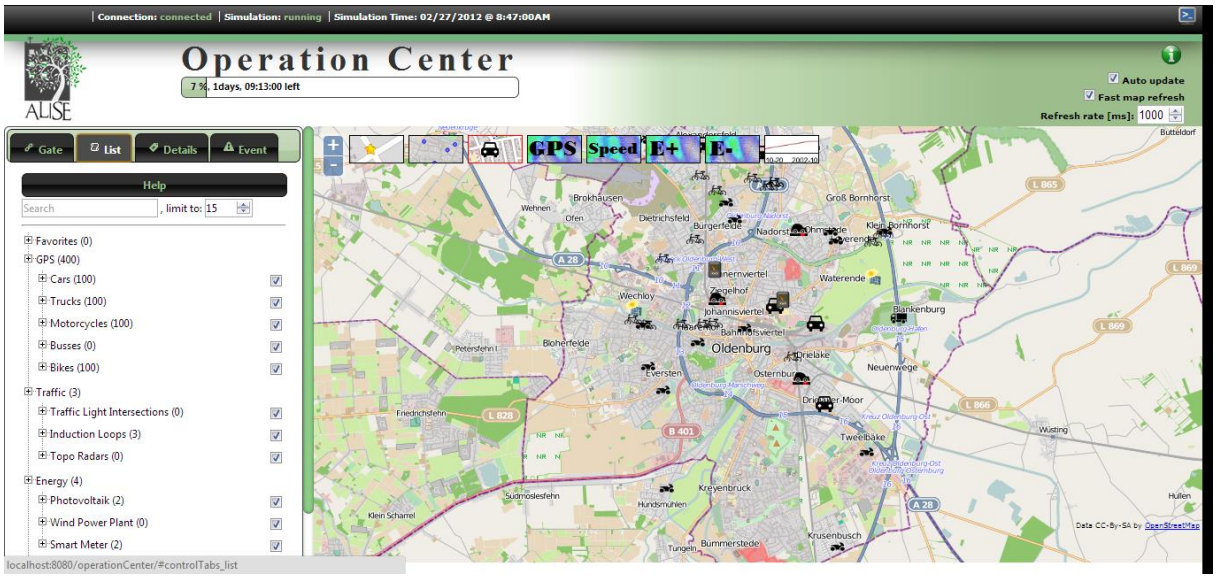


Figure 92: List view

6.2.3 Details view

The detail view, shown in Figure 93, appears when a specific sensor is selected. This view gives detailed information about the sensor, like average speed for a car sensor, its status, its name and other specific information. There are also options to favor the selected sensor or pan the view to it.

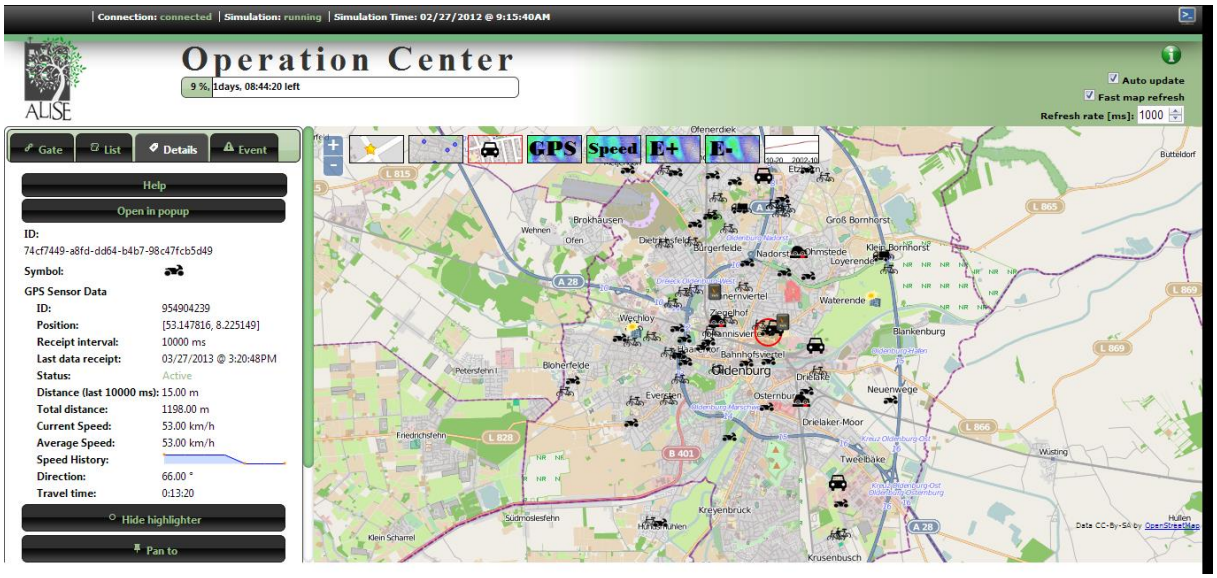


Figure 93: Detail view

6.2.4 Events view

In Figure 94 the event view is shown. It provides information about the incoming events, like weather events, energy events or other.

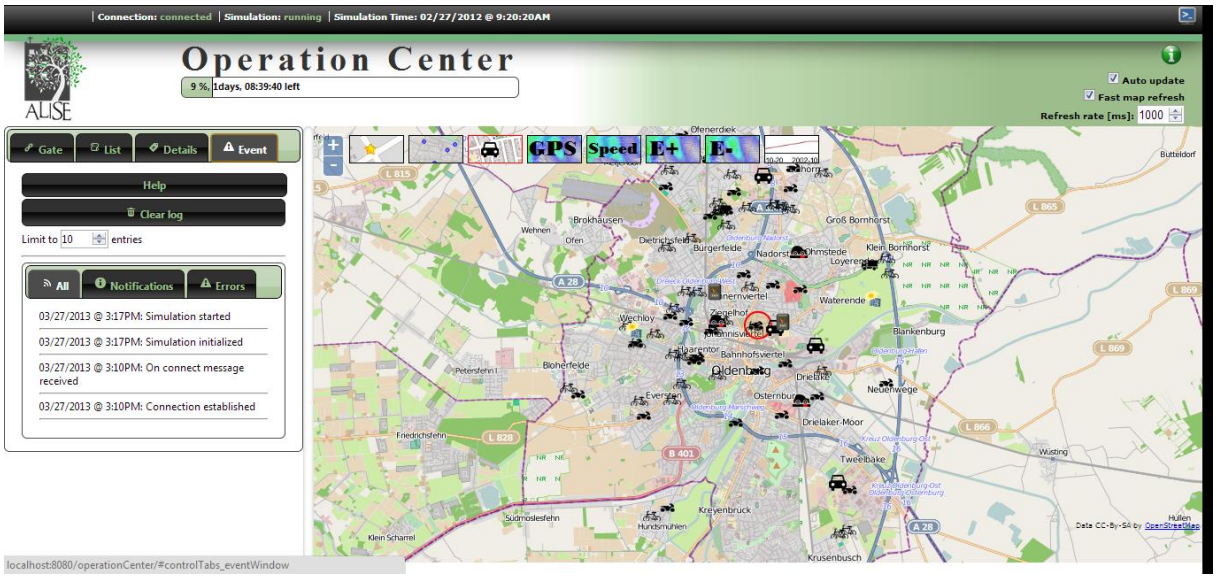


Figure 94: Event view

7 Seminars

The following chapters are the abstracts from the seminar papers. They give a short introduction to the groundwork, done for the project. Those papers were written in German.

7.1 Generierung von meteorologischen Daten

Im Rahmen der synthetischen Datengenerierung einer Stadt für die Projektgruppe ALISE spielt die Erzeugung von meteorologischen Größen über Sensoren eine wichtige Rolle. Die Temperatur, der Niederschlag oder die Sonnenscheindauer nehmen maßgeblich Einfluss auf andere Sensordaten wie die Stromerzeugung aus regenerativen Energiequellen. Zudem können sie für Skalierungstests der darunter liegenden Hardware verwendet werden. Mit dem Wissen über relevante Zustandsgrößen, deren Speicherformat, das Stadtklima und die Wetterprognose werden mögliche Alternativen für die Generierung aufgezeigt. Hierzu zählen die Messdaten von Wetterstationen, aktuelle Informationen von Wetterdiensten und eine vereinfachte Simulation von Wetterdaten, die neben der Datenbasis aus Wetterstationen und -diensten, Bauernregeln, Tagesund Jahresgänge der Zustandsgrößen, Wetteranomalien sowie die Gesetzmäßigkeiten des Stadtklimas berücksichtigt.

7.2 IBM – Eine Betrachtung der Smarter City Vision

Im Rahmen der Projektgruppe ALISE ist es wichtig die gesamte Smarter Planet bzw. Smarter City Vision von IBM zu betrachten und zu verstehen. Die Smarter City wird von IBM in verschiedene Bereiche unterteilt, die jeweils unterschiedliche Ziele verfolgen. Es ist wichtig diese Bereiche zu kennen und zu verstehen, was diese Bereiche ausmacht. Ein weiterer wichtiger Punkt von IBMs Vision sind die drei Schichten, die jeweils unterschiedliche Aufgabenbereiche vereinen. Eine Betrachtung von anderen Smarter City Vision ist dabei genauso wichtig wie die Beobachtung von Fallbeispielen.

7.3 IBM Cognos Business Intelligence

IBM Cognos ist eine Business Intelligence Plattform zur Analyse und Visualisierung von Daten aus unterschiedlichen Quellen. Das drei Schichtenmodell der Software sowie deren Komponenten ermöglichen es geschäftskritische Informationen jederzeit abrufen zu können und mit Mitarbeitern zu teilen. Verschiedene Reporttypen verhelfen zu einer übersichtlichen Darstellung der Daten um die Entscheidungsfindung optimal zu unterstützen.

7.4 IBM InfoSphere Streams

Im Rahmen einer studentischen Projektgruppe soll eine Anwendung zur Generierung, Verarbeitung und Visualisierung von Sensordaten im Kontext von Smart Cities entwickelt werden. Zur Verarbeitung soll das datenstromverarbeitende Tool IBM InfoSphere Streams genutzt werden, welches in dieser Arbeit genauer beschrieben wird. Dazu werden einige Pilotprojekte vorgestellt und im Anschluss der Aufbau und die Funktionsweise von InfoSphere Streams beschrieben. Zusätzlich

werden einige Design Pattern für die Verwendung innerhalb der Projektgruppe vorgestellt. Ein konkretes Anwendungsbeispiel im Bereich des Smarter Traffics bildet den Abschluss dieser Arbeit.

7.5 Qualität in Sensordaten

Im Rahmen der Projektgruppe ALISE ist die Datenqualität der Sensordaten für die Datengenerierung wichtig. Sensoren beobachten ein zugeteiltes Umfeld und zeichnen dieses auf. Bei der Erfassung von Daten können verschiedene Problematiken entstehen. Unter anderem kann der Sensor durch Umwelteinflüsse gestört werden und so fehlerhafte Daten aufzeichnen oder zwei Sensoren die das gleiche Umfeld beobachten widersprechen sich. Des Weiteren können die erzeugten Daten bei der Eintragung in eine Datenbank Fehler wie z. B. NULL-Werte aufweisen. Diese Fehler werden durch die verschiedenen Qualitätsdimensionen beschrieben.

7.6 Scrum und seine Anwendung für die PG ALISE

Diese Seminararbeit stellt die agile Projektmanagementmethode Scrum, sowie die Anwendung innerhalb der Projektgruppe ALISE vor. Das Rahmenwerk Scrum ist für eine iterative und inkrementelle Entwicklung von Produkten gedacht und besteht aus den Rollen: Product Owner, Entwicklungsteam und Scrum Master. Weiter besteht es aus den Artefakten: Product Backlog, Sprint Backlog, Impedimentlist und Burn Down Charts. An Ereignissen gibt es den Sprint, das Sprint Planning Meeting, Daily Scrum und Sprint Review. Während des Sprints wird das jeweilige Produktinkrement durch das Entwicklungsteam erstellt und es finden alle anderen Ereignisse statt. Die Ereignisse werden iterativ durchlaufend, bis das Produkt vollendet ist. Für die Projektgruppe ALISE muss Scrum angepasst werden, da es sich um ein studentisches Projekt handelt und nicht um ein Projekt mit Vollzeitbeschäftigung.

7.7 Sensordaten - Energie: Smart Meter/Smart Grid

Im Rahmen der Projektgruppe ALISE führt diese Arbeit in die Themen Smart Meter und Smart Grids sowie deren Sensordaten ein. Smart Meter sind intelligente Zähler, die im Vergleich zu klassischen Zählern den Energieverbrauch feiner aufschlüsseln können. Smart Grids hingegen sind intelligente Netze, die die Voraussetzung für eine intelligente Steuerung, (Lasten-) Verteilung, Speicherung und Erzeugung von elektrischer Energie in Energieübertragungs- und Verteilnetzen darstellen. Wichtige Sensordaten in Smart Grids und Smart Meter sind Lastgänge sowie die standardisierte Form derer, den Standardlastgängen. Diese stellen die Aufzeichnung der Leistungsaufnahme von Verbrauchern über einen bestimmten Zeitraum dar.

7.8 Sensordaten des Individualverkehrs

Im Rahmen der studentischen Projektgruppe ALISE der Universität Oldenburg soll eine Software Anwendung zur Generierung, Verarbeitung und Visualisierung von Sensordaten im Kontext von Smarter Cities entwickelt werden. Der Fokus hierbei liegt auf der Generierung von Sensordaten einer

virtuellen Smart City. Diese Arbeit beschäftigt sich mit den Eigenschaften von Sensordaten des Individualverkehrs und betrachtet Ansätze zur Generierung von realitätsnahen Verkehrsdaten.

7.9 Sensordaten ÖPNV

Die Sensordatengenerierung im Bereich vom öffentlichen Personennahverkehr (ÖPNV) spielt im Rahmen von Smarter-City-Umgebungen eine große Rolle. Diesem Themengebiet widmet sich die Projektgruppe ALISE. Die Projektgruppe ALISE hat als Ziel, mit Hilfe von virtuellen Sensoren Daten zu generieren, die im Smart-City Kontext Anwendung finden können. Diese Seminararbeit befasst sich unterdessen mit der Sensordatengenerierung im öffentlichen Personennahverkehr. Der Fokus liegt insbesondere auf den öffentlichen Straßenverkehrsmitteln. Interessant sind in diesem Zusammenhang vor allem die positions- und kapazitätsbestimmenden Sensoren, wie GPS oder Infrarotsensoren. Sie können Missstände im ÖPNV wie z.B. wiederholte Verspätungen oder häufige Fahrzeugüberlastungen aufdecken. Beispieldaten sind bereits in einem von Google spezifizierten Format vorhanden. Sie können dazu dienen weitere, synthetische Daten zu erzeugen und weiter in Google Maps zu visualisieren. Ferner lassen sich diese Daten natürlich auch verwenden, um Systemstresstests durchzuführen. Mit diesem Ergebnis können letztendlich Smarter-City-Systeme weiter ausgebaut werden.

7.10 Virtuelle Sensoren

Gute virtuelle Sensoren liefern fehlerhafte Daten, denn Sensorsimulationen scheitern heute an ihrer unrealistischen Perfektion. Fehler, die bei echten Sensoren die Messergebnisse verfälschen, müssen in virtuellen Sensoren synthetisch erzeugt werden. Im vorliegenden Paper wird geklärt, warum im Zuge der Projektgruppe ALISE bei der Konzeption virtueller Sensoren die Fehleraufladung der synthetisch erzeugten Daten ausschlaggebend ist. Die betrachtete Thematik wird in den Kontext des Gesamtprojekts gestellt, danach werden die Schritte der Konzeption virtueller Sensoren erst theoretisch und dann praktisch am Beispiel von GPS verdeutlicht. Zuletzt werden die Ergebnisse der Ausarbeitung kurz in einer Retrospektive zusammengefasst.

7.11 Verarbeitung und Speicherung von Sensordaten

Die Verarbeitung und Speicherung von Sensordaten spielt eine wichtige Rollen im Rahmen der Projektgruppe ALISE. In dem dreischichtigen System von der synthe- tischen Erzeugung der Sensordaten bis hin zur deren Visualisierung, besetzt sie die mittlere Schicht und nimmt somit die Rolle eines Vermittlers ein. Die Visualisierung und die Generierung der Sensordaten stellen den wichtigsten Aspekt in der Projekt- gruppe dar. Mit dem Wissen über die geeignete Verarbeitung und Speicherung der Daten kann eine bessere Skalierbarkeit des Systems mit steigender Datenlast gewährleistet werden.

7.12 Vergleich ausgewählter Simulations-Frameworks und Bibliotheken

Diese Seminararbeit wurde im Rahmen der Projektgruppe (PG) ALISE gefertigt, deren Ziel es ist, mit Hilfe von Simulationen und virtuellen Sensoren realitätsnahe Wetter-, Vekehrs- und Energiedaten zu generieren. In dieser Arbeit werden zu diesem Zweck drei Simulationsframeworks vorgestellt und überprüft, welches von ihnen sich für den Einsatz in der PG eignet. Zuvor werden Simulations-Grundlagen zum besseren Verständnis der Funktionsweisen solcher Frameworks vermittelt und die Vergleichskriterien offen gelegt.

8 User Guide

This user guide describes how to set up your local work environment to use the smart city application. It includes checking out source files, configuring build environment and starting the application. You can use this guide with Windows and Linux. In addition to that it describes how to start the data stream management system InfoSphere Streams 3 and Odysseus.

8.1 Setting up and starting the smart city application

Choosing a Branch

This guide has only been tested with the distributed branch. Other branches may work with this guide.

Setting up a Linux build environment

The following instructions have been tested on an Ubuntu 12.10 64-bit machine. Since all source files are basically compiled by Java this may work on other versions, distributions and architectures, too. But reports about successful or failed builds on other systems are appreciated.

To build and run the application you will need following tools:

- Oracle JDK 7 or OpenJDK 7
- Apache Maven at least version 2
- Apache Subversion
- Apache TomEE

Installing the JDK

Since the Oracle JDK is no longer in Ubuntu's package repository you are advised to install the open source implementation OpenJDK (see Method 1). Otherwise if you don't want to use OpenJDK download the Oracle's JDK for your architecture from their website (see Method 2).

Method 1
To install the OpenJDK through Ubuntu's package repository type:

```
$ sudo apt-get install openjdk-7-jdk openjdk-7-source openjdk-7-demo
openjdk-7-doc openjdk-7-jre-headless openjdk-7-jre-lib
```

Method 2
Download the Oracle's JDK for your architecture from Oracles's website (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html) and extract the Tarball to /usr/lib/jvm. Then type the following commands in your bash and replace VERSION with yours:

```
$ sudo update-alternatives --install "/usr/bin/java" "java"
"/usr/lib/jvm/jdk1.7.0_VERSION/bin/java" 1
```

```
$ sudo update-alternatives --install "/usr/bin/javac" "javac"
"/usr/lib/jvm/jdk1.7.0_VERSION/bin/javac" 1
$ sudo update-alternatives --install "/usr/bin/javaws" "javaws"
"//usr/lib/jvm/jdk1.7.0_VERSION/bin/javaws" 1
$ sudo update-alternatives --install "/usr/bin/jar" "jar"
"/usr/lib/jvm/jdk1.7.0_VERSION/bin/jar" 1
```

Set the new Java version:

```
$ sudo update-alternatives --set "java"
"/opt/Oracle_Java/jdk1.7.0_VERSION/bin/java"
$ sudo update-alternatives --set "javac"
"/opt/Oracle_Java/jdk1.7.0_VERSION/bin/javac"
$ sudo update-alternatives --set "javaws"
"/opt/Oracle_Java/jdk1.7.0_VERSION/bin/javaws"
$ sudo update-alternatives --set "jar"
"/opt/Oracle_Java/jdk1.7.0_VERSION/bin/jar"
```

Installing Apache Maven

You will need Apache Maven to resolve dependencies and build the project. Although it is recommended to install maven through official Ubuntu packages (see Method 1) you can also manually download it from the website (see Method 2). You can grab version 3.x by a manual download whereas Ubuntu's package repository only provides version 2.x. Both versions are suitable for building the project.

Method 1
To install through Ubuntu's package repository type:

```
$ sudo apt-get install maven
```

Method 2
Go to http://maven.apache.org/download.cgi and download your preferred version. Extract the Tarball to /opt/maven3.

Type:

```
$ sudo chmod 755 /opt/maven3/maven
```

Then add set an alias in your .bashrc:

```
$ echo 'alias maven='/opt/maven3/maven'' >> ~/.bashrc
```

Set up Apache Maven

Since there is some closed source software used in the project you'll need to put an extra settings.xml in your .m2 folder. Move the file to ~/.m2.

settings.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">

<pluginGroups>
</pluginGroups>
<proxies>
</proxies>
<servers>
</servers>

<mirrors>
<mirror>
<id>PGAliseMirror</id>
<mirrorOf>PGAliseRepository</mirrorOf>
<url>http://134.106.56.230:18081/nexus/content/groups/public</url>
</mirror>
</mirrors>

<profiles>
      <profile>
<id>PGAlise</id>
<repositories>
<repository>
<id>PGAliseRepository</id>
<url>http://</url>
<releases>
                  <enabled>true</enabled>
                  <updatePolicy>daily</updatePolicy>
            </releases>
<snapshots>
                  <enabled>true</enabled>
                  <updatePolicy>always</updatePolicy>
            </snapshots>
</repository>
</repositories>
<pluginRepositories>
<pluginRepository>
<id>PGAliseRepository</id>
            <url>http://</url>
<releases>
                  <enabled>true</enabled>
                  <updatePolicy>daily</updatePolicy>
            </releases>
<snapshots>
                  <enabled>true</enabled>
                  <updatePolicy>always</updatePolicy>
            </snapshots>
</pluginRepository>
</pluginRepositories>
</profile>
</profiles>
```

```
<activeProfiles>
<activeProfile>PGAlise</activeProfile>
</activeProfiles>
</settings>
```

Installing Apache Subversion

SVN is needed for downloading the source files from the central repository.

Paste following command into your bash in order to install SVN through Ubuntu's package repository:

```
$ sudo apt-get install subversion
```

Set up Apache TomEE

To run the application after building you will need to have a TomEE server running. There is already a preconfigured TomEE available. Download it from our homepage (http://www.pg-alise.com) and extract it into your home folder.

Download source files on Linux

Download the source files from GitHub under the following link:

https://github.com/luxmeter/pgalise

Another way to get the source files is to check out source files from our SVN. To check out the files, navigate to C:\, create a new directory, name it "distributed". Now right click on it and choose the TortoiseSVN checkout function from the context menu. Insert

```
svn://duemmer.informatik.uni-
oldenburg.de:1295/Java/Development/Branches/distributed
```

as destination and click checkout.

Building and Deploying with Linux

When copying the following commands it is assumed you checked out the source files and extracted TomEE to your home folder. Otherwise you'll need to adapt the paths:

```
$ svn update ~/distributed && mvn -DskipTests=true -e -
f/home/$USER/distributed/pom.xml clean install && rm -rf
~/tomee/apps/simulation && mv ~/distributed/ear/target/simulation
~/tomee/apps && sh ~/tomee/bin/catalina.sh run
```

You can now access the Control Center and Operation Center by invoking http://localhost:8080/controlCenter and http://localhost:8080/operationCenter.

Setting up a Windows build environment

The following instructions are tested on a Windows 8 64-bit machine. Reports about successful and failed builds on other versions and architectures are much appreciated.

To build and run the application you will need following tools:

- Oracle JDK 7
- Apache Maven at least version 2
- TortoiseSVN
- Apache TomEE

Installing the JDK

Download Oracle's JDK installer for your platform from the download site (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html) and run the wizard that will guide you through the installation.

Installing Apache Maven

Download latest stable version of Apache Maven from http://maven.apache.org/and extract it to Files. Rename it that you have Files\maven. Add Files\maven\bin to your PATH. Also be sure JAVA_HOME is set.

Set up Apache Maven

Since there is some closed source software used in the project you'll need to put an extra settings.xml in your .m2 folder. The mentioned xml file is described in section “Setting up a Linux build environment”.

Installing TortoiseSVN

TortoiseSVN is needed for downloading the source files from the central repository.

Download the latest version from http://tortoisesvn.net/downloads.html and install.

Set up Apache TomEE

To run the application after building you will need to have a TomEE server running. There is already a preconfigured TomEE available. Download it from our homepage (http://www.pg-alise.com) and extract it to C:\Program Files\tomee.

Checking out source files on Windows

Download the source files from GitHub under the following link:

https://github.com/luxmeter/pgalise

Another way to get the source files is to check out source files from our SVN. To check out the files, navigate to C:\, create a new directory, name it “distributed”. Now right click on it and choose the TortoiseSVN checkout function from the context menu. Insert

```
svn://duemmer.informatik.uni-
oldenburg.de:1295/Java/Development/Branches/distributed
```

as destination and click checkout.

Checking out source files may take a long time, since you have to grab approximately 650 MB. It basically depends on your network connection.

Building and Deploying with Windows

When conducting the following instructions it is assumed you both checked out the source files and extracted TomEE to your home folder. Otherwise you'll need to adapt the paths. Open cmd and type:

```
cd C:\distributed
maven -DskipTests clean install
```

Now copy C:\distributed\ear\target\simulation to C:\Program Files\tomee\apps\. Before each new build you are supposed to delete the old simulation from TomEE.

The last step is about starting TomEE.

Type:

```
cd C:\Program Files\tomee\bin
catalina.bat run
```

You can now access the Control Center and Operation Center by invoking http://localhost:8080/controlCenter and http://localhost:8080/operationCenter.

8.2 Starting the data stream management system

To start the data stream management system the user has to connect to the server with DSMS. In the following description the steps to connect and to start the streaming application are explained in detail.

Connecting to the Server

Two server exist with a DSMS, one CentOS server with InfoSphere Streams 3 (134.106.56.58) and a Windows 2008 (134.106.56.230) with Odysseus.

Connection to the CentOS Server (InfoSphere Streams 3)

1. Terminal based

In your terminal you can connect via the ssh command:

```
ssh -c arcfour,blowfish-cbc -XC root@134.106.56.58
```

2. With a GUI (recommended)

To use a GUI it is mandatory to install a VNC Viewer on your PC.

Now you can establish the connection in your terminal via:

```
ssh -N -L 5900:localhost:5901 root@134.106.56.58
```

This means the monitor will be tunneled to your pc. Now start your VNC Viewer and select „Localhost“ as your VNC Server. Then you can connect to the desktop of the CentOS server.

Connection to the Windows 2008 Server (Odysseus)

To connect to this server you just need a RDP-Client and type in the login data.

Starting InfoSphere Streams 3.0

To start the streams application you need to open the Streams Studio. Just double-click on the alias on the desktop screen or just open it from the Streams Studio folder in the home directory.

To start the streams application you find the project explorer on the left hand side. Open the AliseStreamProcessing project, then choose the composite applicationTest and open the AdvancedAliseProcessing streaming application. Under this application you find two different build configurations do a right-click on the Distributed and click on “set active”. After this, click on the application and launch it. In the next window you can define output level and start the application. Now this application is deployed as a job. You can view this job with its processing elements in the Streams Explorer. With a right-click on the application you can view the application graph. To view different metrics on these job just open the view with the Task Launcher → Publish and Run → Monitor jobs.

When you are restarting eclipse you need to set the installation folder of InfoSphere Streams and the studio wants to log in at InfoSphere Streams. The installation folder can be found in the home directory, it is called InfoSphereStreams. To log on just input *ibmadmin* as the user and *ibm* as the password.

Restarting in case of an error

If an error occur it might by necessary to restart InfoSphere Streams completely. To do this go to the terminal and change the user `su ibmadmin` (PW *ibm*).

Then stop the instance of InfoSphere Steams with the command `streamtool stopinstance --force`. After it has stopped, restart the instance with the command `streamtool startinstance.` Now InfoSphere Streams can be used again.

Starting Odysseus

To start Odysseus you need to open the Odysseus Studio (studio.exe). You can find this application under C:\Odysseus. In the Project Explorer (in the Odysseus Studio) open the Project *OneSource*. You can see different Odysseus Scripts. To start this project it is very important to start these scripts in the right order. To start a script open it and press the green play button. The right order is:

1. PQLSource
2. SelectRoute
3. MySQLSink
4. MySQLSelect
5. DeDuplicateNormal
6. PQLSink

Now you can find the running sources and sinks on the left-hand side and the running queries under the editor. To show metrics just click on a query and select the wanted metric.

9 Project Evaluation

This section describes at first some performance tests of the project group. After that a detailed scenario for a scale test of the smart city application are presented. At the end of this section the scale test results follow.

9.1 Performance Tests

The project group tests continuously the performance of the individual software modules during the implementation process. On the one hand the source code was reviewed by other project members to implement more efficient algorithms, which can be executed or solved the problem in a shorter time, on the other hand different implementations of an interface were build to handle a situation in a different style. For instances, there are three implementation of the traffic scheduler, which schedules the departure time of the vehicles. The *ListScheduler*, the *SortedListScheduler* and the *TreeSetScheduler* are distinguished from each other by the way, how they realize the internal lists to save the different departure times. This difference has a huge impact on the performance of the scheduler. The following bar chart shows the execution time to schedule 10.000 vehicles with each implementation:

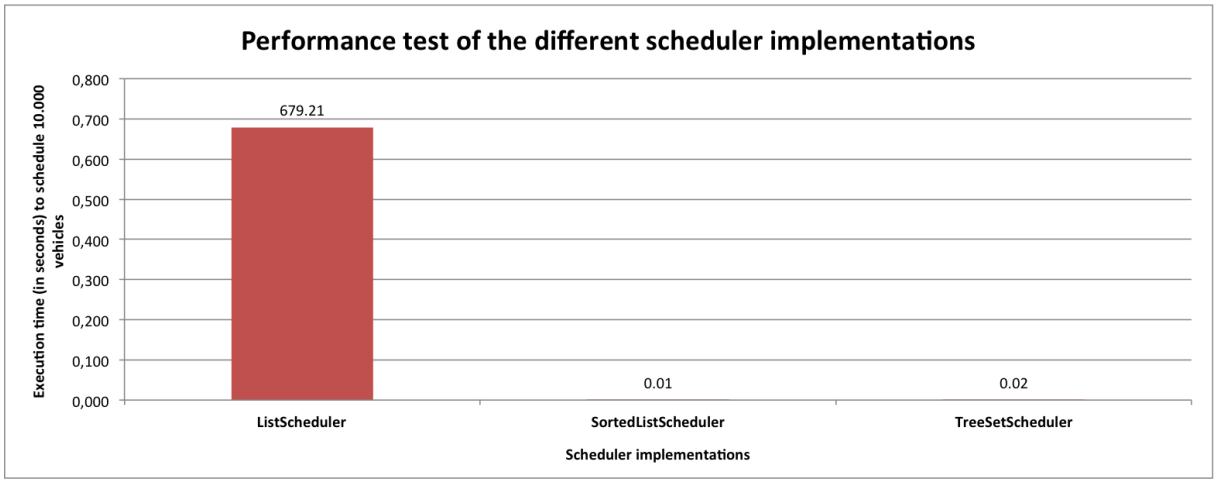


Figure 95: Performance test of the different scheduler implementations

One of the first runnable versions of the simulation needed more than 45 minutes to start with 40.000 vehicles. The main components wasting a lot of time were the traffic scheduler, the Djikstra algorithm for the vehicle route calculations and the algorithm to find the start positions of the vehicles on each traffic server. After changing the used *ListScheduler* to the more efficient*SortedListScheduler*, the simulation start with 10.000 vehicles to schedule needed only about 8 minutes. Performance improvements for the algorithm to find the start positions of the vehicles on each traffic server reduced the start duration of 8 minutes to only 3 minutes. This example shows that the project group tried continuously to identify time wasters and shrink their processing times by enhancing their implementations.

9.2 Scale test

Based on a fixed scenario the project group tested the scalability of the simulation. Besides the scalability, the advantage of the distributed architecture of the simulation is highlighted because the scenario was repeated with more than one traffic server. Furthermore, this scenario could be taken to compare the project group result with further extensions that based on the smart city application.

9.2.1 Scale test scenario

The scenario simulates 20 minutes from 08:00 to 08:20 of the 11thSeptember 2011. The OSM file of the used street network refers to the city Oldenburg. The simulated vehicles are limited to cars. So, no bicycles, trucks, motorcycles or busses are created during the simulation time. However, the timetables for the bus route with the number 301 are parsed. The vehicles follow no traffic rules except the Nagel–Schreckenberg model. All cars are scheduled after one simulation second and each car has a random start position and a random destination. No other sensors are placed in the simulation.The used algorithm to find the shortest path between the start position and the destination is the Dijkstra's algorithm.

The following Table 8 shows the different parameters changing during each simulation runtime of the described scenario above:

Simulation run	Amount of cars to schedule	Amount of traffic servers
1	1.000	1
2	5.000	1
3	10.000	1
4	50.000	1
5	100.000	1
6	1.000	2
7	5.000	2
8	10.000	2
9	50.000	2
10	100.000	2

Table 8: Simulation runs of the scale test

The simulation runs 1 to 5 are executed on one server. The simulation runs 6 to 10 use two traffic servers and the feature of the distributed architecture. In the process the traffic servers are initiated and updated in parallel.

9.2.2 Scale test results

The different scale tests were realized on a machine with the following characteristics:

- Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz (4 cores)
- 8 GB DDR3 memory (PC 1333)
- 5400-rpm hard drive

The following Figure 96 shows the total simulation duration in minutes for all simulation runs described in Table 8:

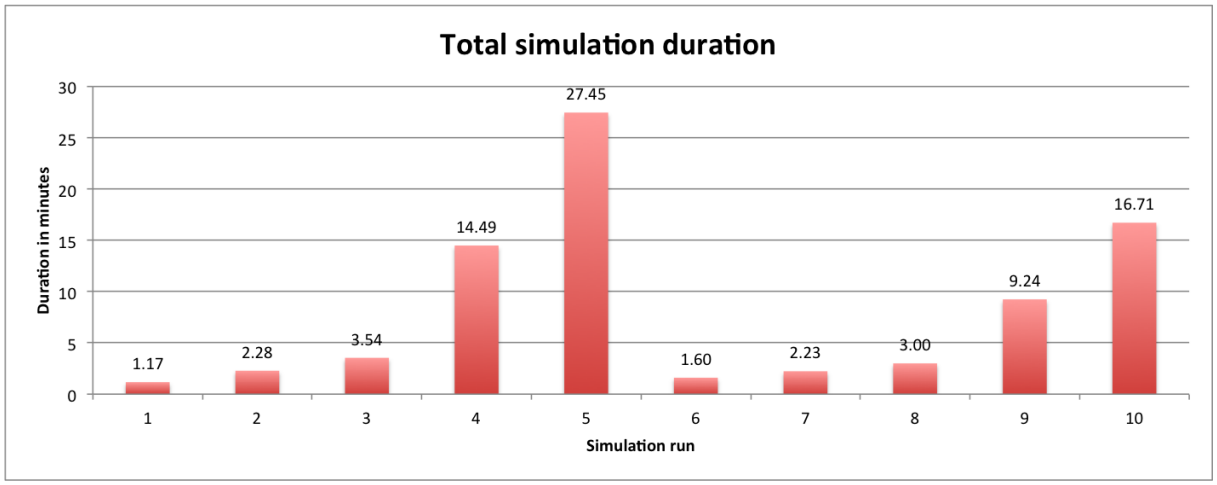
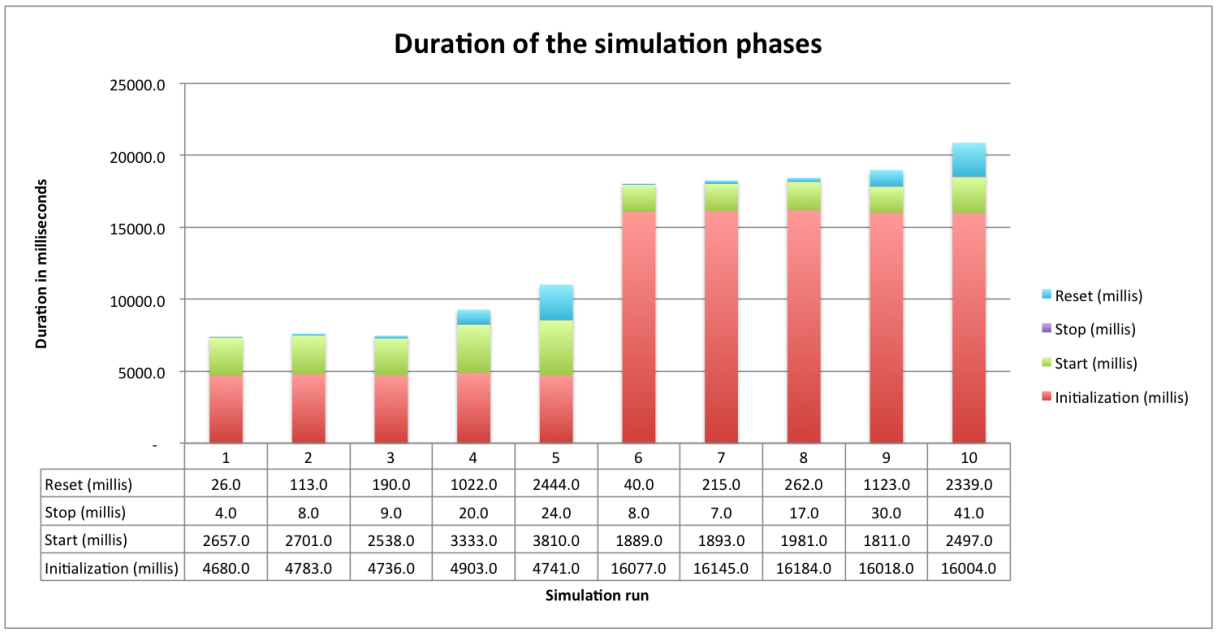


Figure 96: Total simulation duration of all simulation runs

Only simulation run 1 with 1.000 cars takes less time in comparison with the results of the distributed architecture (see simulation runs 6 to 10). The equivalent simulation run 6 takes approximately 5 minutes longer. The simulation duration of the simulation runs with 5.000 cars (simulation run 2 and 7) are nearly equal. However, the simulation runs with 10.000 illustrates that the runtime of the distributed architecture (simulation run 8 with 3 minutes) needs about 30 seconds less than the simulation run 3 (3,54 minutes) on a single server. This difference is even higher with more cars to simulate. So, the simulation run 9 with 50.000 cars differentiates about 5 minutes from the equivalent simulation run 4. The maximum difference of approximately 11 minutes shows the comparison between simulation run 10 with simulation run 5. These two runs simulate 100.000 vehicles.



	1	2	3	4	5	6	7	8	9	10
Reset (millis)	26.0	113.0	190.0	1022.0	2444.0	40.0	215.0	262.0	1123.0	2339.0
Stop (millis)	4.0	8.0	9.0	20.0	24.0	8.0	7.0	17.0	30.0	41.0
Start (millis)	2657.0	2701.0	2538.0	3333.0	3810.0	1889.0	1893.0	1981.0	1811.0	2497.0
Initialization (millis)	4680.0	4783.0	4736.0	4903.0	4741.0	16077.0	16145.0	16184.0	16018.0	16004.0

Figure 97: Duration of the simulation phases

Figure 97 illustrates the durations of the individual simulation phases in milliseconds. It is obvious that the initialization phase takes the most time. Whereas the simulations runs 1 to 5 takes nearly the same time to initiate the simulation, the simulation runs 6 to 10 needs a lot more time (at least 2 minutes more). The changing amount of cars has no impact on the simulation runtime in that phase.

However, the start phase is affected by the changing amount of cars. More cars to simulate take more time. The simulation runs 6 to 10 with two traffic servers needs less time in contrast to the simulation runs 1 to 5 with only one traffic server. The difference is about 1 second.

The stop phase takes least of all time. Although, due to the fact that more cars needs more time (only a few milliseconds), this phase takes no more time than 41 milliseconds in every simulation run.

The reset phase shows also the impact of the changing car amount. Whereas the simulation run 1 with 1.000 cars to simulate takes 26 milliseconds, the simulation run 5 with 100.000 cars needs about 2,4 seconds. Except for the simulation run 10, all other simulation runs with the distributed architecture needs more time in contrast to the other equivalent simulation runs. Notwithstanding that the difference is no more than 100 milliseconds.

The following Figure 98 presents the total time duration of all simulation steps in a simulation run between the start phase and the stop phase.

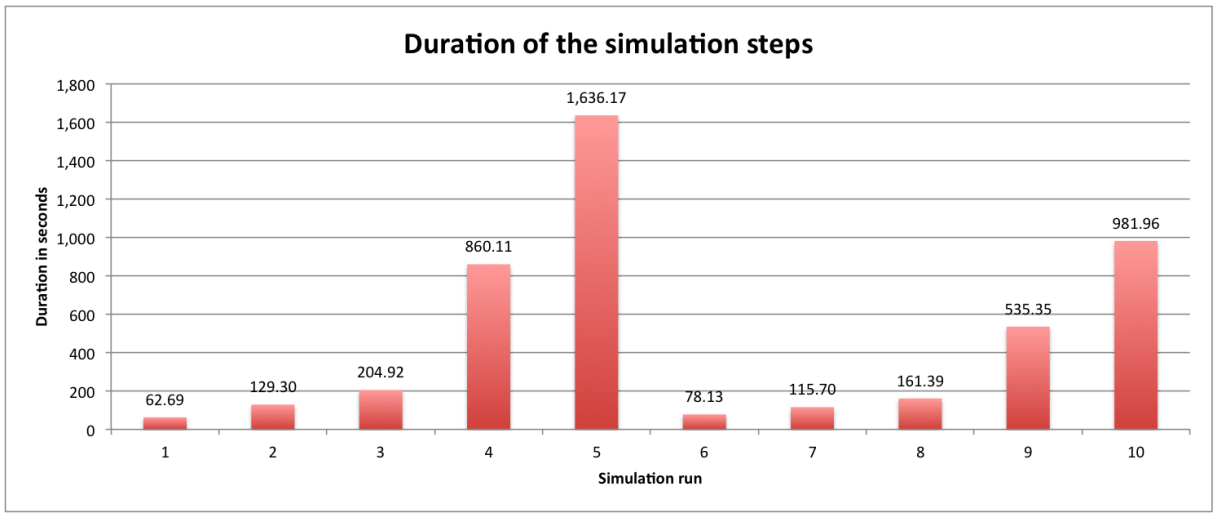


Figure 98: Duration of the simulation steps

At first, the amount of cars in the simulation has a huge impact on the simulation step duration. Whereas the simulation run 1 with 1.000 cars needs only about 1 second, the simulation run with 100.000 cars takes approximately 27 minutes. Notice that the simulation runs with the distributed architecture takes except simulation run 6 less time. The highest time saving highlights simulation run 10 with about 11 minutes time difference to the simulation run 5.

Despite the fact that the average duration of one simulation step is lower by using one traffic server (simulation run 1 to 5), which can be shown in Figure 99 and in Figure 100, the total duration of all simulation steps is mostly higher with growing vehicles to simulate (see Figure 98).

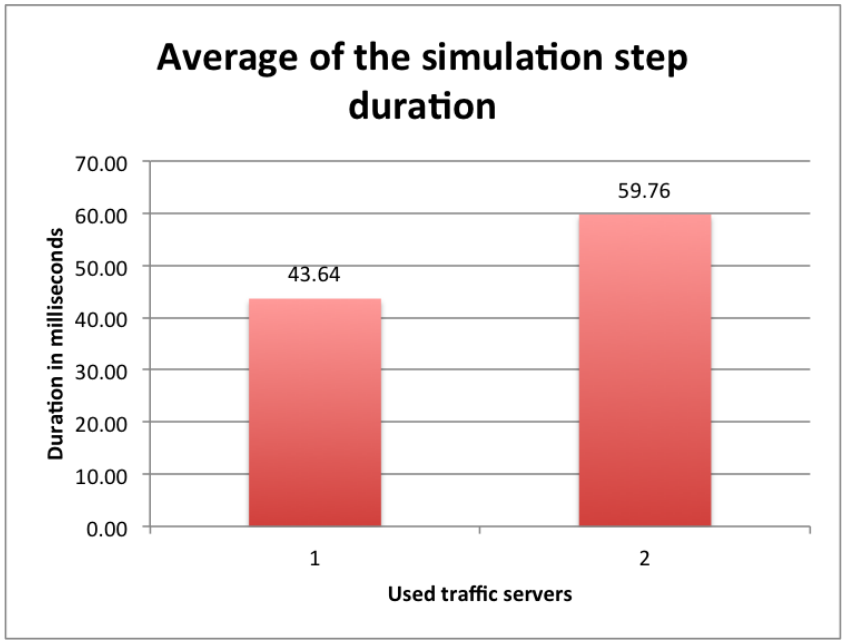


Figure 99: Comparison of the simulation step duration between the used traffic servers

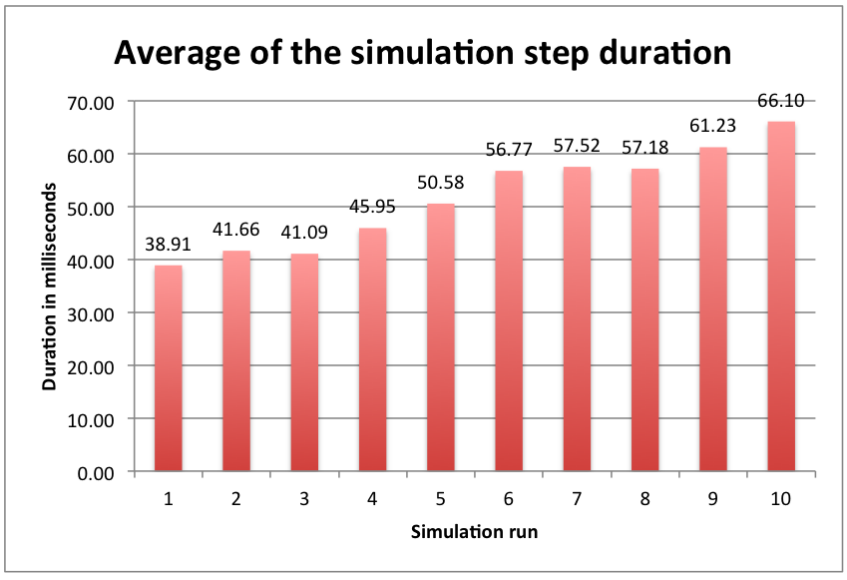


Figure 100: Average of the simulation step duration

The higher time durations of the simulation runs 2 to 5, in contrast to simulation run 7 to 10, are conditioned by the simulation step, where all cars are scheduled. Figure 101 highlights this simulation step.

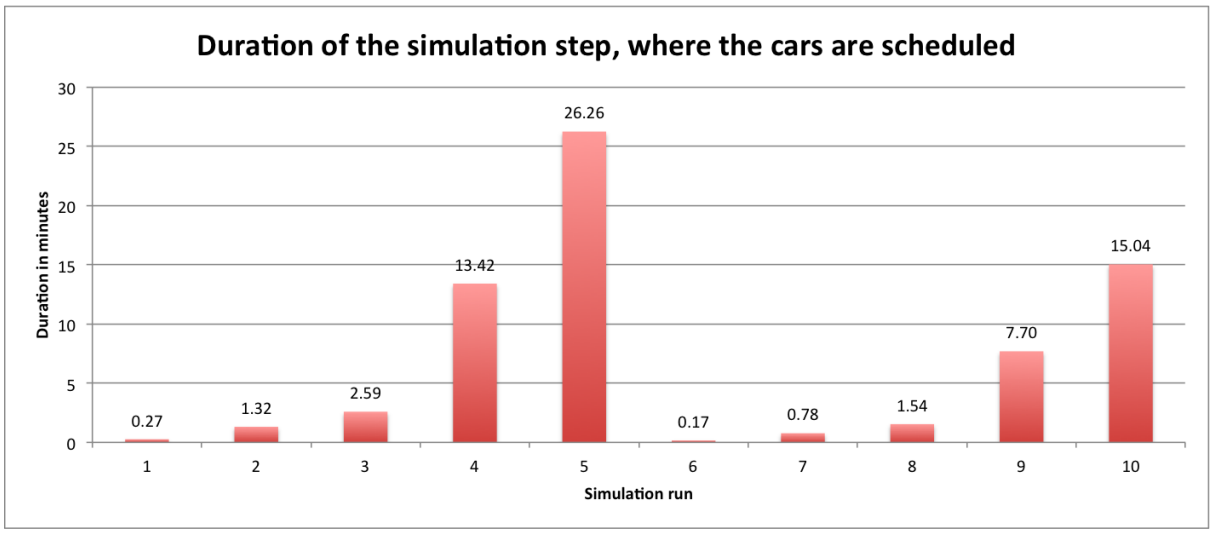


Figure 101: Duration of the simulation step, where the cars are scheduled

All the simulation runs 6 to 10 using two traffic servers take less time in comparison with the simulation runs 1 to 5 with only one traffic server. The difference between simulation run 1 and 6 is only 10 seconds, but the difference is already higher (about 30 seconds) between run 2 and 7, where 5.000 cars are scheduled. The duration of the simulation run 10 emphasizes the highest difference because the equivalent simulation run 5 needs about 11 minutes more.

The complete scale test with all data is saved in the file "*Scale test.xlsx*".

9.2.3 Scale test evaluation

First of all, the results of the scale test highlight the performance improvements by using more traffic servers and the advantage of the distributed architecture. The simulation run with 100.000 on two traffic servers ends about 11 minutes earlier in contrast to the simulation run with only one traffic server.

Take a look at the results. The initialization phase of the simulation runs with the distributed architecture needs a lot more time than the simulation runs with only one server. The reason is that the instance of the *InitParameter* is very big because it includes the big OpenStreetMap file mentioned in section5.2.9. This instance has to be shipped to all traffic servers. In detail, the SimulationController gets the *InitParameter* as copy by value parameter from the *Simulation Control Center* in the scale tests. Normally, this controller gets the *InitParameter* as copy by reference parameter because the *Simulation Control Center* and the *SimulationController* are located on the same server. After that the *SimulationController*calls over RPC all *FrontControllers*. As the result, the *FrontControllers* start and the *TrafficController*calls his *TrafficServers* over RPC with copy by value parameters. All of these calls lead to prolonged initialization and start phases that are illustrated in Figure 97.

The average duration of the individual simulation steps are higher if more server are used (see Figure 99). The cause is that there is a communication overhead by using more than one server. Nevertheless,

more servers can process in parallel and as a consequence they reach a lower total duration time (see Figure 98).

To reduce the duration of each simulation phase, the simulation must take the advantage of the multi processor architecture. Although there are some classes that are using different threads to execute algorithm parallel, the tests had not use this classes. For example the route calculations in the simulation start phase can clearly profit by exploiting multi core processing. In that simulation phase the routes of all vehicles are calculated. Thus, the simulation start phase needs more time by processing sequentially. Another reason for the higher processing time is that the routes are calculated randomly due to the fact that the all vehicles have random start positions and random destinations.

There are also other possibilities to optimize the simulation duration. For instances, caching of previous calculated routes can enhance significantly the simulation processing time. Another improvement could be to use another algorithm to calculate the routes. The scale test uses the Dijkstra's algorithm, which is not the fastest algorithm to find the shortest path between two points in a graph. Moreover, the individual servers should receive the*InitParameter* only once and after that they could distribute the*InitParameter* as a call by reference parameter to the other components located on the individual server. Due to the fact that the OpenStreetMap file is very big, the individual server could possess the needed OSM maps. So, a transfer is no longer necessary.

Another optimization possibility is to divide the graph on one traffic server and give each part to one thread. These threads could update the traffic participants in parallel per server. In the end the graph is divided into subgraphs for each traffic server and each subgraphs is also divided into smaller pieces for each thread running on an individual traffic server. This architecture could improve the duration of the simulation steps as well.

10 Demonstrations

The next section explains the demonstrations and impressions of the various events, where the project group showed their results to the public. A presentation in “Schlaues Haus Oldenburg”, the CeBIT 2013 as well as the BTW 2013 were the bigger events for the project group. Furthermore, there are three presentations in the future.

10.1 CeBIT 2013

From 05.03.2013 to 09.03.2013 the Project Group ALISE took the opportunity to present the Project “Die schlaue Stadt” to a bigger audience. The group could exhibit their results at the stand of the state Lower Saxony at CeBIT 2013 in Hannover, Germany. The product “Die schlaue Stadt” was shown to many people and to Mr. Neuffer from IBM.

Figure 102: CeBIT 2013 team

The press was also very interested in the project. There were several interviews with different newspapers (e.g. NWZ) and one interview with a German radio station (radio Bremen). The prime minister from Lower Saxony Stephan Weil and the Minister for Science and Culture in Lower Saxony

Dr. Gabriele Heinen-Kljajic also visited the stand as shown in Figure 102. They were enthusiastic and happy that we could show them what students in Lower Saxony can achieve. Both were very interested in our project and pleased to visit the stand "Die schlaue Stadt". Various urban planners where interested in the project "Die schlaue Stadt", they all were impressed, how the project and IT in general can help them manage their city in the (near) future. Many would like to see an adaptation of "Die schlaue Stadt" for their own city.

Professors, research associates, and company representatives from different university, companies and research labs expressed their interest in the project itself and further research collaborations. There were more than 30 people that were concerned about research collaborations between their company and the OFFIS respectively the University of Oldenburg. The Future Match conversations with representatives of several companies were also very stimulating. Many of them want further contact with the project group and with the OFFIS.

Mr. Zelder, economic development Oldenburg, visited the stand and was glad to see the outcome of the project, the economic development Oldenburg promoted.

Many private persons were interested in the stand and project itself. Most of them could identify the problems, the project tried to solve, in their own hometown. There were also many pupils and students from other universities that were interested in studying computer science and business informatics at the University of Oldenburg.

Most members of the project group could establish contacts to research and industry for their further career path.

10.2 BTW 2013

The conference of experts, BTW 2013, in Magdeburg from March 11th to 15th 2013 was a great success of the project group. BTW stands for database systems for business, technology and web and is a conference of experts with various presentations and demonstrations concerning databases and data streams.

The smart city application of the project group had enchanted a lot of conference participants with the result that the project "Die schlaue Stadt" won the award for the best demo. Furthermore, new contacts could be made and new perspectives of the project future were revealed by the various conversations between the project members and the other conference participants. Besides, it was the first conference of experts experience for the bigger part of the project group.

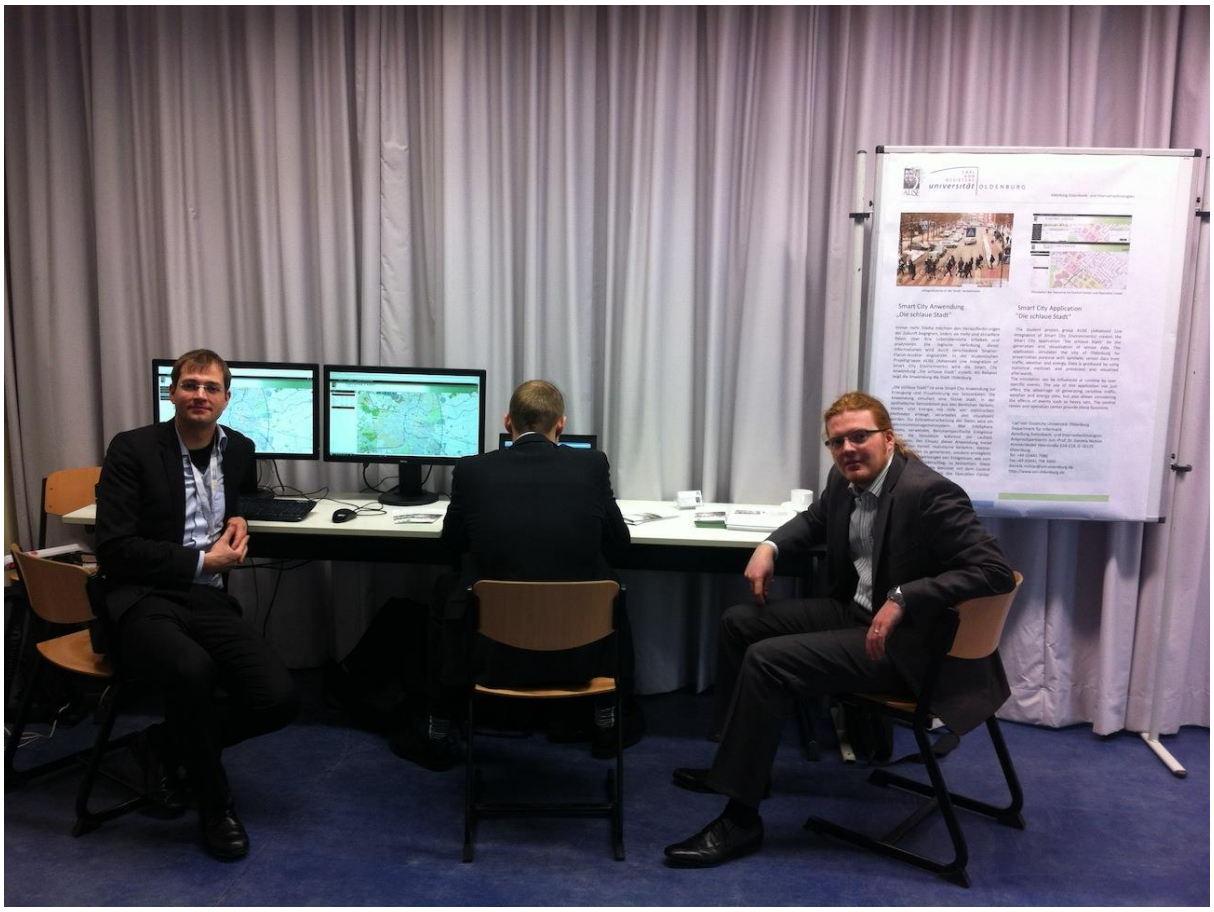


Figure 103: Demonstration of the project group at the BTW 2013

10.3 BUIS Days

The project group's paper for a presentation of the results on the fifth BUIS days was accepted. The BUIS days take place in Oldenburg from 24th to 26th of April 2013 and the project group looks forward to hold a poster presentation. The results of the project will be demonstrated, hopefully starting stimulating discussions.

10.4 IBM Smarter Planet Tour

With the help of IBM Germany the project group is demonstrating the software product at the TU Dresden as well as at the Carl von Ossietzky University Oldenburg. The expected progress of the demonstrations are like the days on the BTW 2013. Two monitors will show the Simulation Control Center and Operation Center during the whole demonstration time. A 20 minute presentation, which will be presented by two project group members, will give an overview about the created simulation framework. The dates for the events are on the 6th and 7th May at TU Dresden and on the 13th and 14th at the Carl von Ossietzky University Oldenburg.

11 Project Conclusion

In this conclusion the results and possible extensions of the results will be examined. After giving a summary of the Final Paper in general, the basics for further implementations will be given. Based on this knowledge some approaches for possible new modules for the simulation will be given.

11.1 Perspective

The following chapters give an overview about the perspectives of this project. They offer a short outlook to the possible extensions, like a framework with a high-level architecture or the adding of new sensors like pedestrians.

11.1.1 Outlook to extend the simulation framework with a high-level architecture

The next section gives an overview about the high-level architecture and its purpose. After that the main components are presented and the connection points with the created simulation framework of the project group are highlighted. This part concludes with a brief statement of the project group.

HLA stands for high-level architecture and is a general-purpose architecture for distributed computer simulation systems invented by US Department of Defense (DoD). HLA is the standard 1516 owned by the IEEE [FR13]. The three main components of the HLA standard make possible that individual computer simulations, also known as federates, can interact with each other in a so called federation (see Figure 104). The interoperability and the reusability of the individual software components are goals of the HLA. The used computing platform of each simulation has no impact on communicating data and synchronizing actions between the simulations because a Run-Time Infrastructure (RTI) manages the interoperability among these simulations. The communication between RTI and the federates uses the ambassador pattern to separate the simulation functionalities with the communication or coordination functionalities. The standard supports a network of individual software modules that do not be necessarily simulations [HM03].

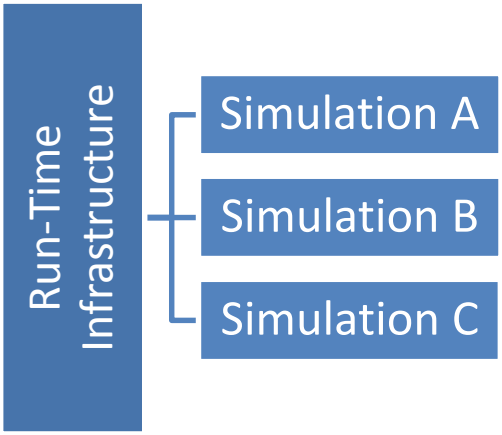


Figure 104: Federation with three federates

The following three main components of the HLA help to define the functional components, design rules and interfaces for the computer simulation systems:

1. Interface specifications
2. Object model template (OMT)
3. Rules

The interface specifications define the interfaces between the RTI and the federation and are available in terms of an API. The used APIs at OFFIS are CERTI[34] and PITCH pRTI[35]. The API has to provide six services to satisfy the HLA standard: federation management, declaration management, object management, data distribution management, time management, and ownership management.

The object model template (OMT) describes the federation object model (FOM) and the simulation object model (SOM). Which means, it defines the participation of the federation, the information that will be exchanged by the federates, and the public signatures (especially the methods and attributes) of the federates.

The rules specifies how the individual federates communicate with the RTI and with each other with the aid of the RTI. Moreover, they define design principles for the OMT as well as for the interface specifications. The different rules are separated into two section: five mandatory rules for the federates and five mandatory rules for the federation [NA09].

Rules for the federates:

1. Federates shall have an HLA simulation object model (SOM), documented in accordance with the HLA object model template (OMT).
2. Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.
3. Federates shall be able to transfer and/or accept ownership of an attribute dynamically during a federation execution, as specified in their SOM.
4. Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM.
5. Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

Rules for the federation:

1. Federations shall have an HLA federation object model (FOM), documented in accordance with the HLA object model template (OMT).

[34] https://savannah.nongnu.org/projects/certi/
[35] http://www.pitch.se/products/prti

2. In a federation, all representation of objects in the FOM shall be in the federates, not in the run-time infrastructure (RTI).
3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.
4. During a federation execution, federates shall interact with the run-time infrastructure (RTI) in accordance with the HLA interface specification.
5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

Due to the fact that the project group spent much time to realize a distributed simulation framework and implement various individual controller with complex logic and functionality, the members had no time to extend their simulation framework with a HLA functionality. Especially the rules one and four for the federation and the whole rules for federates have to be implemented in the simulation framework of the project groupto satisfy the HLA standard. Furthermore, is has to be chosen which RTI implementation will be used.

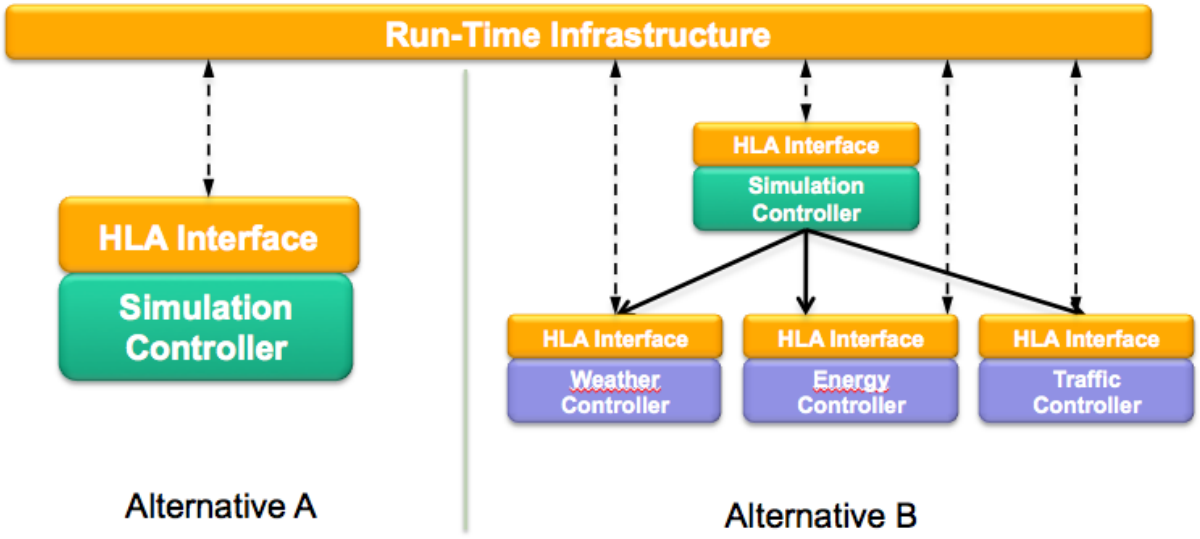


Figure 105: Different implementations for the HLA

Notice that the simulation framework of the project group is separated into individual software models that are coordinated by the simulation controller, it will be easier to switch to an HLA by replacing the simulation controller by the RTI or build an interface on top of the simulation controller which communicates with the RTI. Alternative it is possible to extend one of the controllers with RTI communication functionalities with the result that only individual components of the simulation framework are in line with HLA. The two alternatives are presented in Figure 105. The implementation and connection to other simulations, which are in line with HLA, is an excellent perspective for another project group.

11.1.2 Outlook to extend the simulation framework with pedestrians

To show an example for the extensibility of the architecture, the following section gives an overview about the simulation with a crowd-sensing scenario. We assume pedestrians that send their position and feelings with smartphone apps. To simulate this scenario, five different components need changes or extensions: the *shared* component, the *simulation control center*, the *Traffic controller,* the queries of the *Data stream management system*, and the *Operation center*.

- **Shared (see paragraph 5.2.9 on page 108):** A new sensor type needs to be added in the package shared. This can be done by adding it into the enum *de.pgalise.simulation.shared.sensor.SensorType*, e.g. "PEDESTRAIN". Because no other information is needed, the class *SensorHelper* does not need to be extended.
  Pedestrians have a similar behavior to vehicles, so a new vehicle type in the *de.pgalise.simulation.shared.traffic.VehicleType* enum can be created, e.g. PEDESTRAIN. Different models can be added in the *de.pgalise.simulation.shared.traffic.VehicleModel* enum, e.g. PEDESTRAIN_GIRL.
  Now the pedestrians can be added by using the *CreateRandomVehicleEvent* or *CreateVehicleEvent* in the package *de.pgalise.simulation.shared.event.traffic*.
- **Simulation control center (see paragraph 5.2.9 on page 108):** To add the new sensor type, the server- and the client-side of the simulation control center needs to be extended.
  The server-side needs, two extensions. The class *de.pgalise.simulation.controlCenter.internal.model.RandomVehicleBundle* needs to be extended by an amount for random added pedestrians. Also the setservice for *de.pgalise.simulation.controlCenter.internal.util.service.CreateRandomVehicleService* needs to the extended to handle the pedestrians.
  On the client-side, add the new vehicleTypes and Models to the enumerations in *js/model/enums*.js and extend the Java Script equivalents along the lines of the server-side classes, that is, the simulation start parameter object in *js/model/JAVA_utilityObjects.js* must be adjusted. Obviously, the user interface has to be extended to handle the pedestrians as well. To make the user able to set initial pedestrian amount and smart phone ratio, align the view in *partials/initialState/city.html* to the new requirements and extend the corresponding controller in *js/ctrl/contentCtrl/initialState/cityCtrl.js*.
  If events, for instance the AttractionEvent, shall include pedestrians as well, extend the event class in *js/model/eventObjects.js*, the Java Script-equivalent to the *AttrationData* object (present in *js/model/communicationObjects/JAVA_eventObjects.js*), as well as the corresponding view in *partials/dialogs/eventDialogs/attractionDialog.html* and the corresponding controller in *js/ctrl/dialogCtrl/attractionDialogCtrl.js*.
- **Traffic Controller (see paragraph 128 on page 128):** The TrafficController consists of several TrafficServer that perform the traffic relevant aspects of the simulation. Therefore you

have to take a closer look to the default implementation of the TrafficServer to add new traffic features.

Follow these instructions to introduce pedestrians.

1. Derive a new class from BaseVehicle for the pedestrians and implement the Vehicle interface.
2. Adjust the existing TrafficEventHandler to consider the new type of "vehicle". For example you have to customize the CreateRandomVehicleEventHandler to add random pedestrians along with bikes and motorized vehicles.
3. (Optional) If any new simulation event was added regarding to the pedestrian feature, you have to create a new TrafficEventHandler to handle it (s. 5.2.12.3).

- **Adding and changing behaviour of pedestrians**
    - With the previous added modifications the pedestrians will behave like other vehicles.
    - Pedestrian can be instanciated and walk from one point to another (e.g. by raising the CreateRandomVehicleEvent).
    - To change the behaviour either override the protected methods of the BaseVehicle class in the derived subclass for pedestrians or modify the corresponding vehicle event handler (see page 128128).
    - These changes allow you for instance to react whenever a pedestrian passes a node (e.g. pedestrian passes a node - node is annotated as bus stop - bus helps him/her get faster to his/her destination - stop walking and wait for the bus)
        - Note: If you want this bus feature work properly you have to also customize the implementation of the bus to pick up pedestrians and let them out again.
- **Data stream management system (see paragraph 5.2.7 on page 95):** The amount of work to do depends on the data the pedestrians would send.
    - Assuming they would just send the latitude and longitude of their position it is very easy to integrate them in the streams application. In this case they would use the given scheme. The only change to do is in the split operators (as it needs to be in all the following cases). There the new sensor type id of the pedestrian has to be added to correctly split the data stream.
    - If they are sending additional information, like the number of people around them with a smartphone, it is necessary that these data can match one of the attributes in the given input data stream. If it does, the GPS stream in InfoSphere Streams must be extended by this attribute and so the output will. In this case the operation center has to know about it. Or the pedestrian can be processed individually, so the split operator output this data on their own stream. As the result this data will gets their own TCP output and the operation center has to know about it, too.

- In the worst case when the input scheme has to be changed, the output of all sensors in the simulation has to be changed to, because all of them are using the same scheme. The output has to be changed in the sensors in the StaticSensorController (see 0), in the TrafficSensorController (see 4.6.7) and in the sensor in the SensorFramework (see 0). As a consequence the input operators in the DSMS needs to be changed, too. The next changes would be the same as described above.

- **Operation Center (see paragraph 5.2.4 on page 72):** The operation center needs extensions on the server- and client side.
  On the server-side is a new model needed that extends *de.pgalise.simulation.operationCenter.internal.model.sensordata.SensorData* by values for latitude, longitude and the feeling. It is also possible to extend *GPSData*, but then the data stream management system needs to calculate the extra values, e.g., for speed. This new model needs to be used in *de.pgalise.simulation.operationCenter.internal.DefaultOCSensorStreamController*. A new thread is needed, that can listen to the new pedestrian TCP/IP sink of the data stream management system and sends the updates to the *OCSimulationController*. For this purpose the "properties.props" file can be extended by the data stream management IP and port for the new TCP/IP sink.
  On the client side, the changed in code are rather complex, as new logic needs to be added in several areas of the application. A good way to approach the extention is to follow the logical path, starting at the servlet communication service (*js/modules/services/servletCommunicationServices.js*). It was tried to design the application data flows as generic as possible. However, performance was seen most important, which diminished the requirements towards extensibility to a certain degree.
  Extend the *applyNewVehiclesMessage* function in *js/modules/simulationModule.js* to react on the pedestrians and accordingly the private parseSensor function. Alter the *removeSensor* function to delete pedestrians. Moreover, extend the *applySensorDataMessage* function and the *updateSensor* function. In *js/model/sensors.js*, add a new sensor and a new class for the pedestrian in *js/model/vehicles.js*. Both will be instantiated in the *applyNewVehiclesMessage*. In the map service (*js/modules/services/mapServices.js*), add styles and a layer and push them into the *layerMap* along the lines of the other mobile sensors. Push the new layer to the *vectorLayers*. Alter the *addToFavorites* and *removeFromFavorites* functions and add styling logic to the strategy activation and deactivation functionss, which are denoted in the code. Lastly, extend the activation and deactivation functions in *js/ctrl/mainCtrl.js*. Obviously, the new sensor type needs a new icon which should be present in the *img* folder and referenced in *css/olStyle.css*.

11.2 Lessons Learned

During the last year the project group had to master many challenges. They all have been solved well but retrospectively things could have been done better during its solving process.

As already described the process model SCRUM has been used to plan and control the group's activities. But using SCRUM in its conventional sense was not the optimal way to satisfy the needs of a students' project group. To cope with the students requirements the "daily SCRUMS" were compressed to two days a week. In addition the "SCRUM Master" has changed all two weeks, so everybody could have the chance to get experience by taking the leading role in the project. Furthermore the role of the "SCURM-Master" could not be filled out conventionally, since everybody had to accomplish his or her tasks in addition to the role.
By taking SCRUM, the project group consequently had to split the tasks into an appropriate extend. In fact the task participations were not adequately elaborated at the beginning. As a result the Backlog milestones had to be delayed quiet often till the group learned how to define tasks decent. That's why SCRUM was not the best choice as process model because it does not fit perfectly for a students' project group.

Except for getting to know "Microsoft Sharepoint" as an enterprise software with its functionalities, the decision to use it also was a failure. The only used features were the calendar, the document repository and the time tracking. A task assignment would have been available but it was too time-consuming considering many little tasks everybody has gotten after the sprint planning. So only "Microsoft Excel" as backlog has been used for this purpose.

Another difficulty was to get familiar with "IBM Cognos" and "IBM InfoSphere Streams". Both are pretty complex and equally to handle. Since both provide a high number of functionalities it took a lot of time to install and adjust them in the given hardware infrastructure. Temporary it was planned to replace "IBM Cognos" with "IBM Jviews" in order to avoid the installing problems by simply programming the needed data-charts independently. But fortunately the "IBM Cognos" installation succeeded after all. To solve the problems, in the end ALISE had to make extensive use of IBM's support forums and a lot communication was needed till things ran smoothly. Consequently it took a lot of time.

A feasibility analysis considering the distributed architecture also needed time but it was a crucial requirement so it was actually entitled. Moreover in this case a progress was visible which made the invested time worth it. As a result the former service orientated architecture was superseded by the java enterprise edition specification.

11.3 Reflection of the project group

After one year that the project group was running through everybody has gotten experience as individual but also in the project group as an organization.

The majority of the group knew each other and partially worked together in different modules so far. So the people were attuned and knew with whom they get involved with. To strengthen this bond the group had several team events such as a go-cart race, dinners, barbecues, cinema events and things like that. Thus the social level of the team was harmonious.

Nevertheless the teamwork level was not likewise, since the each of the members lacked in different kind of know-how. That is why sometimes a discussion came up which could have last briefer. On the other hand it helped the team getting to a conclusion and a higher knowledge. In general when a lively discussion occurred and it was not propellant for whole group the “next cards” were raised. The next cards are devices for finding a general consent about when a topic is unimportant for the actual project status.

In the daily SCURMs some people were not necessarily interested in the work other people have done as far as it did not concern their actual work directly. In consequence they were distracted by the internet or their mobile. Since it is not quiet polite not to pay attention to someone else’s work a laptop and mobile ban has been established, so the participations had to pay attention to the speaker.

Because of the seminar phase nearly everybody has grown into her and his role of a specialist of their approached topic. At those topics which were not suitable for the project group the actual processors had to find another field of duty. But fortunately that was no big problem for the concerned persons.

List of Literature

[AG12] H.-J. Appelrath, D. Geesen, M. Grawunder, T. Michelsen, and D. Nicklas, "Odysseus: a highly customizable framework for creating efficient event stream management systems," in Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, New York, NY, USA, 2012, pp. 367–368.

[BBB13a] Behrendt, M., Böhm, M., Borchers, M., Caylak, M., Eylert, L., Friedrichs, R., Höting, D., Knefel, K., Lottmann, T., Rehfeldt, A., Runge, J., Schnabel, S.-C., Janssen, S., Nicklas, D., Wurst, M. : Virtuelle Sensordaten für Smart-City-Anwendungen, in Tagung der Fachgruppe Betriebliche Umweltinformationssysteme der Gesellschaft für Informatik e.V. (5. BUIS Tage), Oldenburg, April 2013.

[BBB13b] Behrendt, M., Böhm, M., Borchers, M., Caylak, M., Eylert, L., Friedrichs, R., Höting, D., Knefel, K., Lottmann, T., Rehfeldt, A., Runge, J., Schnabel, S.-C., Janssen, S., Nicklas, D., Wurst, M. : Simulation einer Stadt zur Erzeugung virtueller Sensordaten für Smart City Anwendungen, 15. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, Magdeburg, März 2013.

[BDG09] Behr, T.; Düntgen, C.; Güting, R.: BerlinMOD: A Benchmark for Moving Object Databases. In: The VLDB Journal 18:6 (2009), 1335-1368.

[Bri02] Brinkhoff, T.: A Framework for Generating Network-Based Moving Objects. In: GeoInformatica, Vol. 6, No. 2, 2002, pp. 153-180.

[CIS12] Cisco, Smart+Connected Communities: URL: http://www.cisco.com/web/strategy/smart_connected_communities.html (24.09.2012)

[DK09] Susanne Dirks and Mary Keeling. A vision of smarter cities: How cities can lead the way into a prosperous and sustainable future. 2009.

[FR13] Fraunhofer IESE, Verfahren - High Level Architecture. 2013. URL: http://www.software-kompetenz.de/?2432 (12.02.2013)

[HM03] Miegel, T., Holzmeier R. Interoperable Architekturen WS 02/03 - High Level Architecture (HLA). University Münster (2003).

[IBM12] IBM, Ein Planet der intelligenten Städte: http://www.ibm.com/smarterplanet/de/de/overview/visions/index.html (24.09.2012)

[IBM09] IBM. IBM O_ers Smarter City Assessment Tool to Help Cities Prepare for Challenges and Opportunities of Unprecedented Urbanization. 2009. URL: http: //www-03.ibm.com/press/us/en/pressrelease/27791.wss.

[Jenkins] Jenkins. Welcome to Jenkins CI! | Jenkins CI (2013). URL:http://jenkins-ci.org (As of 16.01.2013)

[NA09] NATO. ITEC09 HLA Tutorial - Manager's Guide to the High Level Architecture for Modelling and Simulation (HLA). URL: http://www.msco.mil/documents/_2_ITEC09%20-%20HLA%20Tutorial.pdf (12.02.2013)

[Sie11] Siemens: Smarter Neighborhoods - Smarter City: Solutions for a More Sustainable New YorkToday. 2011.

[SIE12] Siemens, Smarter Neighborhoods Smarter City: http://www.usa.siemens.com/sustainable-cities/pdf/smarter-neighborhoods-smarter-city.pdf (24.09.2012)

12 Appendix

1. Sprint 1 Review (07.06.12 – 20.06.12)

All participants have successfully finished their seminar paper. Furthermore the group created the basis for the ongoing development. Therefore they studied the according documents as well as dummy sensors. In Addition the first draft of the architecture has been defined and discussed with the product owners.

Particularly the sprint included the following task: IBM InfoSphere Streams has been installed. The homepage of the project group was created and internal and external areas were added. Some relevant documents have been translated in English. Another task was to create dummy sensors for energy, traffic, GPS and weather.

Some of the tasks couldn't be finished by the end of this sprint. They have to be continued in the next sprint. Those open tasks are the installation of IBM Cognos, the creation of some dummy sensors and Jenkins.

2. Sprint 2 Review (21.06.12 – 04.07.12)

The second sprint continued with the open tasks from the first sprint, as well as some new tasks. The errors with IBM InfoSphereStreams had to be corrected. The created documents from the previous sprint must be corrected and extended. Another huge task was the creation of the ESB-architecture with some test scenarios. In addition the realization of the simulation by using networks, dummy cars and nodes was another task.

After completing the second sprint the installation of IBM Cognos hasn't been finished yet. There were also problems using IBM InfoSphereStreams. The external area for the homepage has to be created in the following sprint and the final ESB architecture has to be reviewed by the whole group.

3. Sprint 3 Review (05.07.12 – 18.07.12)

Based on the reason that the members of the project group are in an exam phase, the third sprint was defined as a „light-sprint". The complexity of the tasks has been curbed. In a matter of fact the students had more time to study for their exams.

The tasks were separated in two main areas, the visualization and the simulation. The main task was to install IBM Cognos and fix the deployment error in IBM InfoSphereStreams. As well the visualization of the sensor data has been implemented with the help of Google maps.

The ESB-architecture, the creation of home and work nodes and the implementation of a buss network were some other tasks for the area of simulation.

Some of the tasks mentioned above couldn't be completely finished by the end of this sprint. These tasks are IBM Cognos and IBM InfoSphere Streams. These and other tasks were open for the next sprint.

4. Sprint 4 Review (19.07.12 – 01.08.12)

Based on the ongoing phase of exams the fourth sprint continued as a "light-sprint" as well.

The tasks are geared to the tasks from the previous sprint. Concerning to the continuing IBM Cognos problem, the group decided to get external help. Furthermore some new tasks have been added. These tasks include the generation of a weather detector and the street graph partitioning. Due to the fact that this sprint is a light sprint as well, not all of dispersed tasks have been totally completed. The tasks include without limitationIBM Cognos and the generation of home and work nodes from Open Street Map.

5. Sprint 5 Review (02.08.12 – 15.08.12)

As distinguished from the previous sprints (sprint 4 and 5), which were declared as "light-sprints" this sprint. Based on the ongoing problems by using IBM Cognos, Dr. Wurst from IBM Research in Dublin was contacted for some advice. If there is no solution for this problem within this sprint, the group will discuss an alternative solution to eliminate the problem.

The tasks have a bare reference to the street network for this sprint. The tasks include the generation of the VWG roadmap and the bus stops, the creation of an Open Street Map Parser, the creation of an energy sensor and the generation of home and work nodes.

Upon completion the sprint IBM Cognos has been installed completely. Therefore no alternative solution was needed. Most of the shared tasks have been solved. The generation of the sensors have been completed as well but some of them will be adapt in the following sprint.

6. Sprint 6 Review (16.08.12 – 29.08.12)

Based on organization problems the group decided to create milestones for the following sprint. The sprint has been spared into two phases. The first phase contains one week with the tasks to create controllers, servers and events. The second phase contains the creation of some class diagrams including the interfaces.

In fact that some tasks could not been finished completely the sprint has been extended for one more week. In this sprint the controller, server and events has been implemented and the UML diagrams have been finished. The method call by using AsyncCallbacks has been discarded because this type of method call works against the SOA concept. All Callbacks use EJB (Enterprise Java Beans). The integration test was presented to the product owner at the end of the sprint. This test ended with some

exceptions, so it has to be fixed in the following sprint. Furthermore it is necessary to implement bicycles in the simulation.

7. Sprint 7 Review (06.09.12 – 20.09.12)

Based on the errors in the integration test this sprint concentrates on the removal of those exceptions. The controller of energy, weather, simulation and traffic were arranged. The developed operation center should be integrated with a communication interface to the simulation. Furthermore all members of the group were informed about the functional and nonfunctional requirements. Another task was to write a BTW Paper until Monday the 17.09.2012.

Almost all of the distributed tasks have been done completely within the seventh sprint. Even the integration test has ended without any exceptions. The BTW paper was given to Daniela Nicklas.

8. Sprint 8 Review (20.09.12 – 04.10.12)

This sprint started with a visitation at the traffic control central of Oldenburg. Based on the ongoing cooperation between the OFFIS and the traffic control central, the project group has the permission to use some real sensor data from the bus traffic. The tasks of this sprint included the correction of the BTW paper until Wednesday, 26.09.2012, the completion of the test document, the implementation of bicycles and traffic light logic and some adaption on the Simulation Control Center.

Some of the untreated tasks have been inherited in the next sprint. The group got the task to complete the design document as another requirement. Furthermore, the whole group has made a raw draft of the halftime presentation. The implementation of the cyclist and the advancement of the traffic lights have been delayed.

9. Sprint 9 Review (04.10.12 – 18.10.12)

All open tasks from the previous sprint were inherited in this sprint. Some other tasks like finding attributes of cars or creating the halftime presentation and the demonstration of the system were added to this sprint.

Most of all tasks have been completed.

10. Sprint 10 Review (18.10.12 – 24.10.12)

The time of this 10th Sprint was reduced to six days. Due to the test presentation for the imminent halftime-presentation on Wednesday 24.10.2012 with Daniela Nicklas. Until Wednesday the system should run accurate. Some tasks were finished in this sprint like the implementation of busses or the integration of fuzzy logic.

This sprint ended with the presentation of the results. Daniela and Stephan were pleased about the current status of the project group and besides gave some suggestions of improvement. The official audition presentation is planned for the 7th November 2012.

The tasks from the sprint have been done mostly. Some problems with the simulation have to be fixed before the presentation.

11. Sprint 11 Review (24.10.12 – 07.11.12)

Tasks like the control- and operation center have been spread for the eleventh sprint. Another task was the creation of a tiny program for the VLZ Oldenburg. The bus network and the bicycles have be implemented in this sprint. To reach a bigger variance the fuzzy logic has to be extended.

This sprint ended with the half time presentation of the reached goals. This presentation contained the current status of the project and a visual demonstration of the system. It was given to a chosen auditorium. The auditorium had given some interesting suggestions about the project after the presentation. The group will discuss them in the next sprint.

12. Sprint 12 Review (07.11.12 – 29.11.12)

The undone tasks from the last sprint have to be done in this sprint. Those tasks were the traffic control by using traffic rules or tarffic lights, the concretization of the Nagel-Schreckenberg model and the connection to Odysseus.

Not all of the given tasks were totally finished in sprint 12. In a matter of fact only a few new tasks have been shared. A detailed milestone plan has been created at the end of this sprint. This plan contains the finalization of the system, the final acceptance, the different mess and the documentation.

13. Sprint 13 Review (21.11.12 – 05.12.12)

The undone tasks from the last sprint and some new tasks to reach the first milestone had to be done in this sprint. Those tasks are the connection to Odysseus, the creation of the exhibition flyer and the interaction between the fuzzy rules and the cars.

The milestone has not been reached. All the corresponding tasks have been suspended into the next sprint.

14. Sprint 14 Review (05.12.12 – 19.12.12)

To reach the milestone the group distributed concrete tasks to every member of the group. The tasks were the inclusion of the fuzzy logic, the partition of cars onto several servers, connection to Odysseus and the implementation of bicycles, bikes and trucks.

Nearly all of the chosen tasks have been completely done by the team. As well this sprint ends with a conference call to Michael Wurst. This call was necessary to discuss the current status of the project.

15. Sprint 15 Review (19.12.12 – 17.01.13)

In fact of the winter holidays, this sprint was declared as a light sprint. The tasks are the final document, the conference paper for the BUIS and the BTW paper and some other tasks.

This sprint ends after three weeks of work. Mostly all of the tasks were finished. One of the most important tasks was the definition of the simulation components like the Simulation Control Center, the Simulation Controller or the OSM Parser. Not all of those components were finished. They were delayed into the next sprint.

The second milestone was reached as well. Therefor all of the defined tasks were examined and marked as done or as in progress.

16. Sprint 16 Review (17.01.13 – 31.01.13)

The 16th Sprint began like the other ones with the distribution of the new tasks. Some important tasks were the connection to the HLA, the distribution to several servers or the implementation of the superior function.

Just a few tasks were totally completed at the end of this sprint. The task had to be postponed into the next Sprint. Those tasks for example were the HLA connection or the separation of the simulation onto several servers.

17. Sprint 17 Review (31.01.13 – 14.02.13)

This sprint was focused on the completion of the simulation. This was necessary because the team has to perform their end presentation on February 28th and they have to present their product on the CeBIT and BTW. Certainly all undone tasks from the last sprint were inherited into this sprint.

18. Sprint 18 Review (15.02.13 – 27.02.13)

In this sprint primarily tasks were distributed ensureaproper run of the system. Other tasks included the creation of the final presentation, the integration of fuzzy logic and parallelizing the route calculation.

The sprint ended with the presentation of the project in “Schlaues Haus” Oldenburg and with the final presentation and the simultaneous approval of the project at the OFFIS. Smaller notes related improvements were recorded and transferred in the next sprint to the system. Also, all the organizational tasks necessary for the CEBIT were done.

19. Sprint 19 Review (28.02.13 – 14.03.13)

This sprint was not a sprint like the once before. There were no daily Scrums because the group were presenting the system at the CEBIT in Hannover and at the BTW at Magdeburg. Never the less many bugs were fixed during these two weeks.

Both presentations were successful. The group got a lot of positive feedback. Even the prime minister of Lower Saxony Stephan Weil was interested in the system. At the end of the BTW the group won the award for the best live demonstration.

20. Sprint 20 Review (14.03.13 – 31.03.13)

The 20th sprint was the final sprint of the project. This sprint was concentrated on the final system (including all tests) and the final report.

21. Milestone Planning

This section illustrates the detailed milestone plan of the project group during the whole project phase:

01.04.2012 – Milestone 1

- Each project group member finishes and publishes the seminar paper.
- There is a presentation of each seminar topic.

25.06.2012 – Milestone 2

- The first version of a project homepage is online.
- A template for the design document is created.
- The sensors that will be implemented are defined.
- A database, IBM InfoSphere Streams and IBM Cognos are installed.

30.08.2012 – Milestone 3

- The final version of the project homepage is online.
- A template for the final paper is created.
- The first version of the architecture is defined.
- The database, IBM InfoSphere Streams and IBM Cognos are configured.
- Weather values will be collected for further purposes.

07.11.2012 – Milestone 4

- Each controller implemented a life cycle.
- The public methods of each controller are tested by a JUnit test.
- The energy controller and the traffic controller interact with the weather controller.
- The controller can communicate with each other.
- The simulation has implemented a first implementation of a traffic model to produce traffic jams.
- Weather events are implemented.
- The simulation can use the street network of Oldenburg.
- A car can drive based on the street network.
- Energy sensors measure energy consumption and energy production.
- There is a first runnable version of the Simulation Control Center and Operation Center.
- An exception concept is defined.
- A halftime presentation is created.

05.12.2012 – Milestone 5

- There is a runnable simulation with one traffic server based on a main program.
- The Nagel–Schreckenberg model includes the properties of the vehicles.
- Energy, weather and GPS sensors can be placed in the simulation.
- Energy, weather and traffic sensors produce a sensor output.
- A definition of the sensor interferer is done.
- IBM InfoSphere Streams processes the transmitted sensor output and forwarded its output to the servlets.
- The Simulation Control Center can import and export the start parameters.
- The Simulation Control Center and the Operation Center are finished according to the simulation progress.
- Every single Java class has at least an author.
- All JUnit tests can be executed successfully.

19.12.2012 – Milestone 6

- Two simulations with the same start parameters produce the same sensor output.
- The Simulation Control Center starts the simulation.
- The Simulation Control Center and the Operation Center use Open Layers.
- The amount of driving vehicles is influenced by the fuzzy logic.
- Bikes, trucks and motorcycles can be simulated.
- Vehicles and busses driving around based on the Nagel–Schreckenberg model.
- The vehicles are controlled by traffic rules (priority to the right and round about) and traffic lights.
- A fuzzy logic influences the energy consumption in the simulation.
- An interface for Odysseus is implemented.
- IBM InfoSphere Streams uses new operationsto process the sensor output.
- The chapter 3 of the final paper is done.
- All Java projects are described with an UML diagram

16.01.2013 – Milestone 7

- All CeBIT deadline are satisfied.
- A flyer and a poster for the BTW 2012 are created.
- The test protocol is reviewed and complete.
- The current general architecture of the application and each Java project are described in the final paper.
- Each controller can be distributed on another machine.

- There is a definition of a higher function.
- A first version of a distributed architecture is implemented.

30.01.2013 – Milestone 8

- The simulation is runnable based on more than one server.
- A perspective of a HLA interface is defined.
- A JUnit test confirms the determinism of the distributed architecture.
- The distributed architecture can be set up by the Simulation Control Center.
- The higher function is implemented.
- First scale tests are executed.

31.03.2013 – Milestone 9

- The release 1.0 of the smart city application is finished.
- There is at least a 10% performance improvement of the simulation.
- The source code is reviewed.
- Each project has documentation.
- The start of the application is documented by a tutorial.
- The source code is published at GitHub under an open source license.
- At least two movies illustrates the smart city application.
- There is a final presentation of the project group.

22. Test Protocol

With the help of the test protocol, the project group had an overview of each JUnit test class at any time of the project phase. The purpose of the test protocol was to determine a brief function description of the JUnit test class and the last test status. The structure of the test protocol is a table with twelve columns. The description of each column is the following:

1. Test number
2. Test name
3. Test description
4. Test character
5. Component
6. Package and class
7. Expected result
8. Actual result
9. Test result
10. Description of the error and further action
11. Test date
12. Executor

The Figure 106 illustrates an extract of the test protocol. The whole test protocol can be found in the excel file *Testdrehbuch.xlsx*.

Testnr.	Testname	Testbeschreibung	Testart	Komponente	Package / Klasse	erwarteter Ergebniszustand	eingetretendes Ergebnis	Testergebnis	Fehlerbeschreibung und Maßnahmen	Datum	Tester
1	DisassemblerTest	Testet die Klasse QuadrantDissembler	Komponententests	Disassembler	de.pgalise.simulation.services / DisassemblerTest.java	Der Test teilt den Graphen in verschiedene Quadranten auf.	Der Test wurde fehlerhaft beendet.	fehlgeschlagen	Probleme mit den EasyMocks	03.09.12	Mustafa
2	LinearGridDeployerTest	Testet die Klasse LinearGridDeployer	Komponententests	GridDeployer	de.pgalise.staticsensor.grid / LinearGridDeployerTest.java	Der Test soll das korrekte Aufteilen der Sensoren testen.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		06.09.12	Mustafa
3	DefaultWeatherControllerTest	Testet den Weather Controller	Komponententests	Controller	de.pgalise.simulation.weather.internal.services / DefaultWeatherControllerTest.java	Der Test den Lebenszyklus des EnergyControliers und holt ein neuen Wert.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
4	DefaultWeatherServiceTest	Testet den Weather Service	Komponententests	WeatherService	de.pgalise.simulation.weather.internal.services / DefaultWeatherServiceTest.java	Der Test testet alle öffentliche Funktionen des Wetterservices.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
5	XMLWeatherLoaderTest	Testet die Klasse XMLWeatherLoader	Komponententests	WeatherLoader	de.pgalise.simulation.weather.internal.dataloader / XMLWeatherLoaderTest.java	Der Test liest eine bestehende XML-Datei mit Wetterinformationen ein und speichert die Werte.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
6	CityClimateTest	Testet das Event "Stadtklima"	Komponententests	WeatherSimEventDecorator	de.pgalise.simulation.weather.internal.modifier / CityClimateTest.java	Der Test erstellt eine Beispielstadt mit Beispielwerten und testet die Veränderungen durch das Stadtklima.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
7	ColdDayEventTest	Testet ein Event "Cold Day"	Komponententests	WeatherEventDecorator	de.pgalise.simulation.weather.internal.modifier / ColdDayEventTest.java	Testet, ob das Event "Cold Day" eintritt.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
8	HotDayEventTest	Testet ein Event "Hot Day"	Komponententests	WeatherEventDecorator	de.pgalise.simulation.weather.internal.modifier / HotDayEventTest.java	Testet, ob das Event "Hot Day" eintritt.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
9	RainDayEventTest	Testet ein Event "Rain Day"	Komponententests	WeatherEventDecorator	de.pgalise.simulation.weather.internal.modifier / RainDayEventTest.java	Testet, ob das Event "Rain Day" eintritt.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
10	ReferenceCityTest	Testet das Event "Klima einer Referenzstadt"	Komponententests	WeatherSimEventDecorator	de.pgalise.simulation.weather.internal.modifier /ReferenceCityTest.java	Es wird getestet, ob die Wetterinformationen mit Hilfe von Daten einer Referenzstadt verändert werden.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
11	StormDayEventTest	Testet ein Event "Storm Day"	Komponententests	WeatherEventDecorator	de.pgalise.simulation.weather.internal.modifier / StormDayEventTest.java	Testet, ob das Event "Storm Day" eintritt.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		11.09.12	Andreas
12	DBPersistenceServiceTest	Testet die Klasse DBPersistenceService	Komponententests	SensorPersistenceService	de.pgalise.ssf.test / DBPersistenceServiceTest.java	Der Test soll Sensoren erstellen, in eine bestehende Datenbank schreiben und anschließend wieder löschen.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		12.09.12	Marcus
13	SensorDomainTest	Testet die Klasse SensorRegistryImpl	Komponententests	SensorRegistry	de.pgalise.ssf.test / SensorDomainTest.java	Testet, ob Sensoren in die SensorRegistry eingefügt, verändert oder gelöscht werden.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		12.09.12	Marcus
14	StaticSensorControllerTest	Testet den Static Sensor Controller	Komponententests	Controller	de.pgalise.staticsensor.services / StaticSensorControllerTest.java	Der Test soll verschiedene Funktionen des Static Sensor Controllers testen. Unter anderem die Funktionen: start, erstellen, status des Sensors, pausieren, fortfahren, zusammenfassen und löschen des Sensors.	Der Test wurde erfolgreich abgeschlossen.	erfolgreich		12.09.12	Marcus

Figure 106: Extract of the test protocol

23. MySQL trigger forthe weather collector

```
BEGIN
      /*Variablen f¸r den Cursor*/
      DECLARE windVelocity FLOAT DEFAULT 0;
      DECLARE temper FLOAT DEFAULT 0;
      DECLARE perceivedTemperature FLOAT DEFAULT 0;
      DECLARE lightIntensity FLOAT DEFAULT 0;
      DECLARE relativHumidity FLOAT DEFAULT 0;
      DECLARE rad FLOAT DEFAULT 0;
      DECLARE precipitionAmount FLOAT DEFAULT 0;
      DECLARE airPressure FLOAT DEFAULT 0;
      DECLARE windDirection FLOAT DEFAULT 0;
      DECLARE measureMinute INT;
      DECLARE measureHour VARCHAR(2);
      DECLARE measureTime VARCHAR(6);

      /*Cursor mit Select der Durchschnittswerte*/
      DECLARE cu CURSOR FOR
            select                    ROUND(IFNULL(avg(WIND_VELOCITY),
NEW.WIND_VELOCITY),3),
ROUND(IFNULL(avg(TEMPERATURE), NEW.TEMPERATURE),3),
ROUND(IFNULL(avg(PERCEIVED_TEMPERATURE),
NEW.PERCEIVED_TEMPERATURE),3),
                  IFNULL(avg(LIGHT_INTENSITY), NEW.LIGHT_INTENSITY),
ROUND(IFNULL(avg(RELATIV_HUMIDITY), NEW.RELATIV_HUMIDITY),3),
                  IFNULL(avg(WIND_DIRECTION), NEW.WIND_DIRECTION),
IFNULL(avg(RADIATION),NEW.RADIATION),
ROUND(IFNULL(avg(PRECIPITATION_AMOUNT), NEW.PRECIPITATION_AMOUNT),3),
                  IFNULL(avg(AIR_PRESSURE), NEW.AIR_PRESSURE)
            from weather_station_data
            where      DATE_FORMAT(MEASURE_DATE,      '%e.%c')      =
DATE_FORMAT(NEW.MEASURE_DATE,
'%e.%c')    /* Tag muss gleich sein */
                  and     DATE_FORMAT(MEASURE_DATE,      '%Y')      <=
DATE_FORMAT(NEW.MEASURE_DATE,
'%Y')             /* Jahr muss kleiner gleich sein */
                  and   MEASURE_TIME   <=   ADDTIME(NEW.MEASURE_TIME,
'0:10:0')                                                  /* 10
Minuten Toleranz nach oben und unter */
                  and   MEASURE_TIME   >=   SUBTIME(NEW.MEASURE_TIME,
'0:10:0');

      /*Cursor ^ffnen*/
      OPEN cu;
      FETCH  cu  into  windVelocity,  temper,  perceivedTemperature,
lightIntensity,
relativHumidity, windDirection, rad, precipitionAmount, airPressure;
            /* Zeit einstellen */
            SET measureMinute = TIME_FORMAT(NEW.MEASURE_TIME, '%i');
            SET measureHour = TIME_FORMAT(New.MEASURE_TIME, '%H');
            IF measureMinute <=15 THEN
                  SET measureTime = CONCAT(measureHour,'0000');
            ELSEIF measureMinute <=45 THEN
```

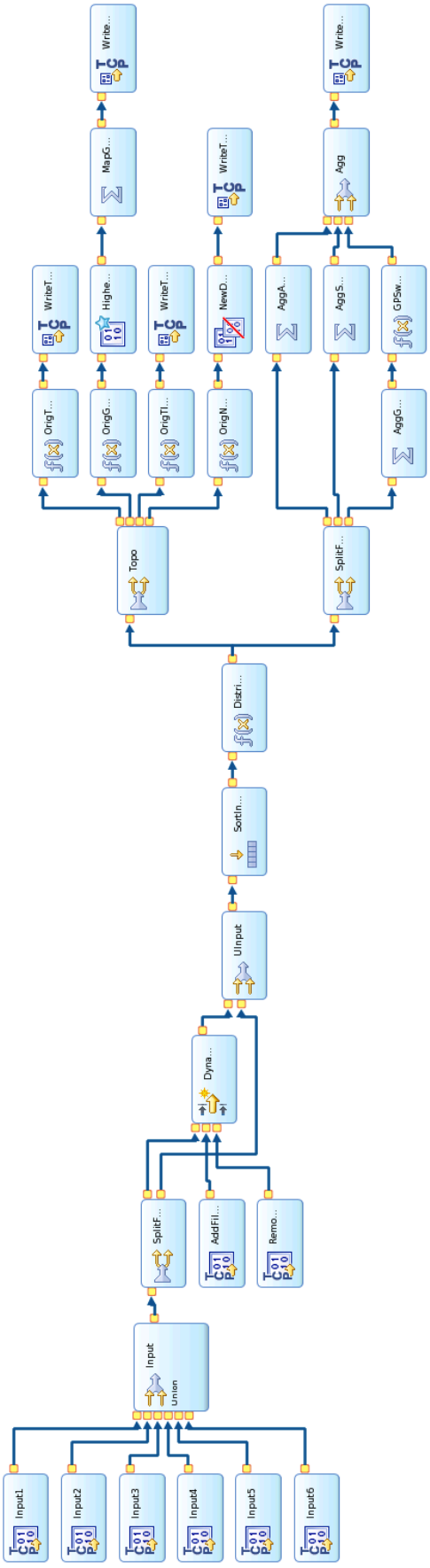
```
                SET measureTime = CONCAT(measureHour,'3000');
            ELSE
                SET measureTime = CONCAT(measureHour,'0000');
            END IF;
            /* Insert-Statement */
            insert into weather_aggregate_station (ID, MEASURE_DATE,
MEASURE_TIME,
WIND_VELOCITY, TEMPERATURE, PERCEIVED_TEMPERATURE,
                LIGHT_INTENSITY, RELATIV_HUMIDITY, WIND_DIRECTION,
RADIATION,
PRECIPITATION_AMOUNT, AIR_PRESSURE)
            VALUES (0, NEW.MEASURE_DATE, TIME_FORMAT(measureTime,
'%H:%i'),
windVelocity, temper, perceivedTemperature, lightIntensity,
                relativHumidity,        windDirection,        rad,
precipitionAmount, airPressure);
      CLOSE cu;
END
```

24. IBM DB2 trigger for the weather collector

```
CREATE OR REPLACE TRIGGER PGALISE.AGGREGATE_STATION_WEATHER AFTER
INSERT
ON PGALISE.WEATHER_STATION_DATA REFERENCING NEW AS NEW_ROW  FOR EACH
ROW
MODE DB2SQL
BEGIN
  DECLARE windVelocity DOUBLE;
  DECLARE temper DOUBLE;
      DECLARE perceivedTemperature DOUBLE;
      DECLARE lightIntensity DOUBLE;
      DECLARE relativHumidity DOUBLE;
      DECLARE rad DOUBLE;
      DECLARE precipitionAmount DOUBLE;
      DECLARE airPressure DOUBLE;
      DECLARE windDirection DOUBLE;
      DECLARE measureMinute INTEGER;
      DECLARE measureHour VARCHAR(2);
      DECLARE measureTime VARCHAR(8);

  Declare CU Cursor for
  Select                      Cast(ROUND(COALESCE(avg(WIND_VELOCITY),
NEW_ROW.WIND_VELOCITY),3)
AS DECIMAL(10,3)),
      Cast(ROUND(COALESCE(avg(TEMPERATURE),   NEW_ROW.TEMPERATURE),3)
AS
DECIMAL(10,3)),
      Cast(ROUND(COALESCE(avg(PERCEIVED_TEMPERATURE),
NEW_ROW.PERCEIVED_TEMPERATURE),3) AS DECIMAL(10,3)),
                  COALESCE(avg(LIGHT_INTENSITY),
NEW_ROW.LIGHT_INTENSITY),
      Cast(ROUND(COALESCE(avg(RELATIV_HUMIDITY),
NEW_ROW.RELATIV_HUMIDITY),3) AS DECIMAL(10,3)),
                  COALESCE(avg(WIND_DIRECTION),
NEW_ROW.WIND_DIRECTION),
      COALESCE(avg(RADIATION),NEW_ROW.RADIATION),
      Cast(ROUND(COALESCE(avg(PRECIPITATION_AMOUNT),
NEW_ROW.PRECIPITATION_AMOUNT),3) AS DECIMAL(10,3)),
                  COALESCE(avg(AIR_PRESSURE), NEW_ROW.AIR_PRESSURE)
  From PGALISE.weather_station_data
  WHERE       Cast(Day(MEASURE_DATE)       as       Char(2))       ||
Cast(Month(MEASURE_DATE) as
Char(2)) = Cast(Day(NEW_ROW.MEASURE_DATE) as Char(2)) ||
Cast(Month(NEW_ROW.MEASURE_DATE) as Char(2))
  AND       Cast(Year(MEASURE_DATE)       as       Char(4))       <=
Cast(Year(NEW_ROW.MEASURE_DATE)
as Char(4))
  AND MEASURE_TIME <= (NEW_ROW.MEASURE_TIME + 10 MINUTE)
  AND MEASURE_TIME > (NEW_ROW.MEASURE_TIME - 10 MINUTE);

  OPEN CU;
      FETCH  CU  into  windVelocity,  temper,  perceivedTemperature,
lightIntensity,
```

```
relativHumidity, windDirection, rad, precipitionAmount, airPressure;

    IF precipitionAmount < 0 THEN
      SET precipitionAmount = 0;
    END IF;

    set   measureMinute   =   Cast(MINUTE(NEW_ROW.MEASURE_TIME)   as
Char(2));
    set   measureHour   =   trim(Cast(HOUR(NEW_ROW.MEASURE_TIME)   as
Char(2)));

    IF measureMinute <=15 THEN
                SET measureTime = CONCAT(measureHour,'.00.00');
            ELSEIF measureMinute <=45 THEN
                SET measureTime = CONCAT(measureHour,'.30.00');
            ELSE
                SET measureTime = CONCAT(measureHour,'.00.00');
            END IF;

    /* Insert-Statement */
            insert       into       PGALISE.weather_aggregate_station
(MEASURE_DATE, MEASURE_TIME,
WIND_VELOCITY, TEMPERATURE, PERCEIVED_TEMPERATURE,
                LIGHT_INTENSITY,  RELATIV_HUMIDITY,  WIND_DIRECTION,
RADIATION,
PRECIPITATION_AMOUNT, AIR_PRESSURE)
            VALUES      (NEW_ROW.MEASURE_DATE,      TIME(measureTime),
windVelocity, temper,
perceivedTemperature, lightIntensity,
                relativHumidity,         windDirection,         rad,
precipitionAmount, airPressure);
  Close CU;
END
```

25. InfoSphere Streams application graph



26. InfoSphere Streams application (SPL Source Code)

```
namespace applicationTest ;

use com.ibm.streams.geospatial.twodimension.geometry::* ;
use com.ibm.streams.geospatial.geometry::* ;
use spl.math::* ;

composite AdvancedAliseProcessing
{
      graph

// The source operator for the first server (Static Sensors)
stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, uint8
axleCountOrStatus, uint16 lengthOrIntersectionID, uint16
axialDistance1, uint16 axialDistance2> Input1 =
                  TCPSource()
            {
                  param
                        role : server ;
                        port : 12545u ;
                        format : bin ;
                        reconnectionPolicy : InfiniteRetry ;
                        parsing : fast ;
                        //receiveBufferSize : 1048576u ; //1MB

                  config
                        restartable : true ;
                        relocatable : false ;
            }

// The source operator for the second server (Traffic)
stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, uint8
axleCountOrStatus, uint16 lengthOrIntersectionID, uint16
axialDistance1, uint16 axialDistance2> Input2 =
                  TCPSource()
            {
                  param
                        role : server ;
                        port : 12546u ;
                        format : bin ;
                        reconnectionPolicy : InfiniteRetry ;
                        parsing : fast ;
                        //receiveBufferSize : 1048576u ; //1MB

                  config
                        restartable : true ;
                        relocatable : false ;
            }
```

```
// The source operator for the third server (Traffic)
stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, uint8
axleCountOrStatus, uint16 lengthOrIntersectionID, uint16
axialDistance1, uint16 axialDistance2> Input3 =
                TCPSource()
          {
                param
                      role : server ;
                      port : 12547u ;
                      format : bin ;
                      reconnectionPolicy : InfiniteRetry ;
                      parsing : fast ;
                config
                      restartable : true ;
                      relocatable : false ;
          }

// The source operator for the fourth server (Traffic)
stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, uint8
axleCountOrStatus, uint16 lengthOrIntersectionID, uint16
axialDistance1, uint16 axialDistance2> Input4 =
                TCPSource()
          {
                param
                      role : server ;
                      port : 12548u ;
                      format : bin ;
                      reconnectionPolicy : InfiniteRetry ;
                      parsing : fast ;
                config
                      restartable : true ;
                      relocatable : false ;
          }

// The Input Operator for the fifth Server (Traffic)
stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, uint8
axleCountOrStatus, uint16 lengthOrIntersectionID, uint16
axialDistance1, uint16 axialDistance2> Input5 =
                TCPSource()
          {
                param
                      role : server ;
                      port : 12549u ;
                      format : bin ;
                      reconnectionPolicy : InfiniteRetry ;
                      parsing : fast ;
                config
                      restartable : true ;
                      relocatable : false ;
          }
```

```
// The Input Operator for the sixth Server (Traffic)

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64     measureValue1,     float64     measureValue2,     uint8
axleCountOrStatus,     uint16     lengthOrIntersectionID,     uint16
axialDistance1, uint16 axialDistance2> Input6 =
                  TCPSource()
            {
                  param
                        role : server ;
                        port : 12550u ;
                        format : bin ;
                        reconnectionPolicy : InfiniteRetry ;
                        parsing : fast ;
                  config
                        restartable : true ;
                        relocatable : false ;
            }

//Union off all Inputs to one stream

stream<Input1> Input = Union(Input1 ; Input2 ; Input3 ; Input4 ;
Input5 ;
                  Input6)
            {
            }

//Splitting the stream in tuple containing gps-data and non gps-data

(stream<Input1> SplitForDynGPS ; stream<Input1> NormalInput) =
                  Split(Input)
            {
                  param
                        index   :   sensorTypeID   ==(uint8)   2u   ||
                        sensorTypeID ==(uint8) 3u ||
                        sensorTypeID  ==(uint8)  4u  ||  sensorTypeID
                        ==(uint8) 19u || sensorTypeID
                        ==(uint8) 20u ? 0 : 1 ;
            }

// Input operator for the DynamicFilter. This Operator will get an
amount of allowed sensorIds. Only these ids will be passed through.

stream<uint32 sensorId> AddFilterKeysGPS = TCPSource()
            {
                  param
                        role : server ;
                        port : 12345u ;
                        format : bin ;
                        reconnectionPolicy : InfiniteRetry ;
                        parsing : fast ;
                  config
                        restartable : true ;
                        relocatable : false ;
            }
```

```
//Input Operator for the DynamicFilter. This Operator will get the
sensorIds which are not allowed anymore. These ids will be deleted
from the list of allowed ones.

stream<uint32 sensorId> RemoveFilterKeysGPS = TCPSource()
            {
                  param
                        role : server ;
                        port : 12346u ;
                        format : bin ;
                        reconnectionPolicy : InfiniteRetry ;
                        parsing : fast ;
                  config
                        restartable : true ;
                        relocatable : false ;
            }

//This operator lets pass the tuple with the allowed sensorIds. The
addKey data stream adds allowed sensorIds on runtime to a list.
// The removeKey data stream deletes sensorIds on runtime. These
operator makes possible to dynamic change the amount of gps-data on
runtime.

stream<Input> DynamicFilGPS = DynamicFilter(SplitForDynGPS ;
                  AddFilterKeysGPS ; RemoveFilterKeysGPS)
            {
                  param
                        key : SplitForDynGPS.sensorId ;
                        addKey : AddFilterKeysGPS.sensorId ;
                        removeKey : RemoveFilterKeysGPS.sensorId ;
            }

//This operator unions the filtered gps-data tuple with the other
tuples from the source operator.

stream<Input> UInput = Union(DynamicFilGPS ; NormalInput)
            {
            }

// To ensure the right order of the tuple, this operator sorts the
data stream by the timestamp and in an ascending order

stream<Input> SortInput = Sort(UInput as inPort0Alias)
            {
                  window
                        inPort0Alias : sliding, count(50) ;
                  param
                        sortBy : currentMillis ;
                        order : ascending ;
            }
```

```
//This operator duplicates the stream.

stream<Input> DistributeStream = Functor(SortInput)
            {
            }


//This operator splits the stream into four streams depending on the
sensor type, because different sensor types have different schemes,
processing and output ports (later on). There will be one data stream
for the GPS sensors, one for the traffic lights, one for the topo
radars and for the remaining static sensors.

(stream<SortInput> Topo ; stream<SortInput> GPS ; stream<SortInput>
Tlight ;
                  stream<SortInput> Normal) = Split(DistributeStream)
            {
                  param
                        index : sensorTypeID ==(uint8) 18u ? 0 :
                        sensorTypeID ==(uint8) 2u ||
                        sensorTypeID ==(uint8) 3u || sensorTypeID
                        ==(uint8) 4u || sensorTypeID
                        ==(uint8) 19u || sensorTypeID ==(uint8) 20u ?
                        1 : sensorTypeID ==(uint8)14u ? 2 : 3 ;
                  }

//This operator changes the scheme to the specific topo one and drops
all unnecessary attributes.

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
uint8 axleCountOrStatus, uint16 length, uint16 axialDistance1, uint16
axialDistance2> OrigTopo = Functor(Topo)
                  {
                        output
                              OrigTopo:length=lengthOrIntersectionID;
                  }

//This operator changes the scheme to the specific gps one and drops
all unnecessary attributes.

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2> OrigGPS = Functor(GPS)
                  {
                  }

//This operator changes the scheme to the specific static sensors one
and drops all unnecessary attributes.

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1> OrigNormal = Functor(Normal)
                  {
                  }
```

```
//This operator changes the scheme to the specific traffic light one
and drops all unnecessary attributes.

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, uint8 status, uint16
intersectionID> OrigTlight =
                        Functor(Tlight)
                  {
                        output
                              OrigTlight: status = axleCountOrStatus,
                                          intersectionID=
                                          lengthOrIntersectionID;
                  }

//This operator deletes the duplicates in the data stream of the
static sensors. In a window of 0.5 seconds each sensorId will be
proven if their measureValue1 has changed. If it has not changed the
tuple will be dropped, otherwise it will be output.

stream<OrigNormal> NewDeDuplicateNormal = DeDuplicate(OrigNormal)
                  {
                        param
                              timeOut : 0.5 ;
                              key : sensorId, measureValue1 ;
                  }


// This custom operator generates higher information out of the
latitude and longitude with the geospatial toolkit of InfoSphere
Streams 3. In this operator the actual speed, distance and total
distance, the travel time and direction will be calculated.

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, float64 distanceV,
uint32 totalDistance, uint8 speed, uint16 direction, uint64
travelTime>
                  HigherGPS = Custom(OrigGPS as inPort0Alias)
                  {
                        logic
                              state :
                              {
                              //Indices of the map:
                              //0 = old latitude
                              //1 = old longitude
                              //2 = new latitude
                              //3 = new longitude
                              //4 = Cursor
                              //5 = totalDistance
                              //6 = last timestamp
                              //7 = first timestamp
      mutable map<uint32, list<float64> [ 8 ]> PointsPerSensor ;
      mutable list<float64> [ 8 ] temp2 ;
                              }

                              onTuple inPort0Alias :
                              {
      mutable PointPropertyType.point128 pNew =
```

```
        com.ibm.streams.geospatial.twodimension.geometry::createPoint128(measureValue
1, measureValue2) ;
        mutable PointPropertyType.point128 pOld ;
        mutable boolean tst = has(PointsPerSensor, sensorId) ;
                                        if(tst)
                                        {
                                        mutable float64 cursor ;
                                        mutable list<float64> [ 8 ] temp ;
                                        temp = PointsPerSensor [ sensorId] ;
                                        cursor = temp [ 4 ] ;
                                        if(cursor == 3f)
                                        {
                                                cursor = 1f ;
                                                temp [ 0 ] = measureValue1 ;
                                                temp [ 1 ] = measureValue2 ;
                                                }

                                        else
                                                {
                                                cursor = 3f ;
                                                temp [ 2 ] = measureValue1 ;
                                                temp [ 3 ] = measureValue2 ;
                                                }

                                                temp [ 4 ] = cursor ;
                                                if(cursor == 1f)
                                                {
                                                        pOld =

        com.ibm.streams.geospatial.twodimension.geometry::createPoint128(temp
                                                                        [ 2 ], temp [ 3 ]) ;
                                                }

                                                else
                                                {
                                                        pOld =

        com.ibm.streams.geospatial.twodimension.geometry::createPoint128(temp
                                                                        [ 0 ], temp [ 1 ]) ;
                                                }

        //Distance calculation
                                                mutable float64 distanceInM =

        com.ibm.streams.geospatial.twodimension.geometry::distance(Metric.Spherical,
                                                        pOld, pNew) ;
                mutable float64 totalDistance = temp [ 5 ] + distanceInM ;
                temp [ 5 ] = totalDistance ;
        //Speed
                mutable float64 oldMillis = temp [ 6 ] ;
                mutable float64 difInSec =((float64) currentMillis - oldMillis) /
                                                        1000f ;
                mutable float64 speedInKmh =(distanceInM / difInSec) * 36f ;

        //Direction
                mutable float64 azimuth =

        com.ibm.streams.geospatial.twodimension.geometry::azimuth(Metric.Spherical,
                                                        pOld, pNew) ;
                azimuth = azimuth * 180f / PI() ;
                mutable uint8 returnAzi =(uint8) azimuth ;
                temp [ 6 ] =(float64) currentMillis ;
                PointsPerSensor [ sensorId ] = temp ;

        //Output
```

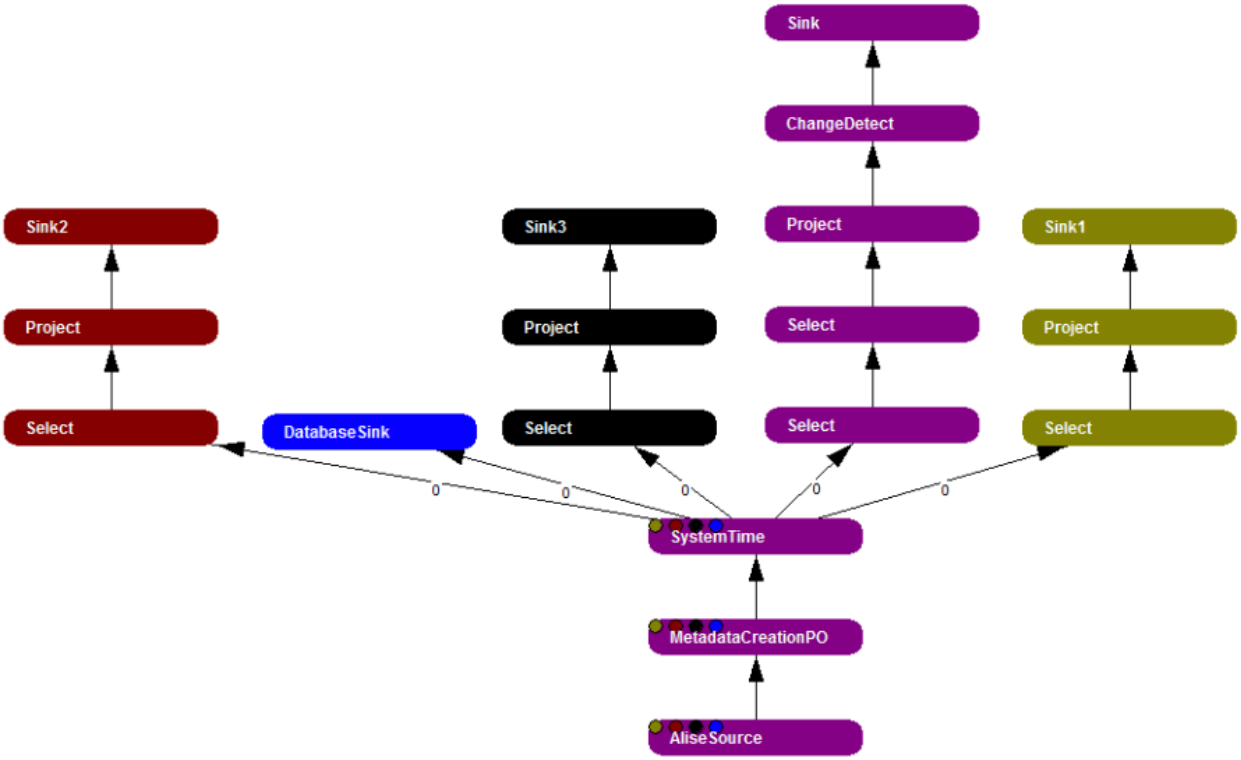
```
                    submit({ currentMillis = currentMillis, sensorId = sensorId,
sensorTypeID=   sensorTypeID,   measureValue1   =   measureValue1,   measureValue2
=measureValue2,   distanceV   =(float64)   distanceInM,   totalDistance=(uint32)
totalDistance, speed =(uint8) speedInKmh, direction =(uint16)returnAzi, travelTime
=(uint64)(temp [ 6 ] - temp [ 7 ]) },HigherGPS) ;
                                        }
                    //for the first run
                                else
                                {
                                        temp2 [ 0 ] = measureValue1 ;
                                        temp2 [ 1 ] = measureValue2 ;
                                        temp2 [ 2 ] = 0f ;
                                        temp2 [ 3 ] = 0f ;
                                        temp2 [ 4 ] = 1f ;
                                        temp2 [ 5 ] = 0f ;
                                        temp2 [ 6 ] =(float64) currentMillis ;
                                        temp2 [ 7 ] =(float64) currentMillis ;
                                        PointsPerSensor [ sensorId ] = temp2 ;

submit({ currentMillis = currentMillis, sensorId = sensorId,sensorTypeID =
sensorTypeID, measureValue1 = measureValue1,measureValue2 = measureValue2, distanceV
= 0f, totalDistance = 0u,speed =(uint8) 0, direction =(uint16) 0, travelTime
=(uint64) 0 },HigherGPS) ;
                                        }

                                }

                    }

//With this aggregate operator the average speed per vehicle type
will be calculated.

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, float64 distanceV,
uint32 totalDistance, uint8 speed, uint8 avgSpeed, uint16 direction,
uint64 travelTime> MapGPS = Aggregate(HigherGPS)
                    {
                          window
                                HigherGPS : sliding, count(100) ;
                          param
                                groupBy : sensorTypeID ;
                          output
                                MapGPS : avgSpeed = Average(speed) ;
                    }

// This operator creates a TCP sink for the static sensors.

               () as WriteTcp = TCPSink(NewDeDuplicateNormal)
                    {
                          param
                                role : server ;
                                port : 12600u ;
                                flush : 1u ;

                    }
```

```
//This operator creates a TCP sink for the GPS sensors.

            () as WriteGPSTcp = TCPSink(MapGPS)
                  {
                        param
                              role : server ;
                              port : 12601u ;
                              flush : 1u ;
                  }

//This operator creates a TCP sink for the topo radar.

            () as WriteTopoRadarTcp = TCPSink(OrigTopo)
                  {
                        param
                              role : server ;
                              port : 12602u ;
                              flush : 1u ;
                  }
//This operator creates a TCP sink for the traffic light.

            () as WriteTlightTcp = TCPSink(OrigTlight)
                  {
                        param
                              role : server ;
                              port : 12603u ;
                              flush : 1u ;
                  }

//This operator splits the stream into three streams depending on
their sensor type and their reasonable aggregation behavior.

(stream<SortInput> SplitForAggAvg ; stream<SortInput> SplitForAggSum
; stream<SortInput> NoOut) = Split(DistributeStream)
                  {
                        param
                              index : sensorTypeID ==(uint8) 0u ||
                              sensorTypeID     ==(uint8) 1u ||
                              sensorTypeID     ==(uint8)     5u     ||
                              sensorTypeID     ==(uint8)     8u     ||
                              sensorTypeID     ==(uint8)     9u     ||
                              sensorTypeID    ==(uint8)    10u     ||
                              sensorTypeID ==(uint8) 13u
                              ||   sensorTypeID   ==(uint8)   15u   ||
                              sensorTypeID ==(uint8) 12u ||
                              sensorTypeID  ==(uint8)  16u  ?  0  :
                              sensorTypeID ==(uint8) 6u ||
                              sensorTypeID     ==(uint8)     7u     ||
                              sensorTypeID ==(uint8) 11u ? 1 : 2 ;
                        }
```

```
//This operator counts the vehicle or sensors of a GPS sensor type
every 10 minutes simulation time.

stream<uint64 currentMillis, uint32 sensorId, uint8 sensorTypeID,
float64 measureValue1, float64 measureValue2, uint8
axleCountOrStatus, uint16 lengthOrIntersectionID, uint16
axialDistance1, uint16 axialDistance2, int32 count>
                        AggGPS = Aggregate(NoOut)
                        {
                              window
                                    NoOut : tumbling,
                                    delta(currentMillis,(uint64)
                                    100000u) ;
                              param
                                    groupBy : sensorTypeID ;
                              output
                                    AggGPS : count =
                                    CountDistinct(sensorId) ;
                        }

//This operator maps the attribute count to the measureValue1 to keep
the normal scheme.
                  stream<SortInput> GPSwCount = Functor(AggGPS)
                        {
                              output
                                    GPSwCount : measureValue1
                                    =(float64) count ;
                        }

//This operator takes the average measure value and the last
timestamp of a sensor every 10 minutes simulation time.

stream<SortInput> AggAvg = Aggregate(SplitForAggAvg)
                        {
                              window
                                    SplitForAggAvg : tumbling,
                                    delta(currentMillis,(uint64)
                                    100000u) ;
                              param
                                    groupBy : sensorId ;
                              output
                                    AggAvg : measureValue1 =
                                    Average(measureValue1),
                                    currentMillis =
                                          Max(currentMillis) ;
                        }
```

```
//This operator takes the sum of measure value and the last timestamp
of a sensor every 10 minutes simulation time.

stream<SortInput> AggSum = Aggregate(SplitForAggSum)
                        {
                              window
                                    SplitForAggSum     :     tumbling,
                                    delta(currentMillis,(uint64)
                                    100000u) ;
                              param
                                    groupBy : sensorId ;
                              output
                                    AggSum     :     measureValue1     =
                                    Sum(measureValue1),  currentMillis
                                    = Max(currentMillis) ;
                        }

//This operator unions the three different aggregated streams into
one stream.

stream<SortInput> Agg = Union(AggAvg ; AggSum ; GPSwCount)
                        {
                        }

//This operator creates a TCP sink for the aggregated data. After all
this data stream will be saved in database (DB2) and used in Cognos
for reporting.

                  () as WriteAggTcp = TCPSink(Agg)
                        {
                              param
                                    role : server ;
                                    port : 12604u ;
                                    flush : 1u ;
                        }

            }
```

27. Odysseus application graph



28. Odysseus application (Source Code)

PQLSOURCE

```
///OdysseusScript
#PARSER PQL
#TRANSCFG Standard

///This operator defines the tcp source for the sensor data streams.
It uses
/// MarkerBytes at the beginning and at the end (-1). This source is
listening on his port
/// for tuple to come in.
#QUERY
AliseSource := ACCESS({source='AliseSource', wrapper='GenericPush',
transport='TCPServer',protocol='MarkerByteBuffer',
dataHandler='Tuple',
 options=[['host','localhost'],['port','54321'],['start','-
1'],['end','-1'], ['ByteOrder','BIG_ENDIAN']],
schema=[
 ['timestamp', 'LONG'],
 ['id','INTEGER'],
 ['sensorTypeId','byte'],
 ['measureValue1','DOUBLE'],
 ['measureValue2','DOUBLE'],
 ['axleCount','byte'],
 ['length','short'],
 ['axialDistance1','short'],
 ['axialDistance2','short']
 ]})
```

SELECTROUTE

```
///OdysseusScript
#PARSER CQL
#TRANSCFG Standard
///These select operations split the data stream into four data
streams depending
/// on the specific sensor type.
#ADDQUERY
create view OutputNormal from ( Select timestamp, id, sensorTypeId,
measureValue1 from AliseSource where sensorTypeId <> 2 AND
sensorTypeId <> 3 AND sensorTypeId <> 4 AND sensorTypeId <> 18 AND
sensorTypeId <> 19 AND sensorTypeId <> 20 AND sensorTypeId <> 14 );
#ADDQUERY
create view OutputGPS from (select timestamp, id, sensorTypeId,
measureValue1, measureValue2 from AliseSource where sensorTypeId = 2
or sensorTypeId = 3 or sensorTypeId = 4 or sensorTypeId = 19 or
sensorTypeId = 20);
#ADDQUERY
create view OutputTopo from (select timestamp, id, sensorTypeId,
axleCount, length, axialDistance1, axialDistance2 from AliseSource
where sensorTypeId = 18);
#ADDQUERY
```

```
create view OutputTlight from (select timestamp, id, sensorTypeId,
measureValue1, measureValue2, axleCount from AliseSource where
sensorTypeId = 14);
```

MYSQLSINK

```
///OdysseusScript
#PARSER CQL
#TRANSCFG Standard
///This operator creates the connection to the database and the
associated database sink.
#QUERY
CREATE DATABASE CONNECTION duemmer AS mysql TO odysseus AT
134.106.11.89 : 10832 WITH USER root PASSWORD pg20122013Plizzard
#QUERY
CREATE SINK dbSink AS DATABASE duemmer TABLE ody_output
```

MYSQLSELECT

```
///OdysseusScript
#PARSER CQL
#TRANSCFG Standard
///This operator streams the input stream to the database sink
operator
#RUNQUERY
Stream to dbSink select * from AliseSource;
```

DEDUPLICATENORMAL

```
///OdysseusScript
#PARSER PQL
#TRANSCFG Standard
///This operator checks the measureValue1 of each sensor with ids
unique id and filters out
///duplicate data or tuple.
#QUERY
NewNormalOutput:= changedetect({attr = ['measureValue1'],
deliverFirstElement = ['true'], group_by = ['id']}, OutputNormal)
```

PQLSINK

```
///OdysseusScript
#PARSER PQL
#TRANSCFG Standard
/// These operators create sinks for each data stream on a specific
port.
#RUNQUERY
OutputNormalSink = SENDER({sink= 'Sink', wrapper='GenericPush',
transport='TCPServer', protocol='SizeByteBuffer',dataHandler='Tuple',
options=[['host', 'localhost'],['port','12600']]}, NewNormalOutput)
#RUNQUERY
OutputGPSSink = SENDER({sink= 'Sink1', wrapper='GenericPush',
transport='TCPServer', protocol='SizeByteBuffer',dataHandler='Tuple',
options=[['host', 'localhost'],['port','12601']]}, OutputGPS)
#RUNQUERY
OutputTopoSink = SENDER({sink= 'Sink2', wrapper='GenericPush',
transport='TCPServer', protocol='SizeByteBuffer',dataHandler='Tuple',
options=[['host', 'localhost'],['port','12602']]}, OutputTopo)
#RUNQUERY
```

```
OutputTlightSink = SENDER({sink= 'Sink3', wrapper='GenericPush',
transport='TCPServer', protocol='SizeByteBuffer',dataHandler='Tuple',
options=[['host', 'localhost'],['port','12603']]}, OutputTlight)
```

29. Busstop GTFS File

The following code is an example of the busstop GTFS file:

```
stop_name,stop_id,stop_lat,stop_lon
Fuhrenweg,369886681,53.1350704,8.1458398
Waterender Weg,353044608,53.1547474,8.2447153
Mühlenhofsweg,659957027,53.1658137,8.2385449
Ewigkeit,348086700,53.115238,8.2128886
Westerstraße,677530887,53.1419016,8.1979981
Wahnbäkenweg,673234549,53.1939063,8.2434703
Heidkamp,264128488,53.1891912,8.1565111
Ekernstraße,671193399,53.1865208,8.2431326
Schwertlilienweg,354643355,53.0962443,8.2539436
```