University of Applied Sciences
Bonn-Rhein-Sieg
Department of Computer Science

Fraunhofer Institut
Intelligente Analyse- und
Informationssysteme

Master Thesis

in:
Master Autonomous Systems

"A Rich Client framework for the OGC sensor web"

Manuel Fabritius

First supervisor: ..... **Prof. Dr. Paul Ploeger**
Second supervisor: ..... **PD Dr. Michael Mock**
Delivered at: June 30, 2010

Acknowledgment

I, the undersigned below, declare that this work has not previously been submitted to this or any other university, and that unless otherwise stated, it is entirely my own work.

DATE

author

Contents

Nomenclature

AJAX	Asynchronous JavaScript and XML
BSD	Berkeley Software Distribution
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
DTD	Document Type Definition
ESS	Emergency Support System
FOI	Feature Of Interest
GIS	Geographic information system
GML	Geopraphy Markup Language
JSON	JavaScript Object Notation
O&M	Observation & Measurement
OGC	Open Geospatial Consortium
OSGeo	OpenSource Geospatial Foundation
RIA	Rich Internet Application
SOS	Sensor Observation Service
UI	User Interface
W3C	World Wide Web Consortium
WFS	Web Feature Service
WMS	Web Map Service
XML	Extensible Markup Language

1. Introduction

In the field of accessing and visualization mobile sensors and their recorded data, different approaches were realized. The *OGC*[1] *Sensor observation Service* supplies a standard to access these information, stored on servers. To be able to access these servers, an interface must be developed and implemented. Especially in the context of the ESS [2] project, a way to create client applications, that are capable to access the SOS[3] standard, would be of great interest. In addition, those clients must be able to display the received information in a proper way.

1.1. Motivation

The EU-project *Emergency Support System* is developing a security support system which will provide real time information to crisis managers about abnormal events. The architecture should ideally comply with OGC standards. Mobile sensors will offer emergency case related information to a central operator. The ESS should provide information collected from mobile sensors (e.g. mobile GPS-enabled sensor stations for toxic values) through a portal to an emergency operator. To process these data, a client framework must be developed, that allows to create interactive OGC-conform Web-enabled GIS[4] clients offering services for OGC sensor mapping, visualization and

[1] Open Geospatial Consortium
[2] Emergency Support System
[3] Sensor Observation Service
[4] Geographic information system

OGC-compliant query processing.

The framework should provide facilities for easy configuration of dedicated client applications, including a wide variety of visualization, spatial data selection and data query possibilities (for example, queries including historic data and queries based on data values). The sensor information are stored on an SOS server that will be developed by Roland Mueller. One challenge are mobile sensors. They measure different phenom-

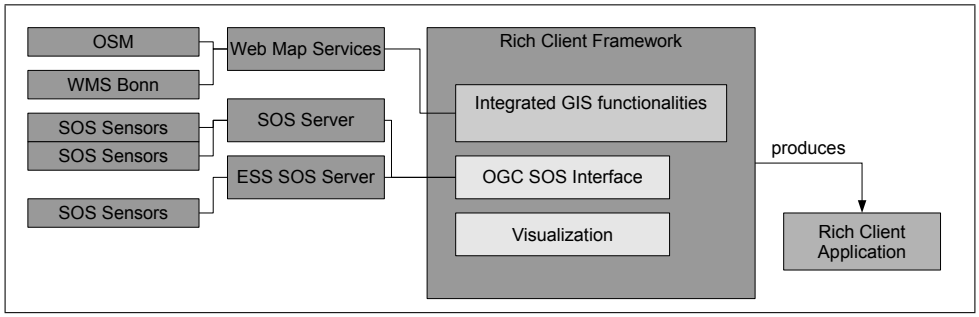

Figure 1.1.: Overview of a Rich client Framework interacting with OGC services

ena in a certain frequency at different positions. Using these positions, a track can be generated. But how to visualize a sensor circling in the same area ?

1.2. State of the art

There are existing approaches that are working with the OGC standard. On the one hand, these approaches which are using large frameworks like 52North do. The 52North Initiative[5] have developed a large Java based framework that is capable of accessing several OGC standards, including the SOS. Based on this framework, different client application and servers have been developed.

On the other hand there are also projects that uses a server to pre-process the OGC sensor information first in order not to overload the client. For example the *OOSTethys* project [6] developed a server that supports the OGC SOS standard. They are also using

[5] http://52north.org/

[6] http://www.oostethys.org/

a lightweight GoogleMaps client to visualize their stored information in a very basic way. The diploma work of Riegger [1] describes a way to access sensor information on a web-based way. But they are not using the OGC SOS standard like OOSTethys.

Alternatively there is also an approach where the web client is doing the whole work like OpenLayers[7]. It is a web-mapping library, that is used by the OpenStreetMap project to access OGC standards for their web-mapping purpose. The community developer (Bartvde) developed a lightweight SOS interface with JavaScript. It is capable of basic requests and can already map sensors. Another client-only based application, done by Dominik Helle [8] also uses OpenLayers. This *SensorGIS* is able to handle basic requests and can show the received information in tables and charts.

This preliminary work has been done visualizing individual OGC sensor values using the OpenLayers library, but building client applications with rich visualization, configuration and query selection facilities are not supported.

1.3. Objective

The result should be a configurable development framework for web-based GIS clients supporting the OGC sensor observation services. In particular the framework should allow continuous position updates of mobile sensors. Visualization features like charts, bounding boxes of sensors and data series should be included. One possible example is shown in figure 1.2 and 1.3. Mobile sensors with their produced tracks should be visualized in a way that every user is able to extract the important information.

[7] http://openlayers.org/
[8] http://maps.terrestris.de/sensorgis

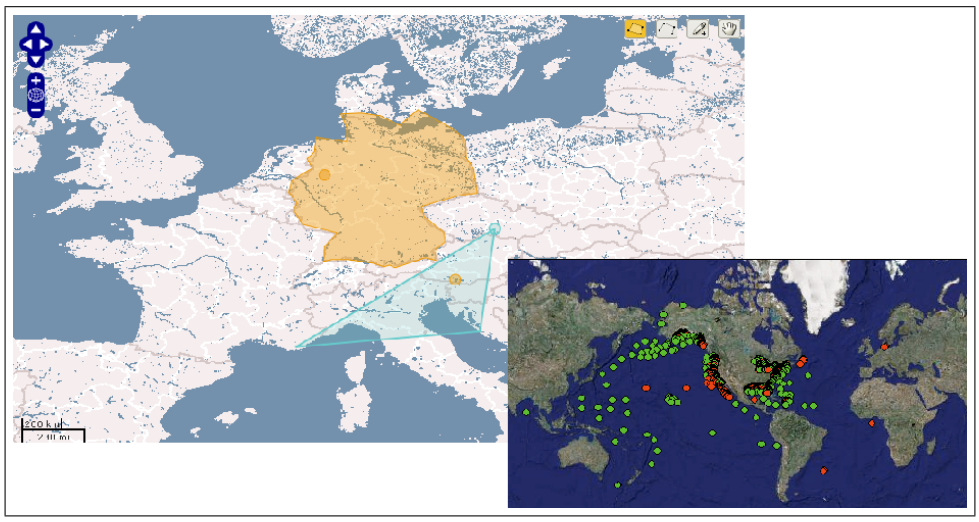

Figure 1.2.: Map visualization example for sensors

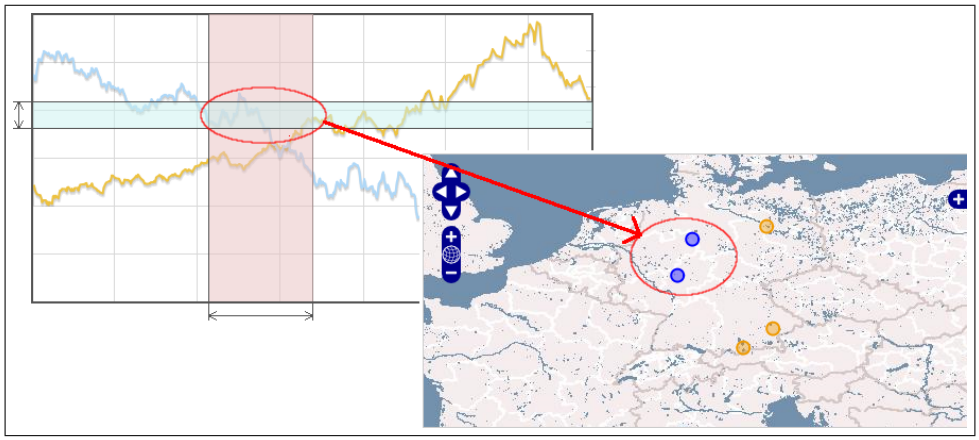

Figure 1.3.: Chart-Map interaction Example for Visualization

1.4. Structure

The following chapter of this work describes the background of the different techniques used during the thesis. Chapter 3 makes a short analysis of related work and the OGC standard itself. In addition, a requirement analysis is done. The concept chapter describes all used components and the developed architecture. The developed framework with its widgets and the SOS interface are described in detail in chapter 5. All results including an example client application are described in chapter 6. Limitations to the

implementation and according benchmarks are also discussed.

2. Background

In this chapter different technical and theoretical fields which are important for the thesis are being explained.

2.1. Rich Client

A Rich Client is a software where the specific user interface and functional logic is implemented on the client side. A Rich Client consists of a GUI with an integrated presentation interface and the functional logic. Some characteristics:

- native widgets
- drag-and-drop functionality
- universal component model

Common application functionalities are abstracted into a framework called *Rich-Client Platform*. On the other hand, the Thin Clients functionality is completely provided by the server. A Fat Client is more similar to the Rich Client, but it does not support the component model with common components.

2.1.1. Framework

A Rich Client Platform is the framework to develop software components (plug-ins) which are combinable like a construction kit. The plug-ins are centrally managed and enables applications to be dynamically extended.

The most famous Rich Client Platform (RCP) is probably the Eclipse RCP framework. There are several more RCP projects like Java Development Tools[1] (JDT), IBM Workplace[2] or Borland Together[3]. All these projects have some common characteristics.

plug-in An application consists of different independent components, which can be loaded and/or removed. Components that support hot-plugin can be removed and added during runtime.

adaptable for enduser Each user specifies his own set of plugins.

adaptable for different devices The framework already supports interfaces for different devices.

solution for different problems Sets of tools can be combined and offered to users on order to solve specific problems.

encapsulation of basic tasks Basic tasks are included to the framework to offer services for all plug-ins. Therefor, developing a framework that supplies solutions for basic tasks, generates some overhead.

2.1.2. Rich Internet Application

Traditional web-applications are created with the client-server architecture, where the client (web-browser) sends requests to a server. After the response arrived, the clients web-page could be reloaded. These synchronous procedures are slow and not very comfortable, because lots of redundant data is being transmitted. [2, p. 30]

With new technologies like JavaScript and AJAX[4] clients have the possibility to process program logic on its own and transmit data asynchronously. *Rich Internet Applica-*

[1] http://java.sun.com/
[2] http://www-01.ibm.com/software/lotus/
[3] http://www.borland.com/de/products/together/index.html
[4] Asynchronous JavaScript and XML

tions have the capability of *Rich* GUI elements and complex features and functionalities like desktop applications. But these RIA's can be executed in modern web browsers.

2.2. Geographic information system

Geographic information systems (GIS) are used to present, analyse, process and gather geographical (spatial) information of different kinds of maps. Data are the core of every GIS. A couple years ago, it was pretty difficult to gather those information. But with modern GPS this process is much faster and more precise.

GIS: "*Ein Geoinformationssystem dient der Erfassung, Speicherung, Analyse und Darstellung aller Daten, die einen Teil der Erdoberfläche und die darauf befindlichen technischen und anministrativen Einrichtungen sowie geowissendschaftliche, konomische und kologische Gegebenheiten beschreiben.*" [3, p. 12]

New GIS are helping to make political, military or economical(advertisement) decisions. Private persons also use GIS for routing.

Free and OpenSource projects, which deal with spatial information, are being supported by the *Open Source Geospatial Foundation* [5].

2.2.1. Desktop GIS

GIS, which must be installed on a client system, are called Desktop-GIS. After a short introduction, the user should be able to work with spatial data. Some only have a viewer to visualize requested data, others also have the capability to modify and store data. With the release of the new WebGIS, the common Desktop-GIS lost its role as viewer, because a WebGIS could be reached by every client with an Internet connection. Additionally, most Desktop-GIS need to be licenced. Newer Desktop-GIS also have interfaces to Web-Services to request data from servers. For example the *Web Map Service* (WMS)

[5] http://www.osgeo.org/

standard (explained in chapter 2.3.1) from the Open Geospatial Consortium. [4, p. 121]. Common DesktopGIS:

ArcGIS ArcGIS is a collection of different software-products from ESRI [6]. The different product families

- Desktop GIS
- Server GIS
- Online GIS
- ESRI Data
- Mobile GIS

have been developed for a variety of requirements which must be fulfilled. The products from one family only differ in the scope of operations. Some products are free to use, but the majority is under a commercial licence.

MapInfo A commercial software from MapInfo Corporation to collect, edit, analyse, visualize and present spatial data.

UDig User-friendly Desktop Internet GIS is a open-Source Java implemented viewer and editor for spatial data. It supports the OGC standards Web Map Service (WMS), Web Feature Service (WFS) and Style Layer Descriptor (SLD). It can directly be deployed or used to create a more specific application. The GNU Lesser General Public License[7] was used.

Quantum GIS is an Open-Source-GIS (GPL) that supports several data formats. The project was initiated in 2002, to make a GIS viewer accessible for common PC's. It was implemented in C++ and supports a Plug-In architecture that simplifies functionality extensions. It supports the most common OGC standards like WMS

[6] http://esri-germany.de/products/index.html
[7] http://www.gnu.de/documents/lgpl.de.html

and WFS[8]. It's also possible to access the spatial data directly from a PostgreSQL or PostGIS database.

2.2.2. Web Mapping

Web GIS or Web-Mapping allows to view GIS data in browsers. These GIS are platform-independent and in most cases no licences has to be bought (i.e. OpenLayers, GoogleMaps). As a result, a large number of users are able to access the information supplied by a WebGIS. Additionally, new technologies like JavaScript combined with fast client computers allow to process real GIS functionalities on the client. For instance, the users are able to switch between different themes or request specific information. The information is shown with overlays on the map. To distribute those JavaScript-Clients, an administrator has to set up a web-server where an HTML page provides access to the clients.

But this architecture needs a server component where all the information are stored, with constant access! For this purpose, the Open Geospatial Consortium defined several standards who deal with those information. Chapter 2.3 explains these standards.

Probably the most famous Web mapping services are:

GoogleMaps In the beginning of 2005, Google Inc. started the web service *Google Maps*[9] where users have access to satellite or map pictures from all over the world. To use the API of Google Maps, a free key has to be requested. Google has its own licence, the Google Terms of Service[10], which offers a non-commercial use only.

OpenLayers OpenLayers [5] was initially released on 26th June 2006. It is a pure JavaScript and AJAX [11] library to integrate maps into web-pages. Map services of different proprietary vendors can be used. Current OGC standards are also

[8] Web Feature Service
[9] http://maps.google.com/
[10] http://www.google.com/accounts/TOS
[11] Asynchronous JavaScript and XML

supported. The project is being distributed under the BSD[12] licence and supported by OSGeo[13]. [4, p. 271]

MapBender MapBender is a framework to manage spatial portals with services for digitalization and metadata management. These portals are hybrid applications using JavaScript with AJAX and a server component (PHP). The complete configuration is web-based, including user management with access restrictions. [14]

BING is a search engine of Microsoft. It also includes a map service similar to GoogleMaps. The *Bing Maps* service is an advancement of MSN Virtual Earth and allows access to spatial data and services. [15]

Those services are based on the OGC Web Map Service to access spatial data. In the next section, this consortium will be described in detail.

2.3. Open Geospatial Consortium

Since new technologies of Web-Mapping provide GIS to a large number of users, these systems became increasingly more important. The spatial data are no longer sored on local computers, instead it is provided by servers. To request the needed data, the servers must provide appropriate services which handle the spatial information with special standardized interfaces. Those interfaces have been defined by the Open Geospatial Consortium. [6, p. 26]

OGC *"The Open Geospatial Consortium, Inc (OGC) is an international industry consortium of 401 companies, government agencies and universities participating in*

[12] Berkeley Software Distribution
[13] OpenSource Geospatial Foundation
[14] http://www.mapbender.org/Main_Page
[15] http://www.bing.com/maps/

a consensus process to develop publicly available interface standards. OpenGIS Standards support interoperable solutions that "geo-enable" the Web, wireless and location-based services, and mainstream IT. The standards empower technology developers to make complex spatial information and services accessible and useful with all kinds of applications. OpenGIS is a Registered trademark of the Open Geospatial Consortium, Inc (OGC) and is the brand name associated with the Standards and documents produced by the Open Geospatial Consortium, Inc (OGC). OpenGIS standards are developed in a unique consensus process supported by OGC industry, government and academic members to enable geoprocessing technologies to interoperate, or "plug and play". You will also find the OpenGIS trademark associated with products that implement or comply to our standards. Make sure that your geoprocessing and location services procurement and technology development programs demand OpenGIS standards!" [7]

The following sections describe WMS and WFS, the most interesting services for Web-Mapping offered by OGC. Additionally the *Sensor Observation Service* will be described.

2.3.1. Web Map Service

The WMS is a standardized service that can be used to request dynamically generated maps. The communication between client and server is defined by the WMS. Four different requests are possible:

GetCapabilities Returns a description of the requested service.

GetMap Returns the requested map.

GetFeatureInfo (optional) Returns the information of the objects in a requested region.

GetLegendGraphic (optional) Returns the legend of the used map.

With MapServer[16], an OpenSource implementation that supports several OGC standards (including WMS), it is possible to request dynamically generated maps. The MapServer can also access other services which offer geo-information. The request needs several further parameters like position, size, projection and the layers. Based on this information a picture of the area is being sent back to the client. [8] Figure 2.1 shows

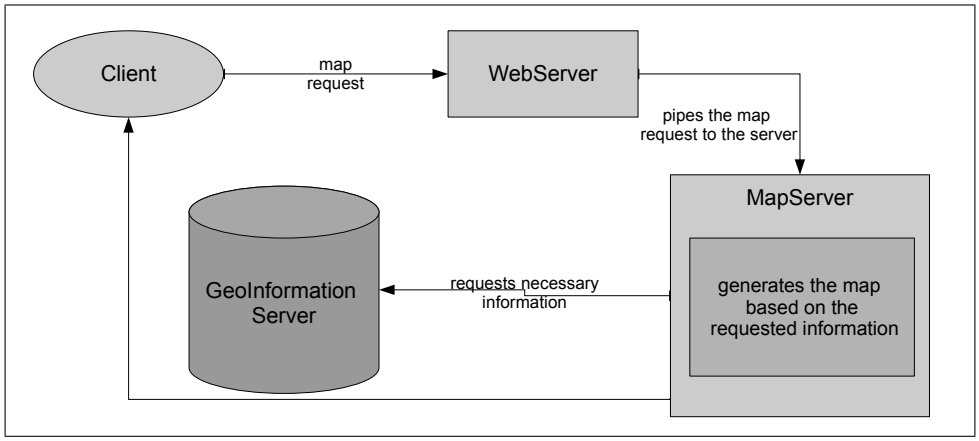

Figure 2.1.: MapServer communication

the communication.

2.3.2. Web Feature Service

The WFS is another OGC standard. It is used to access spatial and non-spatial information instead of pictures as the WMS does. There are two different possible implementations.

Basic-WFS It defines read-only operations:

GetCapabilities Returns meta-information of the WFS

DescribeFeatureType Describes the feature type structure

GetFeature Returns a feature

[16] http://mapserver.org/

Transactional WFS It describes read and write operations:

Basic operations Support for the basic WFS operations.

Transaction Write operations (create, update, delete)

LockFeature, GetFeatureWithLock

The WFS returns the requested information in XML documents. This information (features) must be handled by the client. OGC conform clients, who support WFS, are able to parse the XML-format GML[17]. This *raw data* needs more processing on the client side, but it allows the client to handle the information more flexible. [9] Fields of applications are:

- Search for objects
- Zooming and highlighting
- Analysis and post-processing operations
- Direct data manipulation on the server

2.3.3. Sensor Observation Service

The *Sensor Observation Service* is similar to the WFS. It defines an API, in which real-time sensor information are stored and interchanged in a standardized format. It is possible to request descriptions of the sensors or directly request the observations. In this context, the *O&M*[18] describes the requested observations and the *SensorML* describes sensors. It is also possible to send insert requests to the server. The different operations are separated in three different profiles:

Core profile: The basic operations which should be supported by all servers/clients. [10, p. 20-33] A more specific analysis of these operations can be found in chapter 3.2.

[17] Geopraphy Markup Language
[18] Observation & Measurement

GetCapabilities Returns a description of the service, including offered operations, sensors, sensor-measurements(time-range and phenomena)

GetObservation Returns the requested observations. There are different time and spatial filters available to get a more specific result.

DescribeSensor Returns meta information about one specific sensor. Typical information are sensorID (and other sensor names), sensor position and observed phenomena (temperature, humidity...).

Transactional profile: Operations for sensor/server communication. [10, p. 33-37]

RegisterSensor Registers a new sensor on a running SOS.

InsertObservation Inserts measurements to a registered sensor.

Enhanced Profile: Optional client/server operations. [10, p. 38-]

GetObservationById Returns an observation based on an identifier.

GetResult Repeatedly obtains sensor data from the same set of sensors without sending and receiving similar requests and responses.

GetFeatureOfInterest Returns a featureOfInterest, what is defined in the SOS capabilities document.

GetFeatureOfInterestTime Returns the time periods for measurements of a feature of interest.

DescribeFeatureType Returns the XML schema for the specified GML feature.

DescribeObservationType Returns the XML schema describing the observation type that is returned for a particular phenomenon.

DescribeResultModel Returns the schema that will be used, when the client requests the result model by the ResultName.

The SOS standard also introduces several additional definitions, which are used in the different XML documents:

Feature of interest The *FOI*[19] could be any geographic object, what should be observed. It depends on the project, what a feature is and which properties are being observed.

Observation Sensors are producing observations. One observation is one result of one event of a *FOI*.

Offering It is a logic grouping of observations.

Phenomena Observed physical properties of a geo-object are phenomenas (temperature, toxic...).

Procedure One procedure generates observations. In most cases it's a sensor.

[10] To access *Sensor Observation Services*, an interface must be implemented during this work.

2.4. AJAX

Asynchronous JavaScript and XML applications are running in web browsers. These AJAX[20] pages are able to communicate with the server after being loaded into the browser. Depending on the user commands and the page itself, only parts of the page will change. Compared to the classic HTML model, where each user interaction request a full page, the AJAX model needs much less overhead. [11, p. 2] Figure 2.2 on page 23 illustrates the two models. The fundamental concept behind AJAX is the separation of user interaction and network interaction. Several important aspect can be derived from the picture:

- The AJAX layer can access the network or only change the interface (i.e to verify a form).

[19] Feature Of Interest
[20] Asynchronous JavaScript and XML

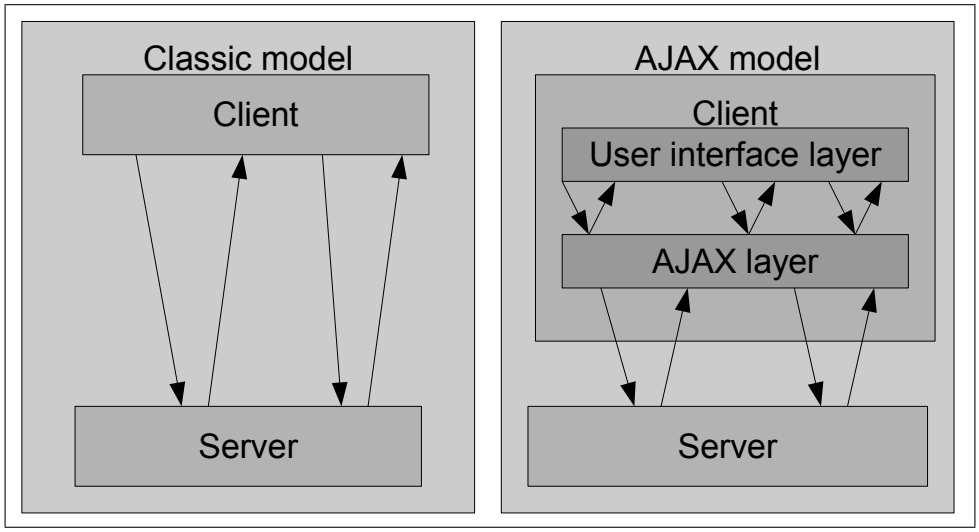

Figure 2.2.: AJAX comparison to classic HTML. Taken from [11, p. 3]

- Communication with the server mostly fetches small parts of a page. As a result, the database access on the server can more often use the cash and thus, the responses need less time. The pages will be loaded faster.
- The UI[21] layer is independent from the network communication. Server access will not block user interactions.
- Some server communications do not affect the UI layer. For example preloading of elements, or informing the server of the users action.

The more AJAX is used to build a web page, the more it acts like a desktop application - with one exception. For initialisation, there must be a server connection.

2.4.1. Asynchronous

As the name already says, the concept of AJAX is the asynchronous communication, which means: After a request was sent to a server, the client can proceed with other operations. The operation is non-blocking. If the server sends the response, the client will

[21] User Interface

handle the received information. But in fact, the communication can also be switched to synchronous. In some critical situations this might be useful. For example, if a form was sent with AJAX and the user can only proceed with the response.

2.4.2. JavaScript

Once named *LiveScript*, now everyone knows it as JavaScript, but the official independent name is *ECMAScript*.[22]. It is the most used language for client side scripting and of course the most important language for AJAX. JavaScript is a classless, object-oriented prototype, based programming language. Beside AJAX, there are several additional fields for applications:

- Validation of forms
- Frame manipulation
- Dynamic manipulation of DOM
- CSS[23] manipulation without reloading

To create web applications with JavaScript, there already exist lots of different frameworks and libraries, which help to develop browser compatible applications. Some of these frameworks generate a completely new abstraction level. This changed the programming style from the scratch. The base for object-oriented programming, comfortable DOM manipulations and graphic animations was born. Since the release of AJAX in 2004, JavaScript reached a new level. It's possible to create rich client applications, which act and look like desktop programs but are still being executed in browsers. [12, p. 7] One of the most important and very powerful feature is

Closures are function definitions inside other functions. The interesting part is the accessibility of variables of the outer function from the inner function. This is

[22] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[23] Cascading Style Sheets

very important, if the inner function was returned as a callback. On that way, it's possible the generate object-oriented properties like inheritance and private/public structures, which is not native supported by JavaScript.

Other points, that make JavaScript so very dynamic: IT is weakly typed language, the objects are dynamic and has a strong literal object-notation. On the other hand, one really ugly property is the global scope of those variables, which are not being defined inside a function.

JSON

The lightweight data-interchange format JSON [24] is easy to read and simultaneously easy to parse and to generate.

```
1  //Common Notation
2  var StringVariable = "string";
3  var NumberVariable = 42;
4  var ObjectVariable = {};
5
6  //JSON Notation
7  var ObjectVariable = {
8        NumberVariable : 13,
9        StringVariable : "string",
10
11       anotherObject : {
12          morestuff : "..."
13             }
14       };
```

Listing 2.1: JSON comparison to common notation

Listing 2.1 shows how short JSON is in comparison to the common notation. Additionally, the notation is arbitrarily expandable. JSON is supported by lots of languages including:

- C, C++, C#, Java, JavaScript, Perl, Python and more

In the case of the prototype based JavaScript, objects and functions can also be defined with the JSON notation. This can result in very complex definitions and statements.

[24] JavaScript Object Notation

Combined with frameworks like JQuery and/or Prototype, the syntax reaches a complexity that is barely readable. [13]

2.4.3. XML

The Extensible Markup Language, another element used by AJAX, is a markup language to present hierarchical arranged data. It is used to exchange information between computers in a platform- and implementation -independent way. XML[25] specifications are released by the W3C[26]. They limit the structural and content possibilities, what results in different application-specific meta languages. The limitations are defined in schema-languages like XML-Schema or DTD[27]. Important terms:

Well-formed Several rules must be kept, to generate well-formed XML.

- There is only one root-element inside the XML document.
- All elements with content must consist of one beginning and one ending tag ($< element > content < /element >$). If the element is empty, it can also be closed with the first tag ($< element/ >$)
- Every begin and end tag must be closed on the same hierarchical level.
- More than one element with the same named is not allowed inside an element.

Valid An XML document is valid, when it is well-formed, the schema definitions are linked and the schema rules were correctly used.

Parser Parser are programs that can read and interpret XML. It is a validation-parser, if it checks the schema-rules.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <rootNode>
3    <firstElement>
```

[25] Extensible Markup Language
[26] World Wide Web Consortium
[27] Document Type Definition

```
4       <name>exampleName</name>
5     </firstElement>
6     <secondElement></secondElement>
7 </rootNode>
```

Listing 2.2: XML example

In listing 2.2, a short well-formed example is shown.

3. Evaluation

Before getting started with designing the framework and its interface to the SOS, different areas, which are covered, need a closer look. Different tools, which needed to be used, must be analysed to get an overview of their characteristics. The OGC SOS standard itself will be analysed to discover weak spots. Additionally, a requirement analyse will name important intentions.

3.1. Related work

Related work including common tools, which are being considered to be used for the framework, must be evaluated. Of course, for visualization of sensors on a map, a Geo-Information-System or a Web-Mapping library must be used. Since the framework should be web-enabled, only Web-GIS are being considered. In this connection, there are already libraries available, that support the OGC SOS standard. It could be wise to connect to an OpenSource library, to reduce the amount of work. Finally, the essential tool for web enabled rich client applications, JavaScript will be covered.

3.1.1. Web-GIS

As already described in section 2.2.2, there are different web-mapping libraries available which offer very basic GIS functionalities (browsing through maps). And there are of course Web-GIS available which are using one of these web-mapping libraries. Some examples are:

- OpenStreetMap uses OpenLayers. [1]
- GPSies uses GoogleMaps API [2]
- Geoportal.RLP (Rheinland Pfalz) uses MapBender. [3]

The GoogleMaps API is very powerful and is being used in a lot of projects. But in order to use it, a *GoogleMaps key* must be created with a Google account.[4] This means, for each copy of the framework, a new Google key must be generated. On that way, one is more or less dependent on Google itself. Since the Fraunhofer institution wants to be independent, the GoogleMaps API will not be considered.

MapBender

MapBender is a hybrid client framework. Beside AJAX, a CGI[5] (PHP) enables the framework to create administrative components with user restrictions. It's completely web-based configurable. Figure 3.1 shows one possible layout. All information about the user interface are dynamically read from a database during runtime. The maps are read from OGC services.

But the usability do not reach the standard of other Web-Mapping clients. It is not possible to zoom with common scroll keys (mouse wheel) and map-sliding is not working as it is known from GoogleMaps or OpenLayers. The whole map interaction is not very comfortable.

OpenLayers

Figure 3.2 shows the basic layout of an OpenLayers map. This AJAX-only library supports to include different kinds of maps. The common OGC standards are supported,

[1] http://www.openstreetmap.de/
[2] http://www.gpsies.com
[3] http://www.geoportal.rlp.de/portal/karten.html
[4] http://code.google.com/intl/de-DE/apis/maps/
[5] Common Gateway Interface

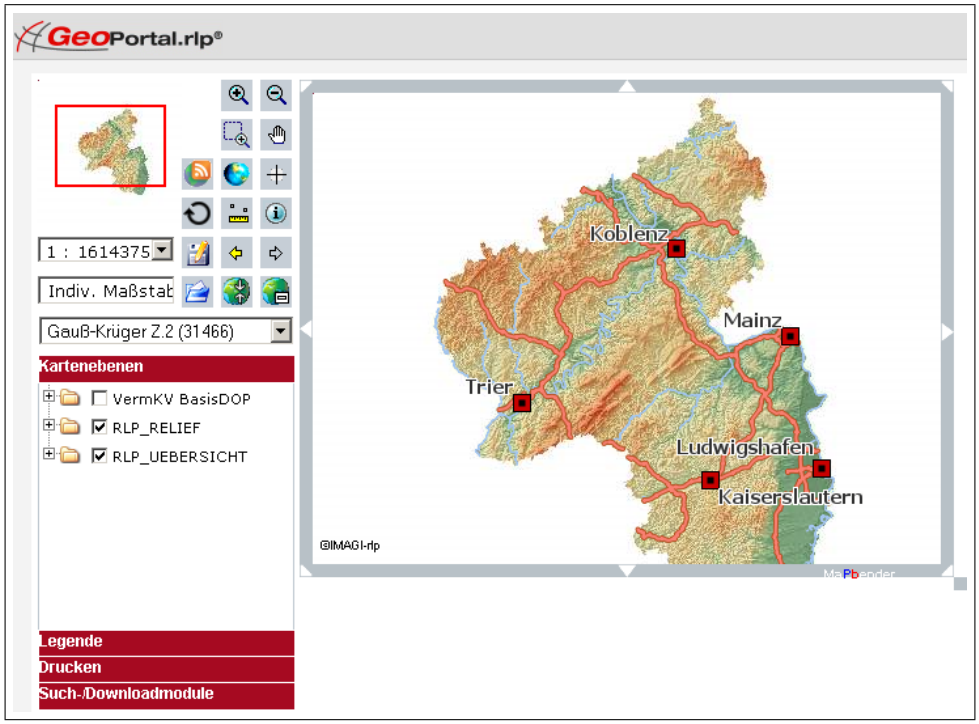

Figure 3.1.: MapBender Example layout, taken from http://www.geoportal.rlp.de/portal/karten.html

as the most proprietary providers. The map interaction like zooming and scrolling behaviour is similar to GoogleMaps. The documentation of this OpenSource project is very extensive and the community is also very active. The project-page offers lots of examples to ease the introduction. On top of that, there is already a sandbox example of an OGC SOS implementation. But more about this in the next section.

3.1.2. SOS implementations

In 2007 the Sensor Observation Service got an official OGC standard. Since this time, several applications and frameworks implemented the service. There are a number of different approaches that will be named in this section. Of course it could be reasonable to use and extend one of these implementations for the ESS SOS interface. The following applications support the SOS standard on one or the other way.

Figure 3.2.: OpenLayers Example layout, taken from http://dev.openlayers.org/releases/OpenLayers-2.9.1/examples/example.html

OX-Framework This large JAVA framework, developed by 52NORTH [6], supports several OGC standards, including the SOS. Different clients are available, where the framework is used as an interface.

- Rich OX-Client
- Thin SWE client
- ArcGIS SOS Extension
- uDig Plugin
- OX Server Applications

Basically, all these versions are using the OX-framework core, implemented in Java. [14]

OOSTethys The OOSTethys offers different implementations(Java, Perl and Python) of the SOS standard. They use GoogleMaps to display the sensors, but the SOS-based computation (request, response, parsing...) is done on the server-side. The server also holds all addresses of the available services. The client receives

[6] http://52north.org/maven/project-sites/swe/clients/index.html

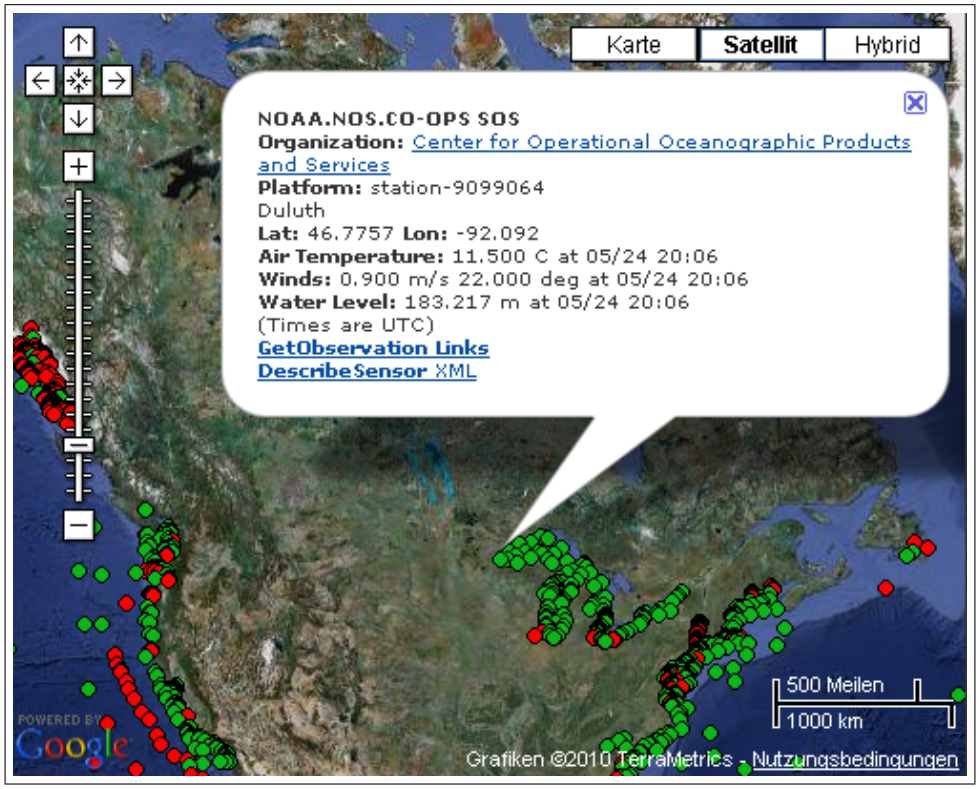

Figure 3.3.: OOSTethys Example, taken from [15]

the sensor-information in XML and has still some computation to do. It is not possible to access a Sensor Observation Service that is not registered on the server. On the other hand, the server can exactly control and log, which service has been requested how often. Figure 3.3 shows an example. Anyway, it would produce some overhead to split a framework in two or more languages, because AJAX has to be used for the visualization. [15]

MapServer It's a server implementation, without any client functionalities and therefor unsuitable for an client framework. [16]

deegree Supplies different Java based solutions (server and clients)for OGC services. But their web client *iGeoPortal* does not support the SOS yet. [17]

Dominik Helle's Diploma work It offers facilities to request a SOS server and is able to

map sensors a OpenLayers map. The received observations can be shown in charts and tables. [18] But the sources and documentation are not available. Anyway, the approach seems to be hard-wired with the infrastructure of *Terrestris*.

OpenLayers SOS The OpenLayers developer *Bartvde* extended OpenLayers with an interface to the *Sensor Observation Service*. At the time of writing, it's still in development. The interface supports three basic requests:

- GetCapabilities
- GetFeatureOfIntrest
- GetObservation

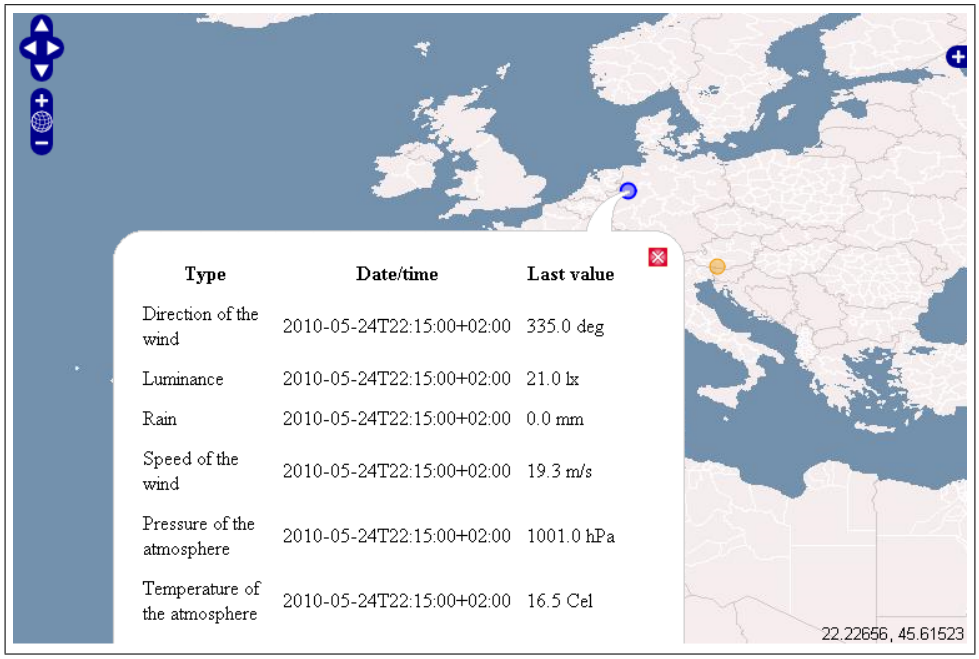

Figure 3.4.: OpenLayers SOS Example layout, taken from http://openlayers.org/dev/examples/sos.html

With this interface, it is possible to access SOS's and display the sensors on the map, if the SOS supports the *GetFeatureOfInterest* request. The GetObservation request is not really OGC conform and does only work with the *UniMuenster SOS*,

because it requests the "latest" value. In my opinion, that filter is missing in the SOS specification. Figure 3.4 shows the available example from Bartvde.[7]

This list does not claim to be complete. The OX-Framework from 52North is very powerful, but with its Java implementation it's more suitable for *Thin Clients*.

But the AJAX technology combined with fast client computers is applicable for *Rich Clients*, because it allows real GIS functionalities, processed only by the client itself. That's also the reason, why an OpenLayers environment is most interesting. The SOS example of Bartvde directly connects to OpenLayers, furthermore it is the only one that is well documented.

3.1.3. JavaScript

Today, new *Rich Internet Applications* are mostly developed with JavaScript/AJAX. During the last years the programming style of JavaScript has changed several times. To develop flexible applications, it is essential to be up-to-date with actual frameworks and newest programming techniques. Without this knowledge, actual JavaScropt source code is barely readable.

One good source to get newest updates is the BLOG of John Resig[8], the JQuery creator.

Cross-Site Scripting

The security restriction of *Cross-Site Scripting* must be avoided. It is not allowed to send requests to more than one domain. This prevents JavaScript being misused by criminals. The standard workaround is a proxy, that is being used to handle all requests. The proxy receives the requests and pipes them to the real destination. On this way, it is possible to receive data from more than one WMS or SOS.

[7] http://openlayers.org/dev/examples/sos.html
[8] http://ejohn.org/blog/

Timeout

New client computers are able to process real GIS functionalities. But by using JavaScript, the computational time of an algorithm is limited. For example the Firefox browser stops a script, if a loop runs longer than five seconds (In the firefox config **dom.max_script_run_time** the value can be changed). The timeout value might change, if other browsers are being used. Additionally, the time needed to compute a script, depends on the computer where the client browser is running. But this limit is the largest problem if complex operations have to be performed. Especially in the context of an *Emergency Support System*, where possibly time-critical actions have to be performed, an AJAX based solution might not be the best choice.

A possible workaround are *web workers*, a new JavaScipt technique for threads. Only some browsers do support them yet. [9]. With their help, it is possible to create new JavaScript threads. Today, computer mostly have multi-core CPUs, what increases the advantage. The problems can be split in small peaces and the user interface is not being blocked. But the web worker are really new and only the newest browser support them.

Frameworks

To enhance the usability of JavaScript and to make programming with it more comfortable, lots of free and commercial frameworks are available. Their purpose are different. Some aim to ease DOM[10] operations and/or create nice animations, others create a new abstraction level that supports classic object-oriented programming style. The following (certainly incomplete) list characterizes some of the frameworks:

JQuery The OpenSource framework[11] supplies a variety of functions:

- DOM selection/manipulation

[9] https://developer.mozilla.org/en/Using_web_workers
[10] Document Object Model
[11] http://jquery.com/

- Animations and effects
- Extended event-system
- Several help-functions and AJAX functionalities

Additionally it is easily extendable. As a result, tons of plugins are available. The JQueryUI extension[12] for example offers lots of user interface tools, including window operations.

Prototype It is also an OpenSource framework.[13]. Core functions are for example shortcuts and AJAX methods. The probably most interesting feature is the support for classic object-oriented programming style. With its help its possible to use the well known design-pattern. Original JavaScript would cause problems with object-oriented based design patterns.

Rico Another OpenSource library[14] for developing *Rich Internet Applications*. It provides different animation and styling effects including drag&drop. Another interesting feature is LiveGrid. It can display information in tables. The behaviour is similar to Excel. Of course, like every JavaScript library, it also supports extra AJAX functions.

Ext JS It is a commercial framework for developing web-based GUI applications. [15] It supplies a wide range of GUI elements.

GeoExt It is an OpenSource toolkit to create Rich Web Mapping Applications based on JavaScript. The JavaScript frameworks OpenLayers and Ext JS are included in GeoExt. The web-mapping framework MapFish is one example, which uses GeoExt.

[12] http://jqueryui.com/
[13] http://www.prototypejs.org/
[14] http://openrico.org/
[15] http://www.extjs.com/

Flex-I-Geo-Web It is a project of the Fraunhofer Institution to supply a Web-GIS platform, where different project are able to interact. It uses OpenLayers and offers a window manager.

OpenLayers itself is using the frameworks Rico and Prototype. The GeoExt toolkit is very interesting to connect to, but to distribute the ESS framework on a known platform, the Flex-I-Geo-Web is being used.

3.2. OGC Standard

A closer look into the OGC SOS 1.0 standard reveals that there are several issues which can lead to complications during the conception and implementation of the interface. Figure 3.5 shows the basic communication from the client, after a OGC sensor server

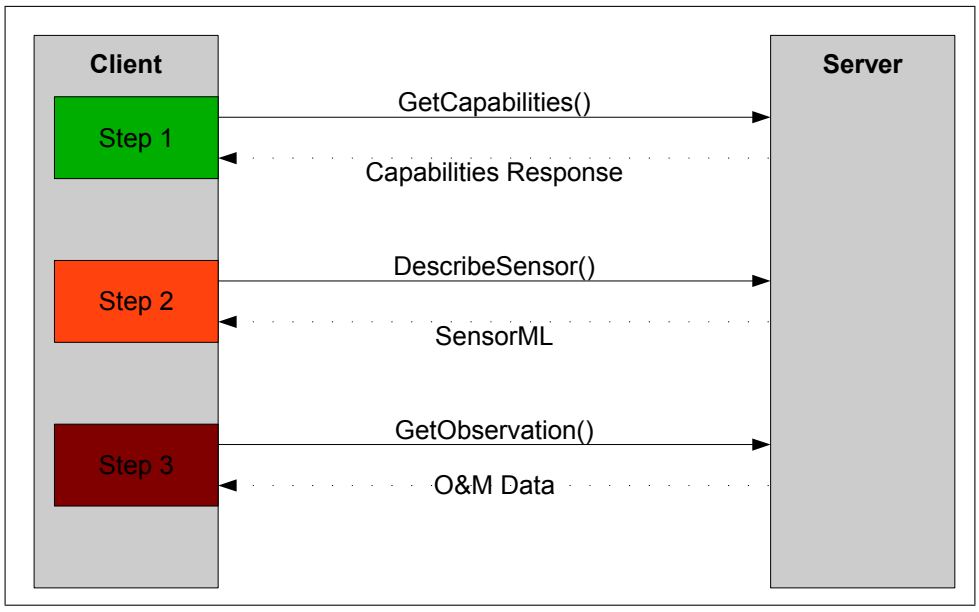

Figure 3.5.: SOS Client Communication

was discovered. But depending on the SOS's logic structure of the contents section in the capabilities document, the communication might be different. For example, multiple *DescribeSensor* requests might be needed to create a suitable *GetObservation* request.

3.2.1. Capabilities

The *Capabilities Document*, the response on a *GetCapabilities* request, provides access to the SOS metadata. The document is basically portioned in five sections:

OperationsMetadata Specifies which operations this SOS supports. That includes the request-method(get/post), metainformation (what parameters ...) and in what format the service can answer requests(xml/zip).

ServiceIdentification Supplies common information about the SOS. (title, keywords, abstract)

ServiceProvider Supplies some information about the provide of this SOS. (name, url, contact ...)

Filter_Capabilities Specifies what filter are being supported(spatial, temporal ...). Filters are used to specify *getObservation* requests.

Contents Basically it contains a list of *ObservationOfferings*. Among other things, each offering contains:

procedures One or more sensor identifier

featureOfInterest One or more locations with a spatial reference

observedProperty What properties (e.g. temperature...) are being observed by the sensors listed in this offering.

The offering can be ordered in any logic way.

With the *Capabilities Document*, all necessary information are given to request observations. But based on the logic structure or the offerings, it might be necessary to send more than one request, to get all observed properties of one sensor, because the *getObservation* request must contain only one offering. Depending on the application scenario, this fact should be hidden by the interface.

3.2.2. Sensor ML

The answer of a *DescribeSensor* request will be returned in the XML-language *Sensor ML* [19]. This description includes meta information like different sensor identifier (longname, shortname ...), the sensor position and of course what phenomena the sensor observes. There is at least one problem that can occur, if the sensor description is processed by a client. The *SensorML* allows to specify the sensors position in two different ways:

Position " *The location and orientation of an object relative to a coordinate system. For body-based systems (in lieu of point-based systems) is typically expressed by relating the objects local coordinate system to an external reference coordinate system. This definition is in contrast to some definitions (e.g. ISO 19107) which equate position to location.*" [19, p. 21]

Location "*A point or extent in space relative to a coordinate system. For point-based systems, this is typical expressed as a set of n-dimensional coordinates within the coordinate system. For bodies, this is typically expressed by relating the translation of the origin of an objects local coordinate system with respect to the origin of an external reference coordinate system.* " [19, p. 20]

As a result, in both cases the standard allows a variety of methods to define the actual position (e.g. lat/lon or northing/easting). Of course, it is very flexible and allows to define very different sensors, like moving satellites or stationary sensors. But the client, who needs to understand these definitions must either support the whole schema definition or be very specialized. The design of the interface should be able to handle this issue.

3.2.3. O&M

The answer of a *getObservation* request will be answered in the XML-language *Observation & Measurement*. This language includes different formats, how an answer can be sent. There are two methods to structure the data(observations):

dataArray Basically all returned values are composed into one large string, separated with defined symbols(decimalSeparator, tokenSeparator, blockSeparator). The string consists of several blocks. Each block contains all requested measurements(tokens) of one time-position. The descriptions of the tokes and meta-information, are separated from the real data.

dataRecords For each returned value, an own member (XML-tree) is being created. The tree consists of the major available meta information about the measurement and its sensor (sensor name, samplingTime, observedProperties, position, offering, unit...). This version is better human-readable, but it produces much more XML overhead.

These two models are completely different. The *dataArray* is obviously the more interesting, if you are working with limited resources like AJAX in a browser. On the other hand, it also depends on the application scenario, what methods fits better.

But this leads to the problem where services, that are using the same API (OGC SOS), are not able to communicate, because they are using different dialects.

Certainly the best way would be to support the complete XML schema. But after parsing on the client side, the different formats must also be handled on different ways. If an interface returns a different format for each server, the further processors must understand all these formats, or there must be an adapter that refactors the data in one consistent format. This leads to the question, why there are different *data models* available, if it only leads to new problems.

3.3. Requirement analysis

The practical part of this Thesis will be the implementation of a Rich Client framework for the OGC sensor web. It will help to create web-mapping applications which can access the Sensor Observation Services. In addition, the received sensor information will be visualized on the map, with the requirement to support mobile sensors. The mobile sensors will log their position and other observations. These information can be combined to manipulate the visual appearance of sensor's tracks. With respect to the *Emergency Support System* this should help the user to make appropriate decisions to actual situations. Of course there won't be a validation of the improvement, because no empirical measurements can be arranged.

The basic requirement is a framework for *Rich Internet Applications*, that is able to access the OGC sensor web. Since there is no real AJAX SOS interface, this would be the first part. The framework itself with its visualization widgets is the second part.

3.3.1. SOS Interface

A Sensor Observation Service compatible client allows to access sensor information that are stored on servers. The following criterias can be deduced from OGC's API definitions (analyse in previous chapter 3.2) and the requirements for the framework:

- Support the core Profile
- It must work with Roland Muellers SOS server.
- Support for time and spatial filter
- The SOS allows to structure its data on different ways. As described in section 3.2.1, logic structure behind an *offering* may differ. To give the framework the possibility to request data based on explicit senors, the SOSInterface should hide this complexity.

- Recognize if servers operations are not supported (soft-requirement)

3.3.2. Rich Client Framework

The following list describe requirements to the framework from the view of a user(programmer):

Operative Functionalities offered to the user (programmer).

- OGC compliant query processing
- The applications should run in a browser (Web-enabled)
- Sensors and other information must be visualized on a map
- Support for chart visualization of sensor data including:
  - Range filter for the shown data series
  - Based on the range filter, only show or highlight the corresponding sensors or sub-tracks on the map. Figure 1.3 on page 10 could give a rough idea.
  - Interactive map updates, if the range filter changes.
- Automatic updates for sensor-information. This also includes mobile sensors, changing their position.
- There should be example widgets for programmers

Passive Transparent characteristics.

- The framework must be expendable. New widgets should be integrated easily.
- The API should not be to complex. Usability is important.
- With respect to the *Emergency support system*, the framework should be robust and with good performance.

Using these requirements, a concept for the SOS interface and the framework is being developed in the next chapter.

4. Concept

After getting an overview of existing tools, a concept must be developed for the framework's architecture and the SOS interface.

4.1. Components

A number of frameworks, libraries and other components have been analysed in chapter 3. The following components have most value for this project.

4.1.1. Flex-I-Geo-Web

FlexiGeoWeb is a Fraunhofer Web-GIS project, that offers a platform where JavaScript applications from different projects are able to interact.

OpenLayers is used as web-mapping service. Each project that use the *Flex-I-Geo-Web* framework is able to manipulate the OpenLayers map and can communicate through listeners with other projects. Projects included into the framework are called *components*. Flex-I-Geo-Web uses a window manager of its own, where every component can request a new window to show its user interface elements in. Three different component types are possible:

Hybrid A project offers a HTML page. The body of the HTML page will be shown inside a window. JavaScript from the HTML head, will also be included to Flex-I-Geo-Web, and therefore be available. Additionally an appropriate JavaScript file

that must offer defined functions, will be executed by Flex-I-Geo-Web. On that way, the component get access to the OpenLayers map.

IFrame HTML A project offers a normal HTML page like a Hybrid component does. But this one will be shown inside an IFrame. The components HTML page must offer a defined JavaScript function. This function will be executed by Flex-I-Geo-Web to distribute necessary objects.

Headless A component without an user interface. Only a JavaScript file will be included to the header. This component type can be used for libraries.

Furthermore its possible to create an application with existing components, only via the user interface. The state can be stored and executed again at any time. The ESS Rich client framework will be included as a component to the *Flex-I-Geo-Web* project. On this way, it can directly be distributed and reused.

4.1.2. OpenLayers

MapBender and OpenLayers are the largest community projects for web mapping. But MapBender produces more overhead [4, p. 282] and map interactions do not behave like modern GIS do. Further more a SOS example for OpenLayers has already been created, as described in section 3.1.2. It will be used as basis for the *ESS SOS Interface*.

4.1.3. JQuery

JQuery is one of the most used JavaScript libraries [20, p.114]. It supplies nice animations, DOM manipulation, great documentation and, probably most interesting, a lot of plugins. Since OpenLayers is already using Rico and Prototype, their advantages will also be available. To visualize sensor data, a certain plugin of JQuery, *flot* will be used [21]. It adds interactive charts to the functionalities of JQuery. It can draw several

time series in one graph and is completely interactive. One explicit example [1], where it is possible to mark a range inside the chart, could be used for partially visualization of GPS tracks. One requirement in section 3.3.2 already explained such a feature.

4.2. Architecture

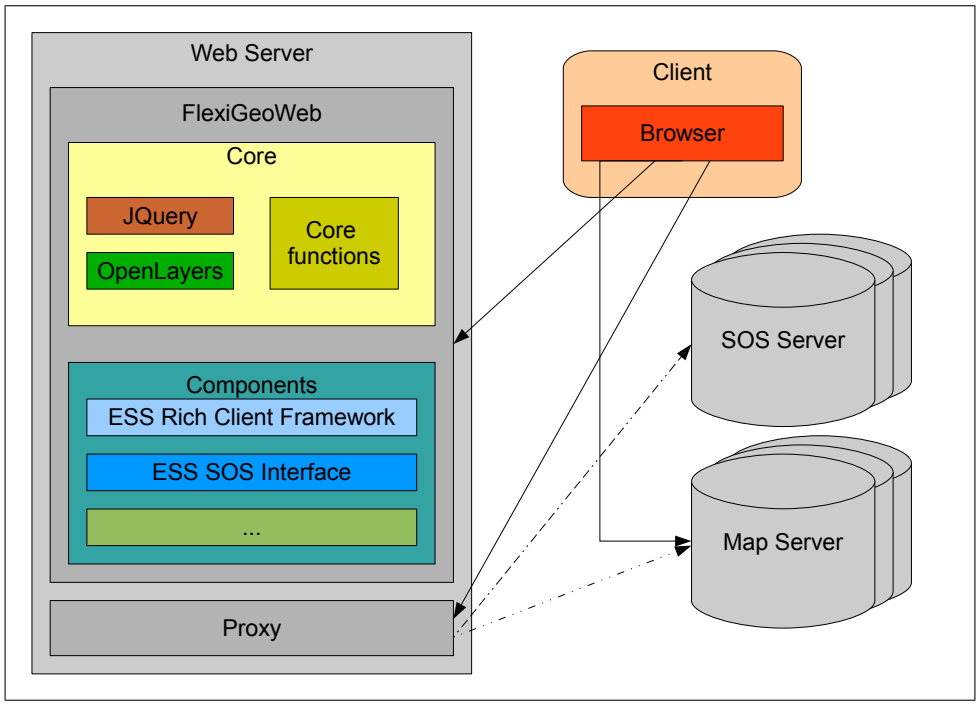

Figure 4.1.: Complete architecture

A web-mapping application has its own base-architecture, that first has to be build. Basically it needs [4]:

Webserver The web-server (i.e. Apache HTTP server) supplies the webpage to end-dusers. The page contains the map and tools for the users and sometimes offers a proxy to enable the client to access more than one domain (Cross-Site Scripting see chapter 3.1.3).

[1] http://people.iola.dk/olau/flot/examples/selection.html

Mapserver A server that supplies the needed geographic information (pictures). Actually a lot of free MapServer are available, which already offer lots of different maps.

Client A Web-Mapping-Client or a library like OpenLayers or MapBender must be used as a WMS interface. For this project, OpenLayers will be used.

Internetconnection To supply the GIS to users, a fast Internet connection is necessary.

Based on this architecture, the SOS interface and the *ESS Rich Client Framework* will be developed. A complete overview is shown in figure 4.1. The explicit architecture of the *ESS SOS Interface* and framework will be introduced in next sections.

4.2.1. OGC SOS Interface

As already explained, the OpenLayers library with its SOS interface example will be used. Bartvde did a great job, integrating the SensorML and O&M schema-parsing to OpenLayers. It allows basic OGC-compliant query processing. But a short analysis in

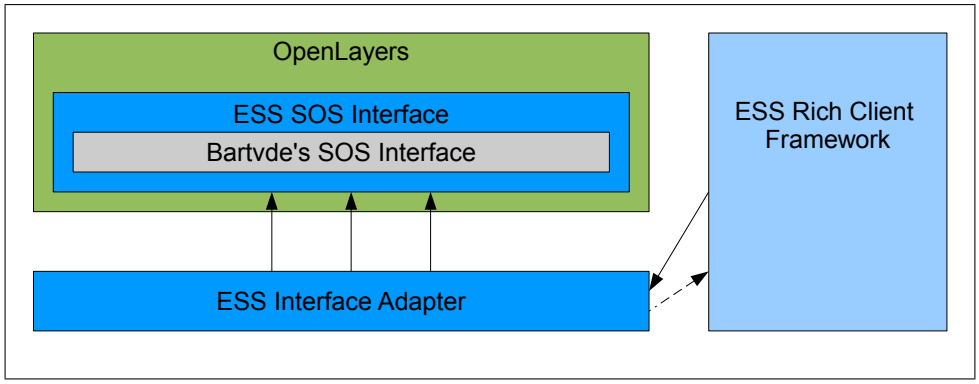

Figure 4.2.: SOS interface architecture

chapter 3 has shown that some functionalities are more rudimentary. To enhance the functionalities and to allow a simpler access to SOSs, the architecture will be changed from a complete OpenLayers integration to a hybrid model. Figure 4.2 shows the changed

architecture. An external *interface adapter* will handle the access to the SOS interface, to hide the complexity. The resulting class layout is shown in figure 4.3. The most

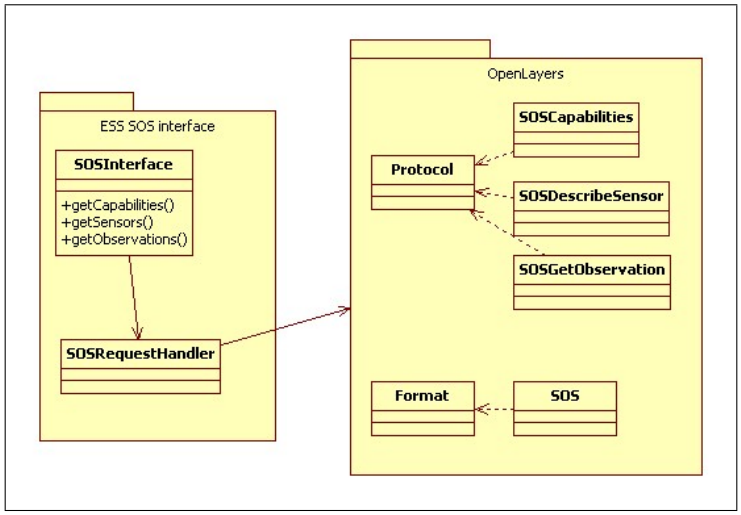

Figure 4.3.: The class layout of the ESS SOS interface

important advantage of this architecture is the reduced complexity to access OGC sensor server. The user does not have to know the internal structure of the SOS layout. For instance, if all measurements of one sensors are being requested. Based on the specific SOS, there might be several requests required. The interface adapter will hide this fact and returns the bundled answers.

Furthermore its not necessary to access SOSs through the OpenLayers API, but it's still possible to use Bartvde's OpenLayers integrated API. The interface definition of the ESS interface adapter offers two functions in addition to the constructor:

SOSInterface The constructor of the ESS SOSinterface class. It returns a new interface object including the capabilities of a SOS server, which URL is specified as a parameter.

getSensors It adds the positions of all sensors and meta information to the object of the interface.

getObservations It requests observations from one or more sensors. Optional time and spatial filters can be included.

A exact example, how the interface can be used, is shown in listing A.2 on page 91.

4.2.2. Rich Client Framework

The ESS framework will be used to build rich internet client applications, where a variety of *widgets* are available. These *widgets* should not be hard-wired to each other, to be able to use only a subset of the available *widgets*. For this purpose, the *observer pattern* was used to publish any changes. *Widgets* can register to an *observer* (event) and if it was triggered, the *observer* publishes the information to all registered *widgets*. In this thesis the *observer* is used as an event-system, that offers the following interface:

addListener Add a listener (observer) to an specific event.

removeListener Remove a registered listener from an event.

fireToListener Trigger a specific event. The observer will distribute the information to all listeners.

Using such an *event-system*, programmers of new widgets only have to know what events are used by the existing widgets. Furthermore, a requirement for the framework is the ability to access OGC sensor observation services. The *ESS SOS Interface* handles sensor server connections, but there must be an additional *interface manager*, to prevent redundant SOS connections of different widgets. In figure 4.4 an architecture overview is shown. The *interface manager* only offers one function:

getSOSInterface If the requested connection to the server already exists, it will return the existing connection. Elsewise a new connection will be created.

The data, returned by the SOS interface, must also be handled by the framework, to keep a consistent copy of the data. Additionally the data need to be restructured in

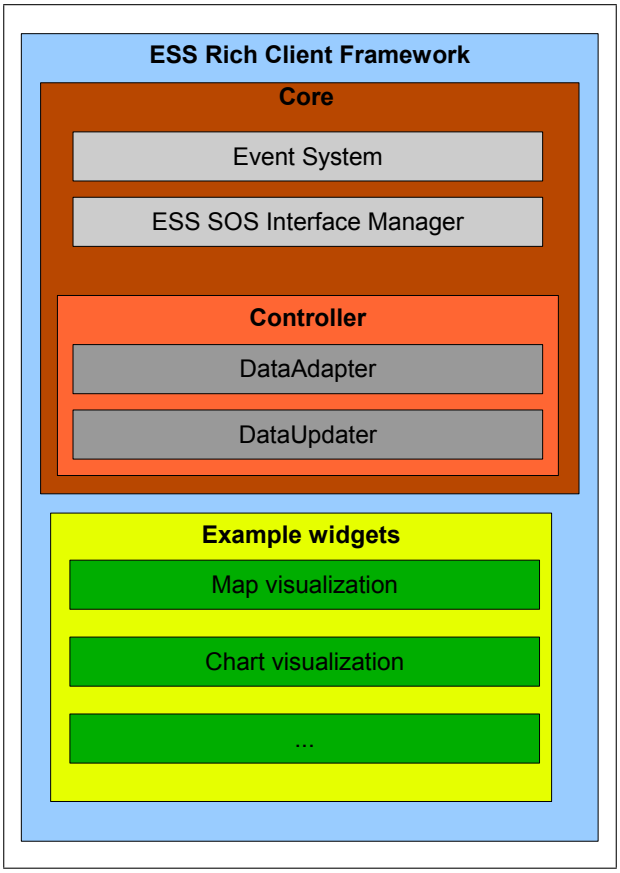

Figure 4.4.: ESS Rich Client Framework architecture

a **data model** that every widget understand. The *DataAdapter* will handle that task. This keeps also the potential to integrate more interfaces into the framework. In this case, only the *DataAdapter* has to be extended.

Each widget will be able to modify its own copy of the data and can still access the **data model** (a consistent copy) from the *DataAdapter*. Another *Controller* is the *DataUpdater* that allows to update already requested data. Both controller are being described in detail in chapter 5.2.3. In figure 4.5 the class layout of the ESS framework is shown. The framework itself consists of one class (**SOSGui**). Both, *controller* and *widgets* inherit from this class, to get access to the *interface manager* and *event-system*.

The *Input* widget will handle user-interactions with SOS servers, but the SOS interface itself will be managed by the *interface manager* of the core system. On this way, other

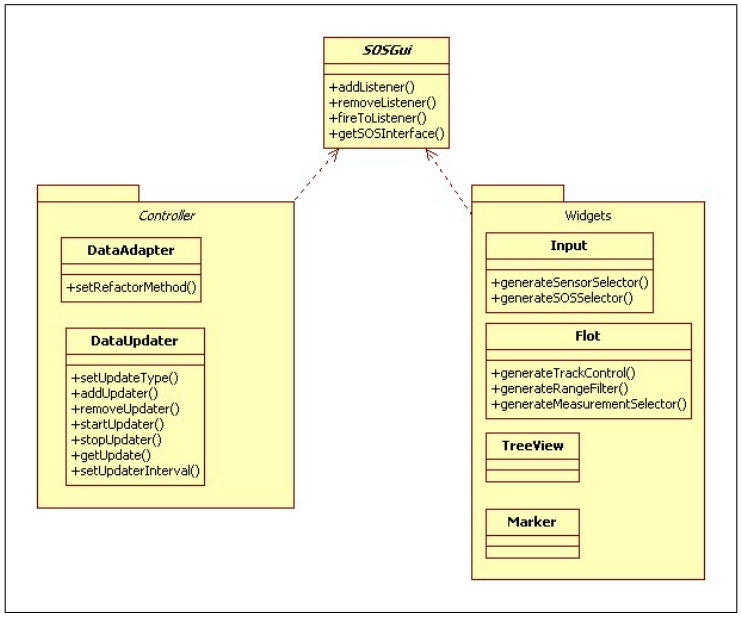

Figure 4.5.: The class layout of the **ESS Rich Client Framework**

widgets can also access the SOS interface, without producing redundant connections.

The visualization will be done by several widgets. The two most important widgets are the map and chart visualization. Their architecture will be described in next section.

4.2.3. Widgets

As defined in chapter 3.3, the framework should support great visualization tools. These tools will be designed as widgets. The architecture of the *widgets* is being described using the example of the *Input* widget, which is capable for user-interactions.

With respect to the *model-view-controller* pattern, the core of the framework (shown in figure 4.4) handles the **data-model**, including an observer. Widgets will serve the **view** and **controller** part. This means, that they are responsibly for the user-interface and the functional logic of visualization.

As shown in figure 3.5 on page 37, the basic SOS uses three different requests. The three requests plus an additional *update request* is explained in the figure 4.6.

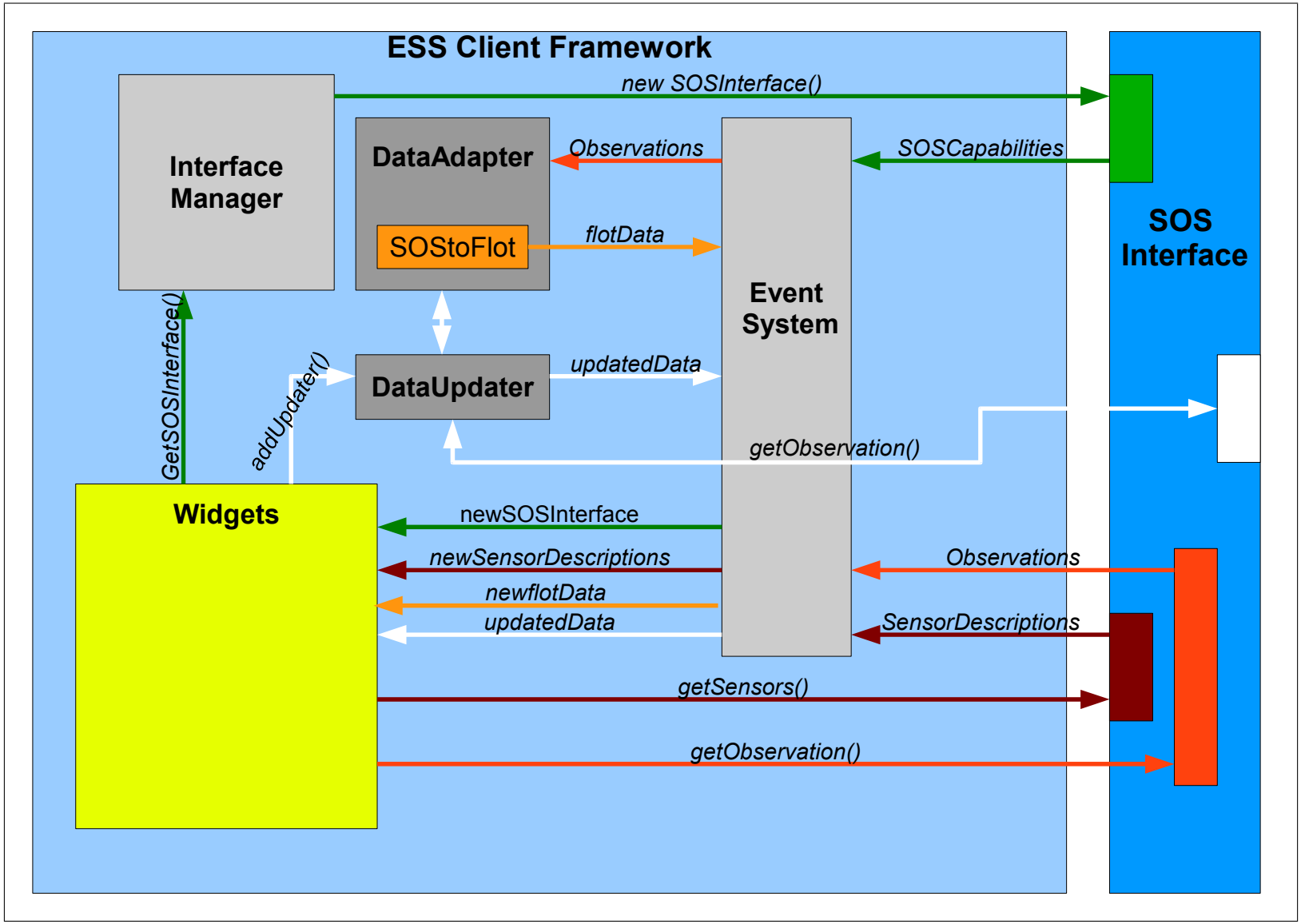

Figure 4.6.: The information flow th the SOS Interface, starting from a widget.

Step 1 An widget creates a new connection (green line) to an SOS server (getSOSInterface) with the help of the **InterfaceManager**. The response is being distributed with the **EventSystem** to all listening widgets.

Step 2 With the received SOS interface-object, a widget can request further information (getSensors) of all sensors (brown line). This information is also being distributed via the **EventSystem**.

Step 3 A widget requests sensor observations (getObservation, red line). The response is being preprocessed by the **DataAdapter**. The result (flotData) will also be distributed by the **EventSystem**.

Step 4 A widget can register an updater (addUpdater) to already received data (white line). The **DataUpdater** will send a appropriate *getObservation* request and add new information to the existing one. Again, the widgets will be informed by the **EventSystem**.

The second step is not coercively necessary to request sensor observations.

The received information (sensor positions & observations) are also distributed to the map (*Marker*) and chart (*flot*) widgets. With their help, it will be possible to visualize different aspects of the connection between sensors positions and the measured data. To show data in a chart, the *JQuery* plugin *flot* will be used and OpenLayers will surely offer the map. These two widgets will have a very strong connection, but using the event system, each one can be disabled.

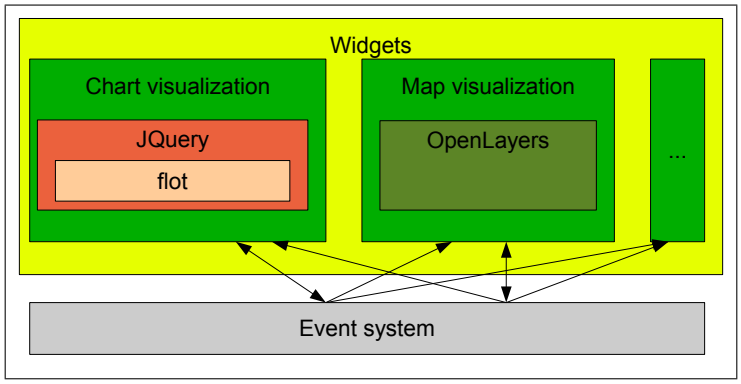

Figure 4.7.: Widgets architecture overview with chart and map visualization as example

5. Implementation

This chapter describes the implementation of the ESS SOS interface and framework. Some components of the framework are based on the SOS interface. As a result, the interface has been implemented before. The development environment will not be described. All tests of the AJAX client were done on the newest Firefox and Safari. The client might not work in other browsers.

5.1. SOS Interface

The interface is based on Bartvde's preliminary work. At first, the basic requests will be described including their communication chain.

Furthermore, advanced *GetObservation* requests will be described, to support spatial and temporal filters.

5.1.1. Basic requests

The example interface from Bartvde is using the *getFeatureOfInterest* request instead of *describeSensor*, to get the current position of an sensor. Since one requirement is, to support the core profile of the OGC SOS standard including servers which do only understand the core profile, the *getFeatureOfInterest* request will be replaced by *describeSensor*. Bartvde's *GetObservation* does yet not support the DataArray response - only the DataRecord. This will be changed because of less overhead produced by the

DataArray. It should always be used. The *GetCapabilities* is not yet parsing all relevant information. This will also be changed. The following three sections are named, according to the offered functions by the interface API.

SOSInterface

To get the capabilities of a SOS, a new SOSInterface object must be created. It takes one parameter, the URL to the server. Listing A.2 shows an example, where a server is being requested. As explained in chapter 3.2.1, the capabilities document contains the necessary information for a client, to request further data from the SOS server. The most interesting section is contents, where all offerings and its corresponding observations and sensors are listed. Anyway, the whole document will be parsed and stored in an object. The *getCapabilities* request will directly be called, if a new SOSInterface object was created, thus it is not directly available for the API.

getSensors

After parsing the capabilities document, the function **getSensors** can be called to request all available sensors. For each sensor a *DescribeSensor* will be sent.

On this way, all sensor positions are being received, but it also generates some overhead in comparison to only one *getFeatureOfInterest* request. On the other hand, a *getFeatureOfInterest* do only work, if *FeatureOfInterest* and *procedures*(sensors) are the same. This is based on the SOS's logic structure and can not automatically be identified yet. It's also possible, that a *FeatureOfInterest* is a spatial area. To prevent that problem, *getFeatureOfInterest* will not be used at all. The **getSensors** function takes one argument, the callback that will be executed if all responses were received. The callback itself will get an object as parameter, containing the received information.

getObservations

The *GetObservation* request is a little more complicated. The framework should be used to show sensors in a GIS and possibly visualize the observed data. But the SOS standard allows to structure the logic organisation of sensors, spatial objects and measured information completely free. For example, a *GetObservation* request must specify an offering. *Procedure* (sensor-id), *observed properties* and *FeatureOfInterest* are not coercive necessary. But the SOS standard allows an offering to be a spatial object, a sensor, an observed property or something else. This means that a request for all observed properties of one sensor, multiple requests may be necessary. To hide the different abstractions, an algorithm was written that searches all offerings for the requested sensors and directly creates the necessary requests. All responses will be bundled. The user of the interface will not recognize if one or multiple requests have been sent. The interface will response if all responses of the SOS have been received. The **getObservations** function takes two arguments:

callback The callback will be executed if all responses were received. It get one parameter, an object containing the received information ordered by sensor-id.

options Consisting of an object that needs the following parameter:

getOfferings A list of offerings

procedures A list of sensors

observedProperties A list of observed properties

filter A filter object. The structure of this object will be described in next section.

Except of *procedures*, the other parameters can be null. But *procedures* must contain at least one sensor-id (procedure).

In listing A.2 on page 91 an code example is shown, how to access the SOS interface. It creates an interface object, requests all sensor descriptions and requests all observations of each sensor.

5.1.2. Filters

A filter reduces the amount of data receiver from a SOS server. Filters are specified in *GetObservation* requests. But requesting data from a SOS server is possible without specifying any kind of filter. If such an request is sent, the server will return all observations it ever made. To prevent this, there should be at least one temporal filter. If the **getObservations** function of the interface is called, it is possible to specify a filter. Actual there are temporal and spatial filter possible.

Temporal It specifies time constraints to the requested data. Following constraints are possible:

- Before a point in time
- After a point in time
- Equal a point in time
- During a time-range
- Latest, the last observation the sensor made (not OGC conform but still relevant)

Spatial Actual there is only one spatial filter possible, a polygon. But with its help its possible to emulate other geometry's.

These temporal and spatial filters are assigned to the request via a parameter. It is possible to combine spatial and temporal filters.

The second parameter of the **getObservations** function takes the argument "filter". It's reserved for the filter object. The structure of the object is shown in listing A.3 on page 92. If the filter object is not null, it must contain an spatial filter (featureOfInterest) or an temporal filter (eventTime) or both. The following list explains the structure:

featureOfInterest The spatial filter: Only polygon was implemented yet.

SpatialOperator "BBOX" or "Contains" or "Intersects" or "Overlaps"

GeometryOperand "gml:Envelope" or "gml:Polygon" or "gml:Point" or "gml:LineString"

lowerCorner Used for bounding box

upperCorner Used for bounding box

lineRing Used for *gml:Polygon*. it takes a list of lat/lon coordinates.

eventTime The temporal filter:

TemporalOperand "gml:TimeInstant" or "gml:TimePeriod"

TemporalOperator "TM_During" or "TM_Equals" or "TM_After" or "TM_Before"

beginPosition ISO timestamp or *latest*

endPosition ISO timestamp

If the interface is directly used, there will not be a logic verification of the filter. The user must know what actions has to be performed, but using the examples, there won't be a problem.

5.2. Rich Internet-Client Framework

The framework for the *Rich Internet Applications* will be integrated into *Flex-I-Geo-Web* as an own component. But it will also be possible to run completely independent from *Flex-I-Geo-Web*.

It consists of three different modules: the core including the controller and of course the widgets which are used to build the applications. Widgets are being separated into basic user-interaction and the rich visualization widgets - but every widget can be en- or dis- abled for an application.

To support common design pattern, an extra class for inheritance was used. John Resig [22] introduced in his blog an interesting way for classic inheritance. Listing A.1 on page 90 shows the appropriate class. This class in combination with another

modification of the common class definition was used to apply common private/public namespaces to JavaScript classes.

5.2.1. Flex-I-Geo-Web

The framework will be integrated into the Flex-I-Geo-Web, to supply the framework on a known platform. In the context of Flex-I-Geo-Web, I made different changes, to enhance the usage. On the one hand, the complete GUI was changed to use JQuery. In addition several internal operations were changed from the scratch. For example, components were loaded into *iframes*. This approach was very simple, but also produced different problems with JavaScript, especially OpenLayers did not function correctly. The new approach loads all HTML elements of an component into an own *div* element. The new JavaScript files are included to the HTML header. To initialize a new component, a predefined class must be provided by every component. This class is being executed by the JavaScript *eval* command. On that way loaded components are working instantly.

5.2.2. Core modules

The two core modules of the framework, the *SOS Handler* and the *Event System*, could be used by every widget. They are included to the main class (**SOSGui**) of the framework. Every widget has to inherent from **SOSGui**, to access the event-system and the SOS-handler.

Event-system

The event system is used by widgets and controllers to communicate with each other. The following interface was created:

addListener If an widget or controller is interested to a certain event, it registers itself to this listener. Of course the programmer must know the name and the parameter assigned to the event. The function *addListener* takes three parameters:

- The *name* of the event interested in (the used event names could be found in the source code of each widget).
- The *callback*, that should be executed if the event is triggered.
- In addition, a *scope* can be assigned. Sometimes it is necessary to execute the callback in a certain scope. Scope of JavaScript refers to the keyword *this* and may change, based on the function, where another function is being executed. [1]

removeListener Its also possible to remove a listener from an event. It takes the *name* of the event and the *callback* as parameters.

fireToListener The complement of addListener will trigger the events. It takes the *name* of the event and an object as parameters. The object can have any structure. Of course its also possible to trigger events, which do not exist (if no listener was registered) but there won't happen anything.

The event-system can only be used from classes which inherited from the **SOSGui** class. The information-flow of widgets using the event system is shown in figure 4.6 on page 51.

SOS-handler

The framework should use the OGC SOS interface to be able to access the sensor observation services. The SOS interface needs for every SOS server an own object (instance). This object will store necessary information from the servers capabilities and sensor descriptions. To prevent creating multiple instances for the same server, a handler was created. It consists of only one function:

getSOSInterface The function take one parameter, the *URL* to the server. It will return the interface-object, created from the SOS-interface.

[1] http://www.quirksmode.org/js/this.html

This handler is necessary, because different widgets might use the SOS interface. This method prevents redundant data.

5.2.3. Controller modules

Controller modules are widgets who are available for every applications. They do not add a user interface. Instead, additional functions are offered that are responsible for data management. Actual there are two controller modules necessary. One module must handle the received data from a SOS server. Another module offers the functionality to update received data.

It's necessary to be ready for more than one interface. Interfaces are not capable of re-factoring the received data. This must be done by a handler, which offers the re-factored data for the widgets. On this way, other data sources are possible, without changing all widgets.

Data handler

The received data from the SOS interface is stored in an object with its own data-structure. To re-factor this data in a format, that all widgets understand, a handler for SOS measurement collections (dataArray) was implemented. The widgets will receive an event through the event-handler if new data was received and re-factored.

Data updater

The SOS server are receiving new observations from their sensors periodically. To update the data on the client side as well, there must be a service that requests the data and updates old data-sets. But why should this be done by a controller instead of the interface itself ? There a multiple reasons:

- There is no real "update" function supplied by the SOS standard. A normal *getObservations* request with a special filter(multiple) will be used instead.

- The update frequency not known. The user must set it on his own.
- The old data needs to be updated anyway.

The *dataUpdater* supply an interface for different kinds of updates. Following functions are available:

setUpdateType The update type can be set on *latest* and *after*. Latest will return only one observation, the latest. The *after* type returns all observations after a specific date. The date cannot be set, this mode brings the data up-to-date.

addUpdater If data was already requested from a sensor, an updater can be added. The function takes two arguments. The *SOSUrl* and the sensor-id (procedure).

removeUpdater Analog to to *addUpdater*, this function removes the updater. It takes the same arguments.

startUpdater Starts updates for all set updater.

stopUpdater Stops all updater.

getUpdate Call one update for all set updater manually.

setUpdaterInterval Set the update-interval in milliseconds for all updater.

5.2.4. Widgets

Widgets basically offers any kind of user-interface and its according functionalities. For demonstration, multiple widgets have been developed, including rich visualization possibilities for the mobile sensors.

As explained before, every widgets has access to the core and controller modules. Each one, has to inherit from the super-class *SOSGui*. In listing A.4 an example shows, how to use inheritance in this framework. The next listing A.5 shows how to instantiate a widget.

The following sections explain some exemplary implemented widgets. Images of these widgets are shown in the chapter 6. All these widgets are using the event system to communicate. This work will only name important events. A list of all events, a widget generates and listens to, can be found in the source code.

User interaction

One widget (**Input**) was created to handle very basic user interactions to SOS server. It creates HTML *form* elements and append them to a given parent. Using these elements, it is possible to request predefined SOS servers. Following operations are possible:

- Requesting specific sensor server. If one server is being requested, directly after the capabilities document was parsed, the *getSensors*(see chapter5.1.1) function will be executed. On this way, all sensor positions will be available without further user-interactions. If the function is ready, the event system will be triggered on **newSensorServer**.
- Get the observations of specific sensors. This operations will trigger the event system on **newSensorObservation**.
- Set basic time(range) filter.

In combination with these operations, another widget(**Control**) will add the possibility to generate a spatial polygon filter. The filter will be applied on the next sent request. The spatial filter will be generated from marked areas, which the user has to "draw" on the OpenLayers map.

An very simple example-widget(**TreeView**) demonstrates the event system. It shows all requested server and their sensors in a tree, build from HTML elements.

5.3. Visualization

The OpenLayers map, combined with the chart, is probably the most interesting part of this work. As already mentioned in the introduction, these tools supply rich visualisation functionalities. In particular, an approach is shown, how to visualize mobile sensors with their produced tracks. Additionally an illustration of non-mobile sensor measurements is solved on a similar way.

The functionalities are implemented as widgets. How the widgets work together was already described in the previous section. The explicit classes and the methods will not be described, instead their functionalities in detail.

5.3.1. Map & Chart

Using the OpenLayers map, some basic functionalities are supported which are known from other GIS.

- Show sensors on the map with interactive markers.
- Show tracks on the map (different colors)
- Bounding boxes for spatial filter (for getObservation requests)

On the other hand there are the basic chart functionalities of *flot*, the JQuery plugin for chart visualization. The **Flot** widget supports:

- Chart with multiple data series (measurements).
- Different colors, dual axis support, time series support
- Background grid, Build-in legend
- The whole chart is interactive with following features:
  - Cursor tracking with the possibility to track the position on the map according to the mouse position on the chart (with info bubble).

- An area on the chart can be marked.
- Possibility for zooming into the chart and resizing of the whole chart.

Flot itself supports more functionalities, but they are not used yet. More about that can be found in its homepage[2].

To get an better overview of the received and shown data, an extra element (HTML form) was created that enables the user to turn on/off explicit data series. In addition, an second chart is shown that helps to zoom in specific sections of the main chart. It is also possible to assign to each data series an own unique color.

But the main purpose of the map - chart interaction is the visualisation of partial tracks with respect to their measurements. Or to illustrate the measured observations of multiple sensors. For this purpose, the *flot* library was changed, to support multiple marked areas on one chart. These *rangeFilters* are used to filter the observations and visualize results on the map. To work with multiple marked areas comfortably, to each a unique color could be assigned. The next two chapters will describe the difference between mobile and stationary sensors in this context.

5.3.2. Mobile sensors

Mobile sensors generate different measurements in addition to their position. If multiple observations are requested, a normal track could be generated. Now, using the *rangeFilters*, the track with it's measurements could be interpreted in different ways.

In the context of mobile sensors, there are basically three different kinds of these *rangeFilters* possible:

- X-filter: The x-axis on a chart is typically used for time. One or more *rangeFilters* applied on the x-axis will show parts of the track according to the specific time ranges.

[2] http://code.google.com/p/flot/

- Y-filter: The y-axis shows the value of the measurements made by an sensor. *RangeFilters* applied of this axis will show only the coordinates where the active measurements are in the range of the filter.

- XY-filter. If both *rangeFilters* are combined, the intersection of both will be shown. The color of the Y-filter will be used for the track.

To enhance the visual effect, the color of the *rangeFilter* is used to mark the track. On this way the track could get very colorful.

On top of that, the active *rangeFilters* could be shifted and resized. The map will be updated live. This is very computationally expensive, but there are several interesting fields of application that are shown in the chapter 6.

Clustering

Another feature to visualize observations with a spatial reference, is the clustering strategy of OpenLayers. Many single features could be clustered into groups. Based on the number of features, the size of the shown marker could vary. On this way the map is not flooded and the computational expensive also decreases. The next chapter will show explicit examples of their advantage.

5.3.3. Stationary sensors

Using stationary sensors, the *rangeFilters* serve a similar functionality compared to mobile sensors. It is possible to create an analog effect, but multiple sensors with different positions are needed. Instead of highlighting the track, the sensor position will be highlighted, based on the measured data. Using one observed property from several different located sensors, a range filter will only show (or highlight) those whose measurements are within the range of the filters.

6. Results

The results of the thesis are shown in this chapter. An example, how the framework works and what it is capable of is shown. Of course there are also several limitations from different sources. There are also issues about the computational expensive processing of JavaScript code.

6.1. Usage

This chapter explains, what actions have to be performed to use the *ESS Client Framework*. Not to start from scratch, a working Apache server and a basic knowledge about JavaScript and HTML is a prerequisite. Of course, all necessary JavaScript files must be included to the HTML header. The framework does not handle the necessary files on its own.

Since the SOS interface is used by one of the frameworks widgets, it will not be described here. How to use it was already described in chapter 5.1 on page 54, including code examples on page 91.

In listing 6.1 the JavaScript part of the HTML file is shown. All available widgets are included there. The placeholder "any_div_element" should be replaced by an *div element*, where you want to show the form elements.

```
1 <script type="text/javascript">
2
3   /*
4    * Create an instance of the userinput widget
5    * It takes one parameter, the div element, where it should be shown.
```

```
6       *
7       * Two functions are also called, to create the form elements.
8       * The SOSSelector needs a list of predefined sensor-server.
9       */
10     var userinput = new sosgui.Input(document.getElementById("any_div_element"));
11      userinput.generateSOSSelector({SOS1 : 'URLtoSOS1', SOS2 : 'URLtoSOS2'});
12      userinput.generateSensorSelector();
13
14      /*
15       * Create the control widget to be able to create polygon filters.
16       * It needs the OpenLayers map object and an div element.
17       *
18       * The function creates the form element.
19       */
20      var control = new sosgui.Control(OpenLayersMapObject, document.getElementById("
             any_div_element"));
21      control.addSpatialSelector();
22
23      /*
24       * Create the marker widget, that is capable of the flot->OpenLayers interaction
25       * It needs  the OpenLayers Map object
26       */
27      var sosmarker = new sosgui.Marker(OpenLayersMapObject);
28
29      /*
30       * Create the flot component. The constructor takes two div elements.
31       * The container for the chart, and another where the chart legend is shown.
32       *
33       * The three function create the form element, needed to interact.
34       */
35      var flot = new sosgui.Flot(document.getElementById("any_div_element"),
36           document.getElementById("any_div_element"));
37      flot.generateTrackControl(document.getElementById("any_div_element"));
38      flot.generateRangeFilter(document.getElementById("any_div_element"));
39      flot.generateMeasurementSelector(document.getElementById("any_div_element"));
40  </script>
```

Listing 6.1: How to instantiate the framework with its existing widgets.

To create new widgets, an example is shown on page 92. The available events could be found in the source files.

6.2. Example

To give a better understanding of the achieved results, three different examples are shown. The focus of this work was the visualization of mobile sensors. On the other hand, there are sensor server available which offers data from stationary sensors. Their illustration is similar to the mobile ones. In addition another feature, the cluster strategy from OpenLayers was uses to visualize lots of different positioned features. All examples

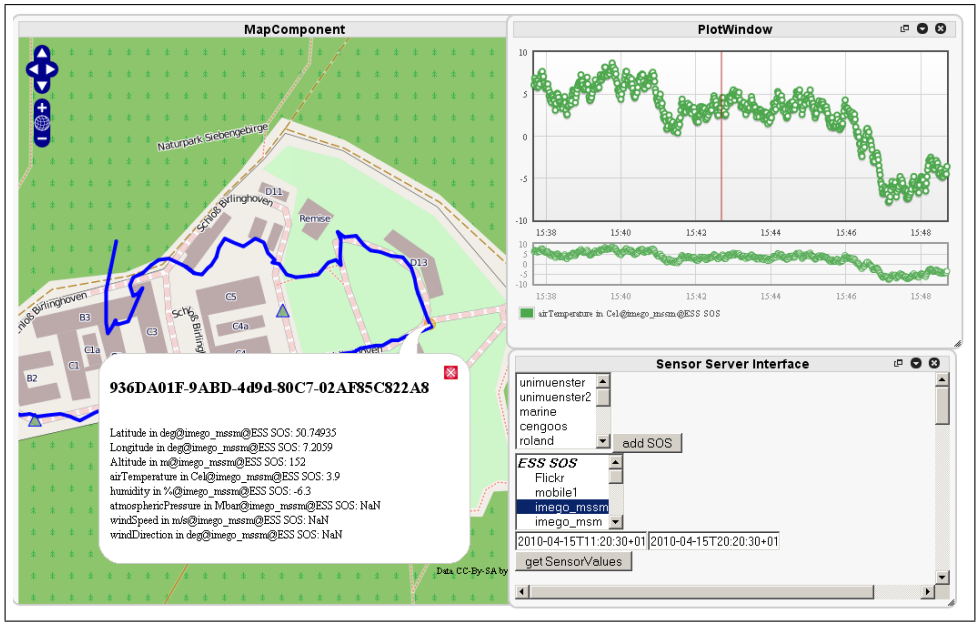

Figure 6.1.: A track produced by a mobile sensor. The chart show the temperature.

are using a small example-application created with the framework. The framework is using the SOS interface to get access to sensor information. A static list of known SOS server was added to the example application. Of course, the example-application is browser-based.

6.2.1. Mobile sensor

The visualization of tracks produces by mobile sensors was the main purpose of this work. To give a better image, how the widgets are working, several screenshots are

shown. In figure 6.1 the standard layout is shown, where a SOS was already requested.

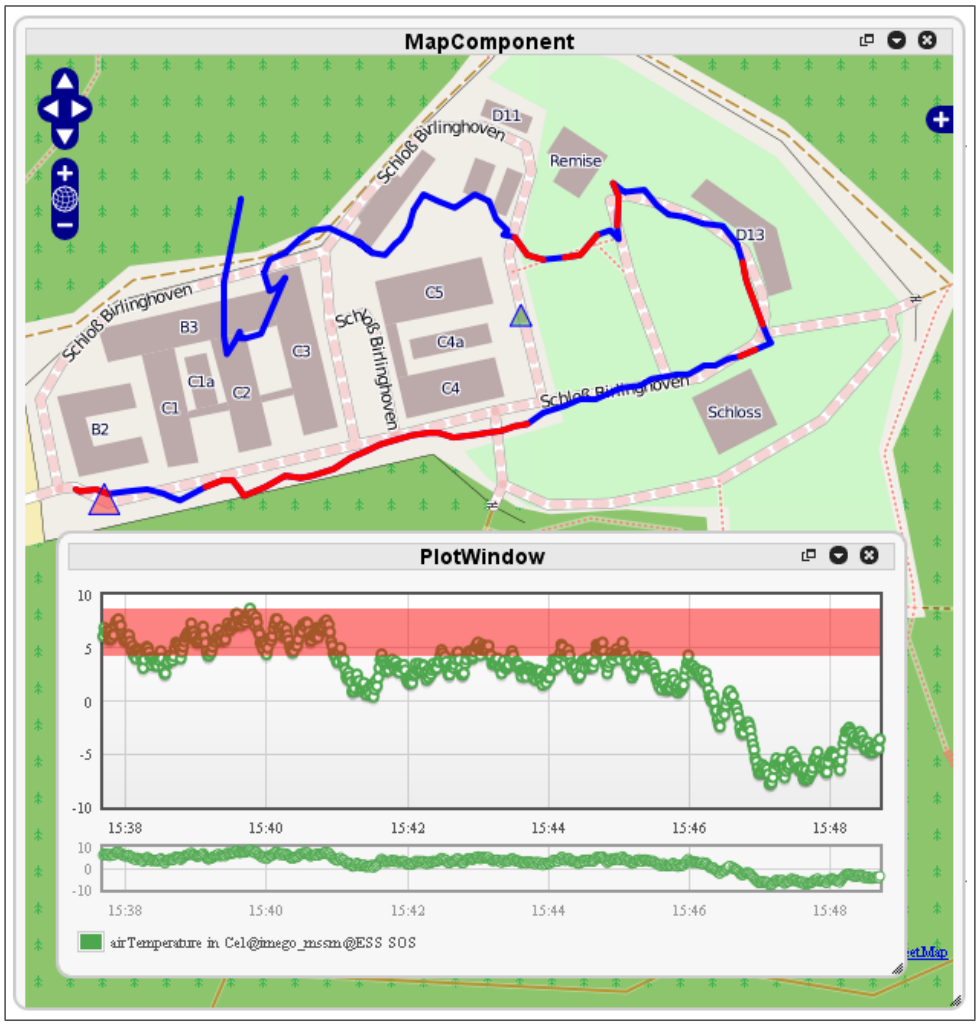

Figure 6.2.: The coordinates, where the temperature is higher than 5 are highlighted.

On the left side, the OpenLayers map element is shown. In this case the OpenStreetMap maps are used, but commercial (Google, BING...) are also possible. The blue track is a small track around the Fraunhofer area. The information was requested from the Fraunhofer intern SOS. If the returned information contains coordinates, the track can be drawn on the map. Additionally the mouse-tracking with info-bubble is already activated. According the the mouse-position on the chart, a marker with an info-bubble is shown in the map.

The upper right shows the flot chart window. It shows data series from the SOS. In this case, all series have been disabled except of temperature.

The lower right shows the control window where basic HTML elements are supplied to the user.

In figure 6.2 the windows have been rearranged a little bit. Only the map and the chart window are shown. A *rangeFilter* was added to the chart (the red area). According to the marked range on the chart, coordinates within the range, are highlighted on the map, using the same color of the *rangeFilter*. In figure 6.3 another *rangeFilter* was added.

On this way, it is very easy to find the interesting parts of the recorded data of an SOS. In this case, the mobile sensors produced tracks, and the parts where the according observations contain important information can be highlighted.

In figure 6.4 another *rangeFilter* was added. This filter is different, because it filters time-ranges. Now, only the intersection is shown in the map. In addition, the original track (blue) was disabled. Only the "interesting" parts are shown. These time-filters are useful, if a mobile sensor are circling in the same are. If the whole track would be shown, no one could extract the important information.

Another feature is the possibility to shift and resize existing *rangeFilters*. The map is being updated live.

The next figure (6.5) shows the same track using the cluster strategy of OpenLayers. The clustering merges features, that are close to each other, to one larger feature. The more features are clustered, the larger is the shown feature (in this case circles). This is useful to visualize, how fast a mobile sensor was moving - or if it was moving in circles. In this case, larger features are shown on the map. In section 6.2.3 another example of clustering is shown with non-track information.

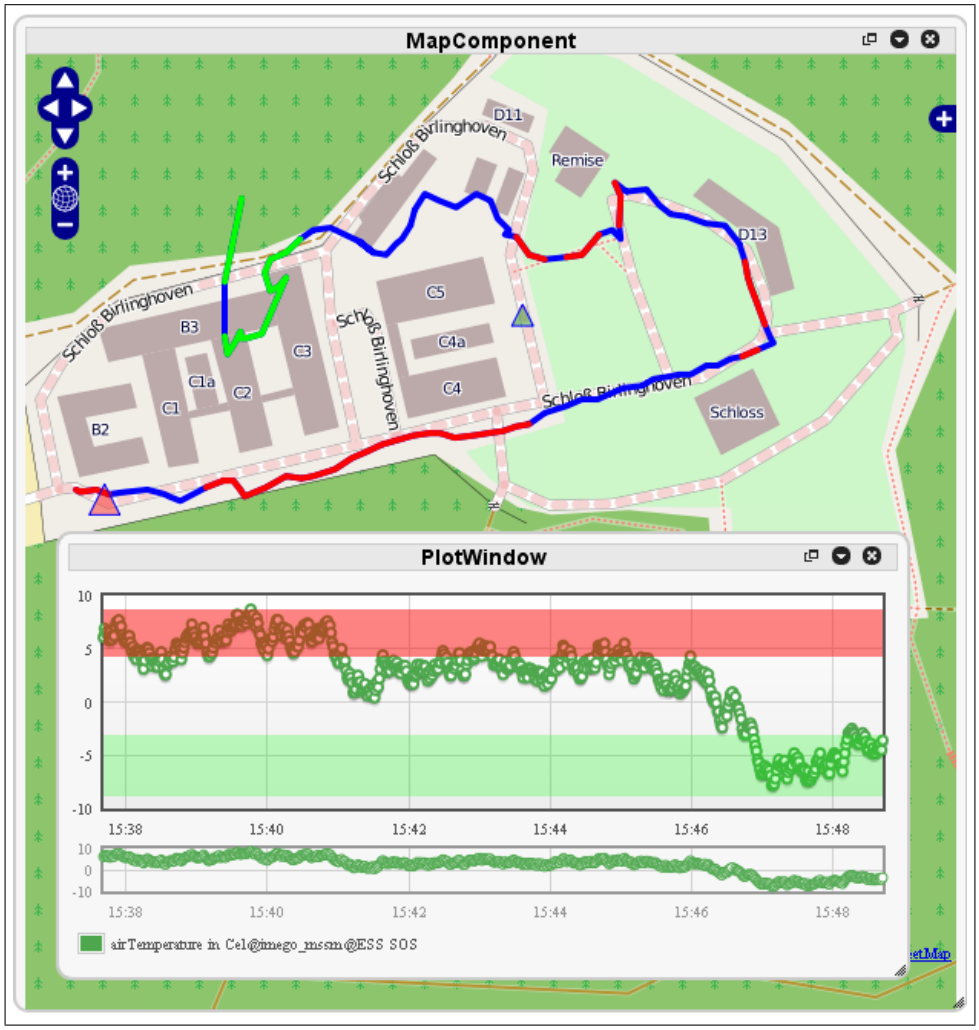

Figure 6.3.: Attitudinal, the coordinates where temperature is lower than -4 are highlighted.

6.2.2. Stationary sensors

Using the technology of *rangeFilters* with stationary sensors, a similar behaviour compared to mobile sensors could be achieved. But in this case, multiple sensors are necessary.

In figure 6.6 the SOS from UniMuenster was requested. They have two sensors which are observing several phenomena. The chart only shows the temperature of a three days time range. The two small triangles on the map are the sensors.

In figure 6.7 a *rangeFilter* was applied to the chart. On the map the sensor is high-

Figure 6.4.: Only the intersection of the two range filter and the timeFilter is shown. The original track was disabled.

lighted, where the measurements are within the range of the filter. In the next figure (6.8), the *rangeFilter* was shifted. Now both sensor measurements are within the range, and both are highlighted on the map. Applying an additional time-range filter to the chart(figure 6.9), again only the intersection is used and as a result, only one sensor is highlighted.

In this example only two sensors are used, but if a large grid is being used, very detailed information of the measured phenomena could be generated. For example this

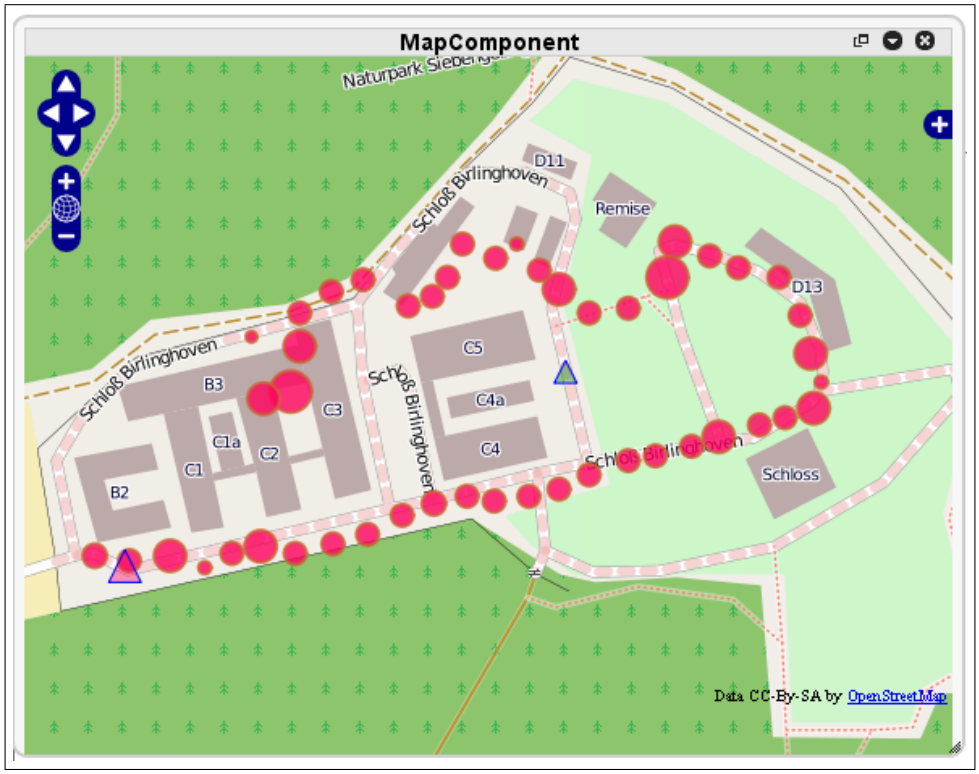

Figure 6.5.: Clusters instead of a track. Many measurements with a short distance are clustered to larger symbols.

technique applied to a grid of sensors measuring toxic concentration in the air. Using the different *rangeFilters* would generate a fast analyse where, when and how high the concentration was.

6.2.3. Cluster

As already described in section 6.2.1 the clustering strategy merge many features to one, if they are in the same area. The number of features shown at the same time could be set freely. Additional the size of the shown feature could be manipulated. In this case, the size is depending on the number of clustered features. But the function might change in other examples.

In figure 6.10 flickr information was used. Flickr is a service, where users can upload

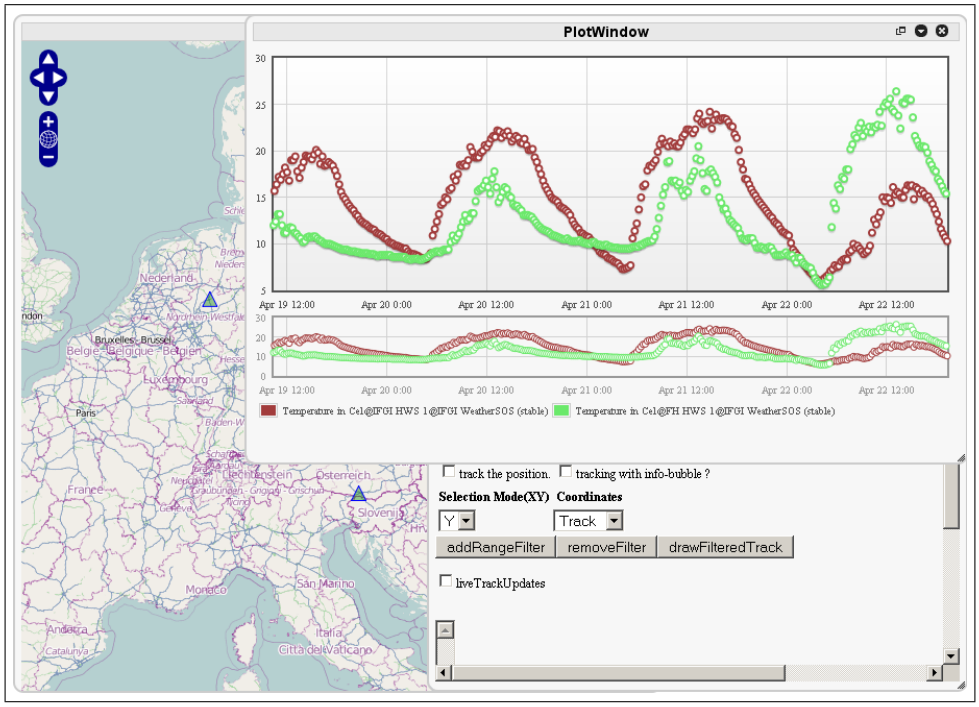

Figure 6.6.: The temperature of two stationary sensors. The sensors are marked with triangles on the map.

spatial tagged pictures, they have made. These data was uploaded to our SOS and can be requested. Im only using the coordinate and the time, where the picture was made. On this way, it can be visualized, where how many people have made pictures (and uploaded them). The hotspots are shown on the map - in this case the are of Berlin using a time-frame of three months.

In figure 6.11 the time filter was resized, to show only several days.

The figure 6.12 shows a marked are on the map. Using this marked area as a spatial polygon filter, applied on the *getObservation* request, returns only data of this area.

6.3. Limitations

During my work on the SOS interface and the framework, I had to make several kind of trade-offs, not to get stuck. So, several limitations are the result. On the one hand,

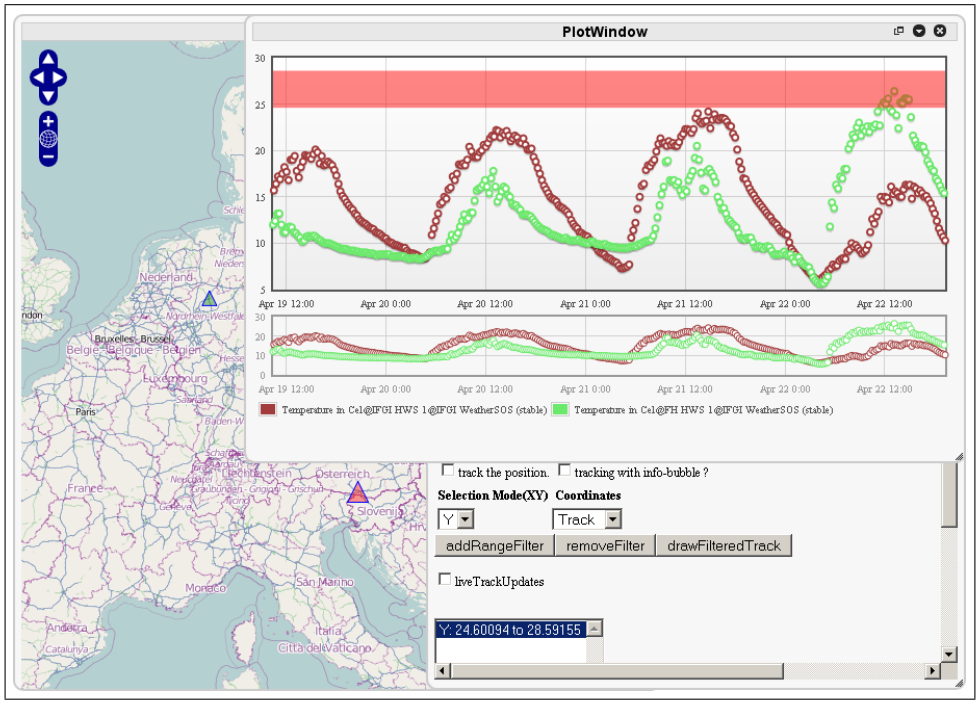

Figure 6.7.: Using a rangeFilter, the sensors in its range are highlighted.

the SOS interface is at some points limited. Of course there are also some issues of the framework with its example-widgets.

SOS interface The interface itself had to be limited. The SOS standard can be interpreted on different ways and there are several operations which serve the same purpose. This had to be restricted.

- Some servers are using only get or only post requests. My interface only uses "get" for the getCapabilities request. The other requests (describeSensors and getObservations) are sent as 'post' requests.
- The request *getFeaturesOfInterest* is not used, because the core SOS profile does not support this feature. Instead of one *getFeaturesOfInterest* for one SOS, an *describeSensor* for each sensor is used. This generates some overhead, but all servers should understand it.
- The 'getObservation' request supports only the *dataArray*. Some servers are

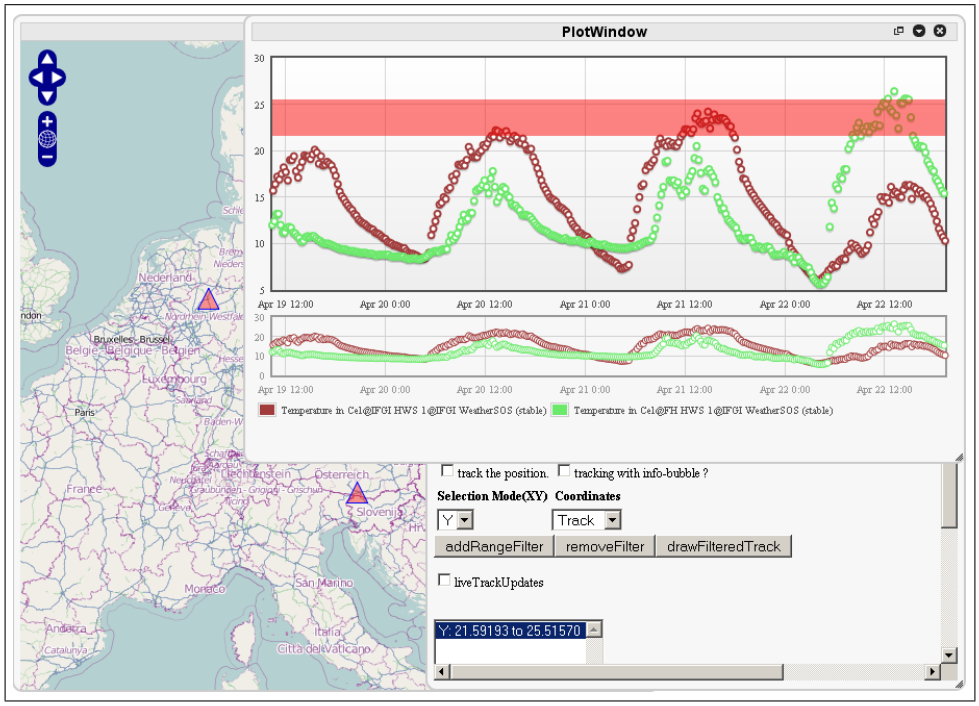

Figure 6.8.: Using a rangeFilter, the sensors in its range are highlighted.

sending the *dataRecord*, if only one observation was requested. In other cases the *dataArray* is being used. The *ESS SOS Interface* currently ignores all *dataRecords*.

- The location of the sensor is taken from the answer of the *describeSensor* request. The OGC standard allows to include a *location* and/or a *position* tag to specify the sensors location. I did not really made a difference between those to versions. In addition the unit to describe the sensors location is mostly lat/lon, but some servers are using *northing/easting*. My interface only accepts latitude/longitude units.
- The whole implementation is not perfect at some points. I just started with JavaScript during the thesis. Thus there are issues about the code.

Framework The framework is not really limited. It complies with its given tasks. But there are points that should be improved.

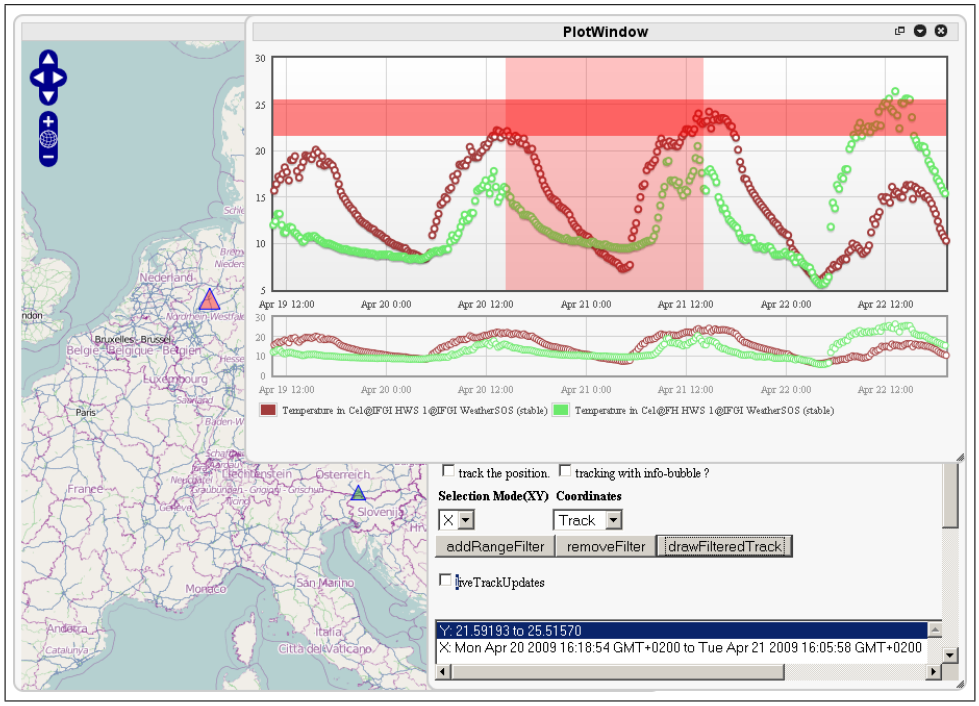

Figure 6.9.: An additional timeFilter reduces also the highlighted sensors.

- The inheritance method I used was taken from John Resig, but a similar version is already included to the *Prototype Framework*. And this framework is included to OpenLayers anyway. Thus the prototype version should be used.
- Some of the example widgets should be partitioned. Each widget is only one class with lots of functions. This should be changed, using inheritance. Especially both widgets capable of the visualization are much too large.
- The ESS SOS Interface recognises, if operations from a server are not supported. But this information is only shown in the JavaScript console. There must be an additional widget, which inform the user if something is not working correctly.
- The algorithms used for the visualization widgets are really bad. They are producing too much CPU load. But this computational complexity will be

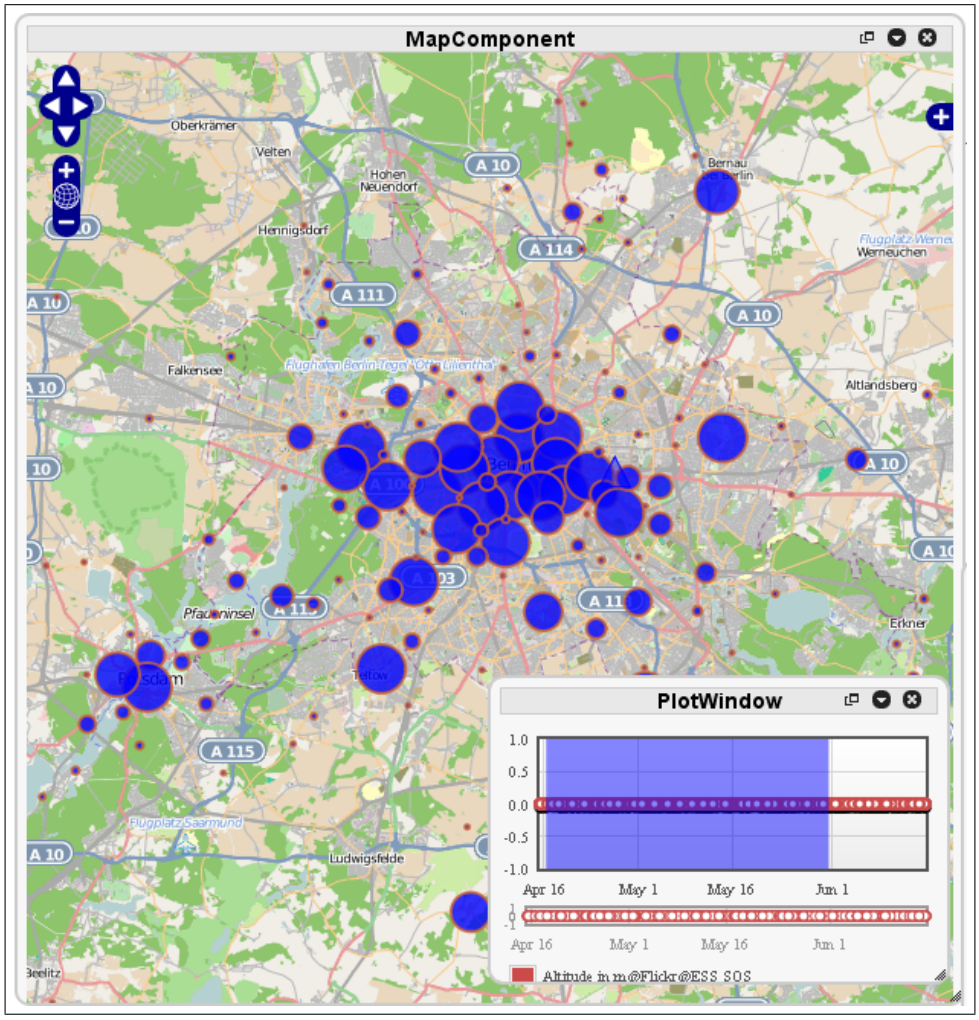

Figure 6.10.: Flickr photo coordinates(several thousand) in a short time-frame shown with clusters.

discussed in detail in section 6.4.

6.4. Benchmarks

The reader of this work might already guess, the operations performed by the SOS interface and the framework with its widgets are somehow computation expensive. JavaScript is only a script language and there are very restricted limits.

In figure 6.13 a small chart of made benchmarks is shown. The benchmark was done

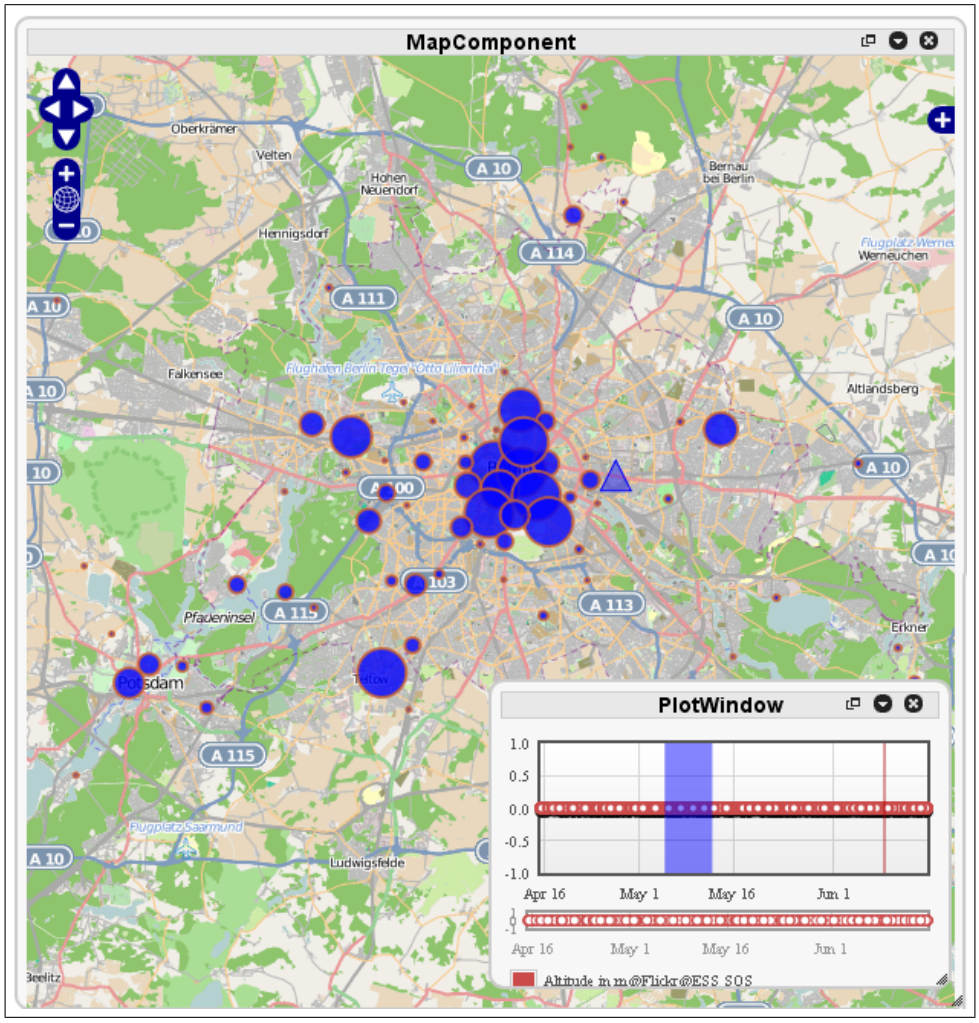

Figure 6.11.: Using the timeFilter, a more specific time interval can be shown.

with *Firefox 3.6* on a Intel DualCore 3 GHz. The red line shows the time, a SOS needs for the response. The blue line shows the processing time, of the received data. The processing includes the XML parsing, interpreting and plotting into the chart. The computational complexity seems to be exponentially. In addition, all browsers are using a kind of interruption condition, if JavaScript needs too long being processed. As described in section 3.1.3, Firefox stops after five seconds, working on the same function. In my benchmark, Firefox requested me to stop the script after 16 seconds. During this time it ran through different parts of the program and suddenly stopped. On this

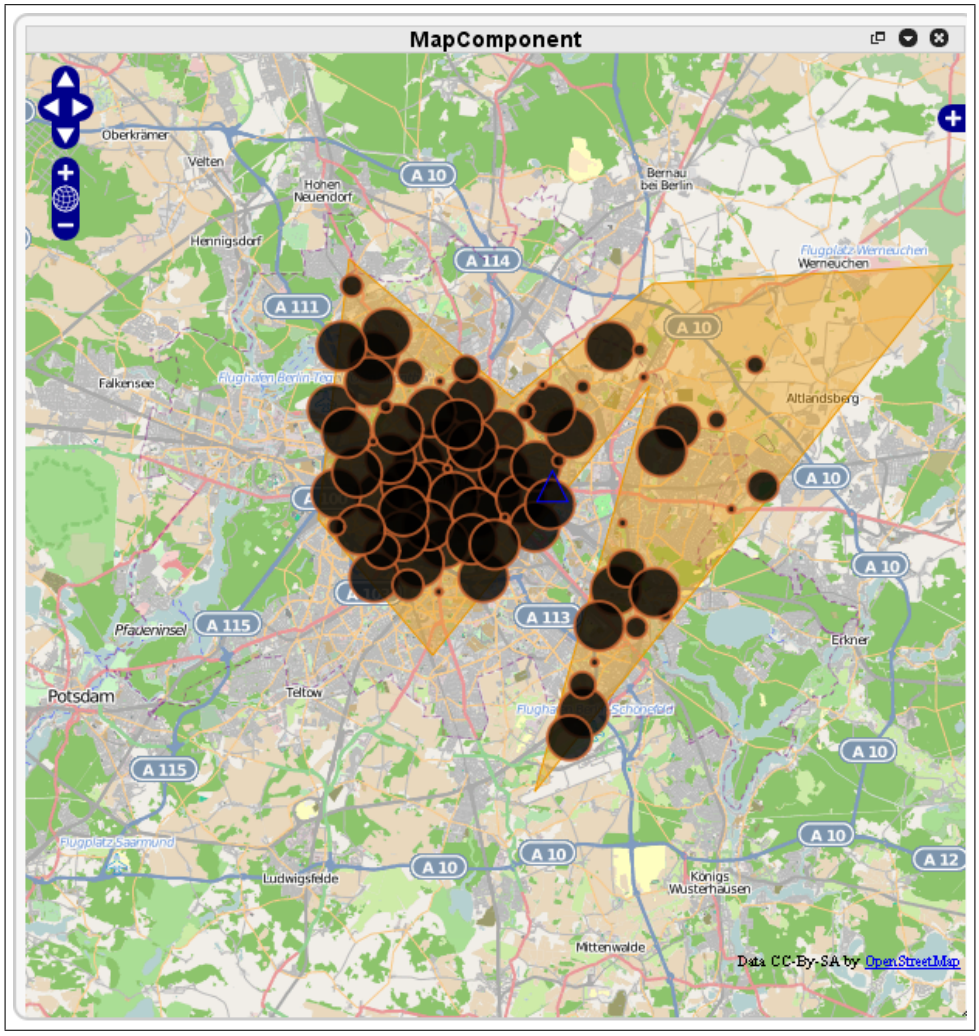

Figure 6.12.: A spatial polygon filter applied on the flickr information.

machine the limit is 8600 requested samples. Each sample consists of 6 measurements (including timestamp).

One weak point might be flot. It needs for each data series time/value pairs. Thus, the whole received data must be rearranged and is being extended by approximately 50%.

Another really expensive operation is the shifting and resizing process of the *range-Filters*. I did no benchmarks on this, but if too many samples are used, the user-interface gets really slow. To improve the performance i implemented a version that

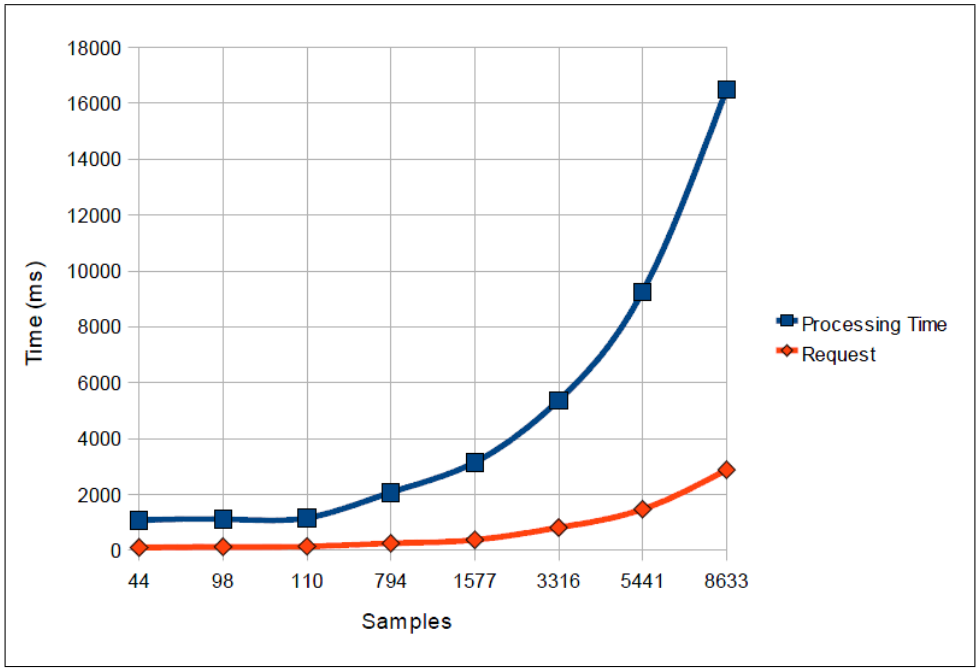

Figure 6.13.: A benchmark chart of the widgets processing time and response-time.

uses JavaScript webworker (already mentioned in section 3.1.3). With their help, threads could be created. But it only worked on the newest **Firefox alpha** (Minefield). The "stable" Firefox 3.6 crashed after several seconds. Other browsers either do not support webworkers or support a version where communication between the workers is done via "strings" and therefor useless (for my purpose). But working with the webworkers could really improve the framework. One point is the computational speed, but on the other hand, all operations of JavaScript which need some time, are blocking the user interface. Common frameworks are working around this problem with really smart timers, but with threads, this problem would be gone.

But there will always be a point, where the received data is too much, to be handled by a browser.

Perhaps JavaScript and browsers are at the time of writing a limitation for *Rich Clients*. In a real emergency situation, it's not acceptable, if a script suddenly stops working. Browser, dealing with large datasets in JavaScript, lack of robustness and are

not scalable. The data must be limited somehow, or datasets must be pre-processed by another server component.

7. Conclusion

During the last years, different solutions have been presented to access the *OGC Sensor Observation Service*. In this work, a framework has been developed to create web-based client applications which are capable of OGC-compliant query processing to sensor observation services. Especially the support for mobile sensors creates new options for rich visualization of their data. The OpenLayers integration to the framework complies the requirement of basic GIS functionalities with sensor mapping and track visualization. In combination with the chart tool *flot* different techniques have been developed to illustrate the received data. Using the different *rangeFilters* on data, recorded by mobile and/or stationary sensors, a fast analysis of actual situations is possible. Especially in the context of the ESS project, a central operator must be able to get a fast overview in a case of a emergency situation. In addition the example of flickr data shows that non classic sensor data could also be visualized with the same techniques.

But, to use a web-based AJAX client for an *Emergency Support System*, the data must be limited somehow. There will always be the point, when the browser stops working caused by too much data.

8. Future work

Different points where improvements or future work is possible can be deduced from the results. On the one hand, there is the ESS SOS interface, that needs support for the enhanced profile of the OGC SOS standard and different other issues which were not considered yet. On the other hand, there is the framework that could be extended to support other data sources. On that way, it would be much more flexible. An additional benefit could be generated, by storing the filtered data on any way. Either the *rangeFilters* could be saved for a specific session, or the filtered observations itself could be sent to server, possibly a *Web Feature Service*. Of course, there are also unused possibilities for visualization. For instance, a real gradient to map measurements to a track. To reduce the needed computational power, new JavaScript techniques like Web-Worker could be included to the framework. The new direct2d API of Microsoft offers real hardware acceleration through graphic cards for browsers on modern Windows systems. Thus, the next browser generation will be able to process much more complex algorithms.

List of Figures

Bibliography

[1] Rigger, "Interdisziplinre referenzimplementierung eines drahtlosen sensornetzwerkes zur erfassung von messdaten und deren simultanen speicherung, auswertung und visualisierung in einem webgis", http://www.terrestris.de/wp-media/downloads/Diplomarbeit_Riegger_2006.pdf, 2010.

[2] Darren James Dave Crane, Eric Pascarello, *Ajax in Action: Das Entwicklerbuch fr das Web 2.0*, ADDISON WESLEY, 2006.

[3] N. Bartelme, *Geoinformatik: Modelle, Strukturen, Funktionen*, Springer, 1998.

[4] Tyler Mitchell, *Web Mapping mit Open Source GIS-Tools*, O'Reilly, 2008.

[5] Open Source Geospatial Foundation, "Openlayers homepage", http://openlayers.org/.

[6] Christina BIAKOWSKI Till ADAMS, "Praxishandbuch webgis mit freier software umn mapserver postgresql/postgis avein! mapbender", *Ein Gemeinschaftsprojekt von CCGIS und terrestris*, 2006.

[7] OGC, "Open geospatial consortium (ogc) about", http://www.opengeospatial.org/ogc, 2010.

[8] OGC WMS, "Open geospatial consortium (ogc) wms", http://www.opengeospatial.org/standards/wms, 2010.

[9] OGC WFS, "Open geospatial consortium (ogc) wfs", http://www.opengeospatial.org/standards/wfs, 2010.

[10] OGC SOS, "Open geospatial consortium (ogc) sos", http://www.opengeospatial.org/standards/sos, 2010.

[11] Scott Raymond, *Ajax on Rails*, O'Reilly, 2007.

[12] Peter Klicman Douglas Crockford, *Das beste an Javascript*, O'Reilly, 2008.

[13] crockford, "Crockford webpage", http://javascript.crockford.com/, 2010.

[14] 52North, "52 north ox-framework", http://52north.org/index.php?option=com_content&view=category&layout=blog&id=30&Itemid=45#availableApps, 2010.

[15] OOSTethys, "Oostethys the ogc oceans ie", http://www.oostethys.org/System%20Architecture, 2010.

[16] MapServer, "Mapserver hompage", http://mapserver.org/ogc/sos_server.html, 2010.

[17] deegree, "deegree hompage", http://www.deegree.org/, 2010.

[18] Dominik Helle Till Adams, "Fossgis conference", http://www.fossgis.de/konferenz/wiki/OpenLayers_trifft_Sensor_Observation_Service_%28SOS%29, 2010.

[19] OGC SML, "Open geospatial consortium (ogc) sensorml", http://www.opengeospatial.org/standards/sensorml, 2010.

[20] Peter Kreuel, "Gut sortiert (javascript-entwicklung mit jquery)", Linux Magazine, 2010.

[21] iola, "flot hompage (jquery plugin)", http://code.google.com/p/flot/, 2010.

[22] John Resig, "John resig blog", http://ejohn.org/blog/, 2010.

A. Appendix

A.1. JavaScript Code

```
/**
 * Author: John Resig
 */

// Inspired by base2 and Prototype
(function() {
    var initializing = false;
    var fnTest = /xyz/.test(function() {
        xyz;
    }) ? /\b_super\b/ : /.*/;

    // The base Class implementation (does nothing)
    this.Class = function() {
    };

    // Create a new Class that inherits from this class
    Class.extend = function(prop) {
        var _super = this.prototype;

        // Instantiate a base class (but only create the instance,
        // don't run the init constructor)
        initializing = true;
        var prototype = new this();
        initializing = false;

        // Copy the properties over onto the new prototype
        for ( var name in prop) {
            // Check if we're overwriting an existing function
            prototype[name] = typeof prop[name] == "function"
                    && typeof _super[name] == "function"
                    && fnTest.test(prop[name]) ? (function(name, fn) {
                return function() {
                    var tmp = this._super;

```

```
35                      // Add a new ._super() method that is the same method
36                      // but on the super-class
37                      this._super = _super[name];
38
39                      // The method only need to be bound temporarily, so we
40                      // remove it when we're done executing
41                      var ret = fn.apply(this, arguments);
42                      this._super = tmp;
43
44                      return ret;
45                  };
46              })(name, prop[name])
47                      : prop[name];
48          }
49
50          // The dummy class constructor
51          function Class() {
52              // All construction is actually done in the init method
53              if (!initializing && this.init)
54                  this.init.apply(this, arguments);
55          }
56
57          // Populate our constructed prototype object
58          Class.prototype = prototype;
59
60          // Enforce the constructor to be what we expect
61          Class.constructor = Class;
62
63          // And make this class extendable
64          Class.extend = arguments.callee;
65
66          return Class;
67      };
68  })();
```

Listing A.1: John Resig's inheritance - taken from [22]

A.1.1. ESS SOS Interface

```
1  var SOSUrl = 'http://v-swe.uni-muenster.de:8080/WeatherSOS/';
2  // Create a new interface to the SOS server
3  var UniMuensterSOS = new SOSInterface(SOSUrl);
4
5  // Request all available sensors offered by this SOS
6  /*
7   * The parameter is a callback(function) that is being called,
8   *  as soon the response is available.
```

```
 */
UniMuensterSOS.getSensors(function(parm) {
   // For each sensor, all observations are being requested
      for (sensor in parm.sensorList) {
         UniMuensterSOS.getObservations(function(observation) {
            /*
             * This is the callback, that will handle the response
             * This one will only print it to the firebug log
             */
            console.log(observations);
         }, {
            /*
             * The second parameter are the options for each request
             */
            getOfferings : null,
            procedures : [ sensor ],
            observedProperties : null,
            filter : standardfilter
         });
      }
   });
```

Listing A.2: A example for the SOS interface with comments

```
    var filter = {
        // Spatial_Capabilities
        featureOfInterest : {
            SpatialOperator : "BBOX",
            GeometryOperand : "gml:Envelope",
            lowerCorner : "38.11 -78.6",
            upperCorner : "38.14 -78.4"
        },
        // Temporal_Capabilities
        eventTime : {
            TemporalOperand : "gml:TimeInstant",
            TemporalOperator : 'TM_Equals',
            beginPosition : "latest",
        }
    };
```

Listing A.3: A fitler example

A.1.2. ESS Rich Internet-Client Framework

```
sosgui.WidgetName = sosgui.extend(new function() {

   var myself, mapControls, map;
```

```
4     // This is the constructor. It will be called, if a new instance of this
5        // class was generated
6        this.init = function(_map, _hook) {
7           /*
8            * Very important to bind the keyword this to myself in the init
9            * function. The "new function" construct will return the function
10           * to the extend method from class, where all super class properties
11           * are copied to this class. After that happened, the init method
12           * will be called. If the keyword this, is used before in here, the
13           * properties are not set correctly
14           */
15          myself = this;
16          map = _map;
17          hook = _hook;
18       };
19       // This is a public function
20       this.public_Function = function() {
21          /*
22           * Offer some methods for application building.
23           */
24       };
25
26       // This is a private function
27       var private_Function = function() {
28          /*
29           * Create some HTML elements and apply them to the hook element.
30           */
31
32          // Using the event-system:
33          myself.addListener("specific event", callback);
34          myself.fireToListener("specific event", parameter);
35
36          // Using the SOS manager
37          SOSInterfaceObject = myself.getSOSInterface('URLtoSOS');
38       };
39    });
```

Listing A.4: An empty template for a new widget

```
1  (function() {
2     /*
3      * Create a new object of a specific widget. This is normal Java known
4      * notation. Based on the widgets, they need a DOM element as parameter, to
5      * append their HTML elements and/or the OpenLayers Map object.
6      */
7     var widgetObject = new sosgui.WidgetName(parameters);
8     /*
9      * Additionally extra methods can be called, if necessary
10     */
11    widgetObject.method();
```

```
12 });
```

Listing A.5: How to instantiate a widget for an application