SMARTRESOURCE PLATFORM FOR WEB SERVICE INTERACTIONS' SEMANTIC LOG

DELIVERABLE 3.3

University of Jyväskylä, Agora Center
October 2006

Author: Industrial Ontologies Group

Contact Information*:* e-mail: vagan@it.jyu.fi

Title: SmartResource Platform for Web Service Interactions' Semantic Log

Work: Technical report

Number of Pages: 18

Abbreviations

SOAP - Simple Object Access Protocol

XML - eXtensible Markup Language

RDBMS - Relational Database Management System

OWL - Ontology Web Language

RDF – Resource Description Framework

AJAX - Asynchronous JavaScript and XML

HTTP - Hypertext Transfer Protocol

Contents

1 Introduction

In a heavily computerized industrial environment a lot of information flows contain data, which is not stored anywhere and is just passed from one node to another. After the information was processed by the receiving node it, may be deleted. However, industry nowadays is seeking for storage mechanisms, which would allow flexible analytical processing of such history data. The reason for storing this data is rather trivial – companies tend to receive more analytical data and increase the level of maintenance processes automation by applying learning algorithms and running intelligent decision support services. But this is not the only reason why companies are looking for more flexible data storage mechanisms than just file-based storages or even relational databases. Another add-value, which companies expect from new storages, is extensibility and integral view on the data. Linking the contents of the storage with all possibly needed relationships, gives industries an ability to extract data from different perspectives and thus optimize their actions and optimize expenses.

2 Problem description

METSO Automation [METSO] provides maintenance and technical support services to its clients all over the world. There is a VPN connection established between customer sites and METSO Automation maintenance center. Customers send fault messages in a SOAP/XML format to the maintenance center, where message data is analyzed by experts and may or may not be stored in a file system. Fault messages contain data which is potentially useful for fault analysis and predictive maintenance. To make it easily and flexibly processable, the maintenance center has to install a storage system of a new generation, which is easy to maintain, extend and query.

3 Solution Description

Based on the problem description, we have selected a semantic web-based approach to storage and retrieval of log data. Metso Automation is a big enterprise which may require a more complex integration solution in the future. Having a lot of customers and providing a large variety of services, METSO Automation will benefit from an integral (pseudo-)centralized storage of its business data including but not limited to maintenance activities. We have selected a Semantic Web technology for implementation because it has a set of distinctive features which enforce data representation. Here we can name at least graph-based data representation, simple, but effective class-subclass and property-subproperty relationships, bringing add values, which are hard to implement using RDBMS [RDBMS]. These features are naturally supported by most of the semantic storages and may be extended using third-party inference engines.

3.1 Architecture of the system

The system presented here comprises a set of components, which serve as an interface for SOAP messages handling and transformation, interaction with semantic storage and storage browser (see Figure 1).

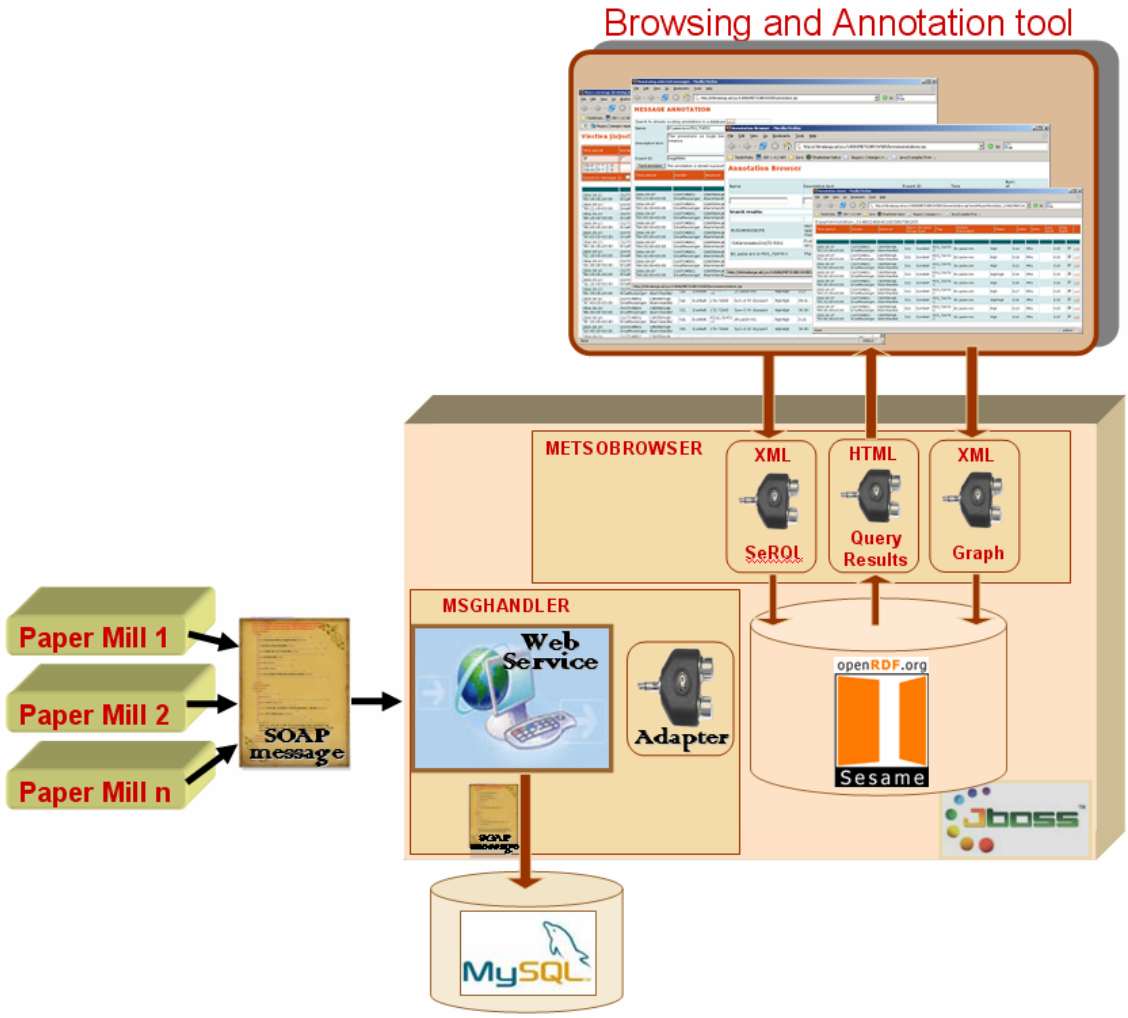

Figure 1 – System Architecture

The system can be divided into two main subcomponents – *Message Handler* and *Message Browser*. *Message Handler* receives and processes SOAP messages from different customers. *Message Browser* provides functionality for message browsing, filtering and annotation. Both subcomponents run on a JBoss [JBOSS] application server and are independent from each other.

3.2 Domain ontology

The ontology plays a role of a schema for all data within the storage. The domain analysis, which goes in parallel with requirements analysis, allows experts to distinguish main concepts and link them with the needed relationships (properties in OWL [OWL] and RDF [RDF] terminology). For ontological development we used a Protégé [Protégé] tool. The ontology elaboration was mainly based on a SOAP messages content analysis. Together with experts from our industrial partner we have made a thorough analysis: We have separated main concepts and have defined allowed ranges for properties' values.
The main concept of the ontology is *Message* (see Figure 2).

Message		
messageUID		String
securityLevel		Integer
time		String
hash		String
hasAlarm	Instance*	Alarm
receiverGroup	Instance*	ReceiverGroup
hasMsgType	Instance	MessageType
messageSender	Instance	Messenger
messageReceiver	Instance	MaintenanceCenter

Figure 2 – Message concept of the domain ontology

The *Message* class describes such message properties as message sender and receiver, message reception time, etc. and refers to *Alarm* class (see Figure 3), which contains information about the reason of message generation in a form of fault data like measurements of sensors, status data and exact module of the production line, where the alarm happened.

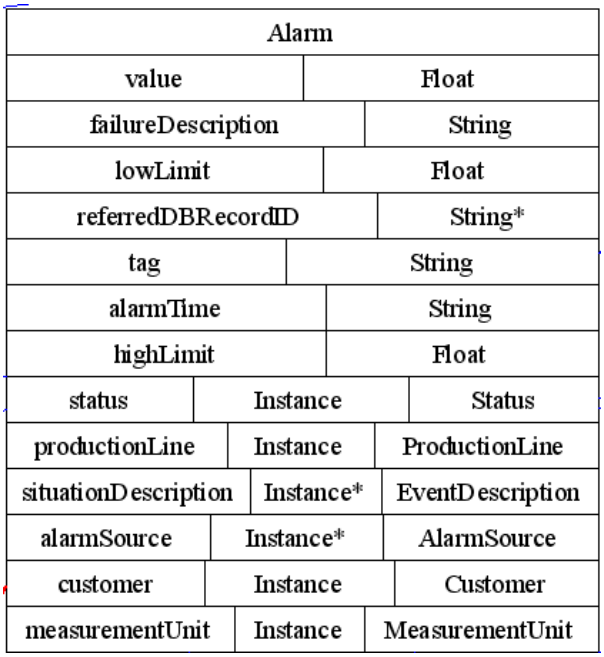

Figure 3 – Alarm class of the domain ontology

We will not go through all the classes of the ontology, because most of them are obvious and do not require explicit explanation. The last class we will mention here is *ExpertAnnotation* class (see Figure 4), which describes the annotation made by expert in a message browsing and annotation tool.

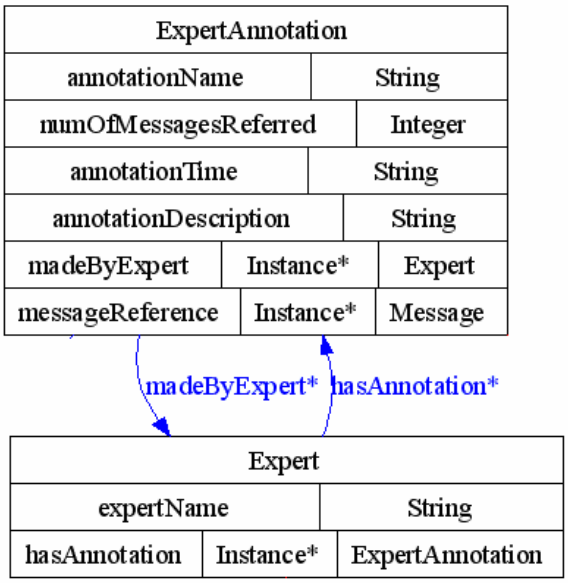

Figure 4 – ExpertAnnotattion and Expert classes of the domain ontology

MadeByExpert and *hasAnnotation* are inverse properties. It means that if an expert refers to some *ExpertAnnoatation*, then appropriate property *madeByExpert* is generated.

3.3 Message Handler

Message Handler (MSGHANDLER module on an application server) is a component which incorporates message receiving and adaptation functionality. The main entry point of the

component is a *MessageReceiver* class wrapped as a web service with name *MessageService* and method *process()*. The service has *message* style, which is stated in a service deployment descriptor file *server-config.wsdd*.

```
<service name="MessageService" style="message">
    <parameter name="className"
    value="org.smartresource.app.metsocase.impl.MessageReceiver" />
    <parameter name="allowedMethods" value="process" />
</service>
```

MessageReceiver class acts as a wrapper and redirects all calls directly to *MessageProcessor* class (see Figure 5). This wrapping is done to simplify the integration with the agent platform.

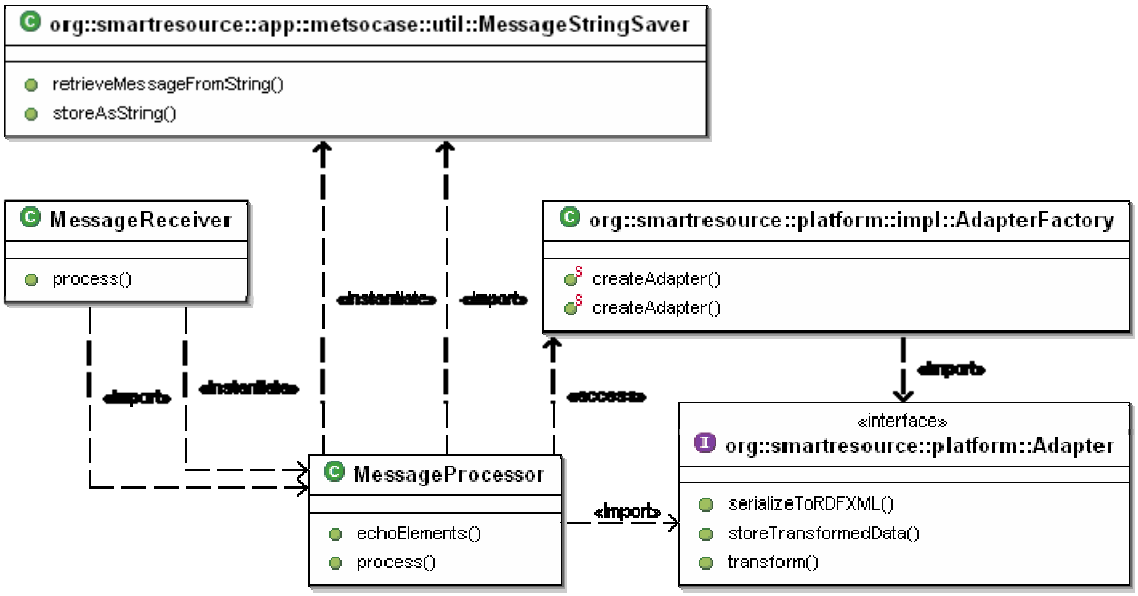

Figure 5 – Class diagram of a *Message Handler*

In *MessageProcessor* the *process()* method instantiates the *Adapter* via *AdapterFactory* and invokes adapter methods for transformation and storage of the incoming SOAP/XML messages.

3.4 Platform Adapter

Adapter is a key enabling element of the platform. It transforms data from one format to another and usually stores transformed data to the ontology storage. In *Message Handler* component the adaptation is done from two XML fragments, which are extracted from SOAP Header and SOAP Body elements, to an RDF graph, compliant with the ontology. The class diagram of the *Message Handler* adapter is shown on Figure 6.

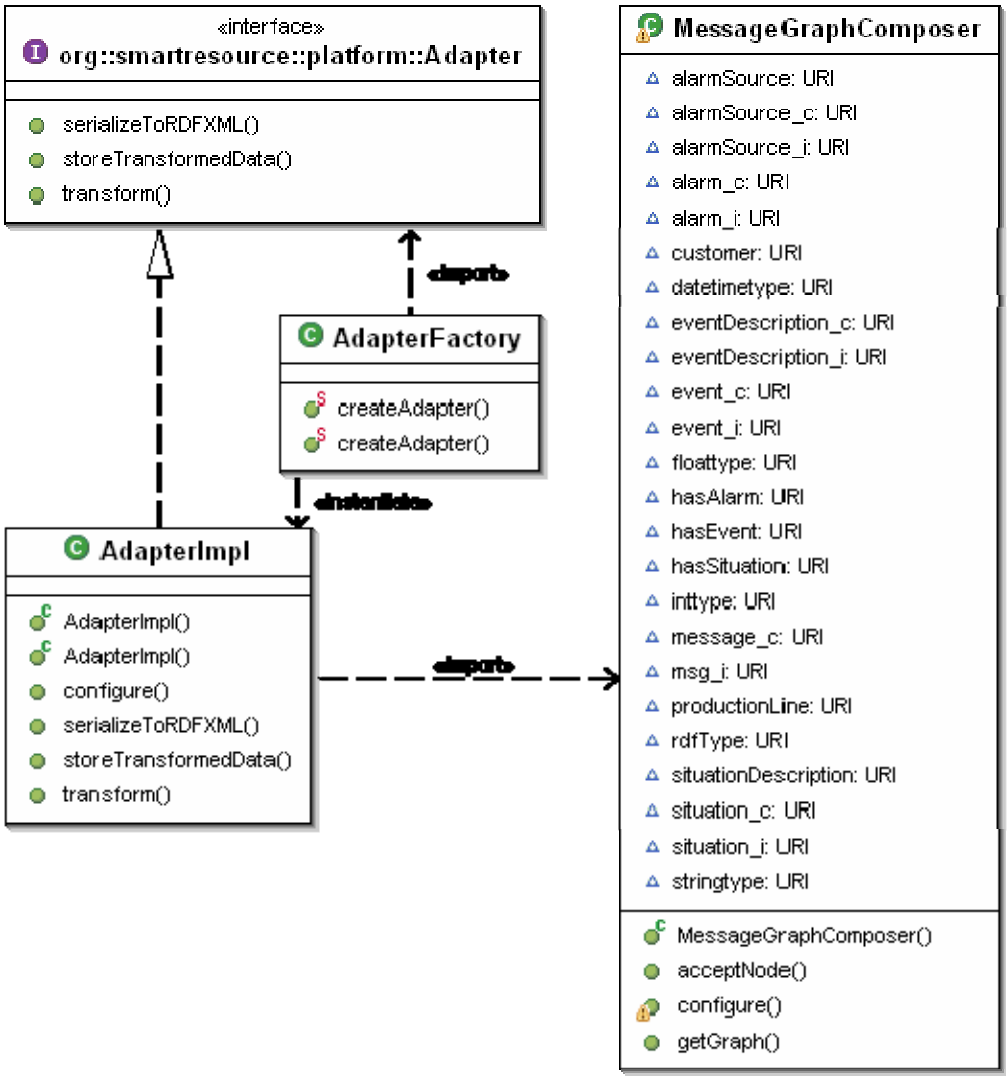

Figure 6 –Adapter class diagram

MessageGraphComposer class produces an RDF graph data from given XML elements. In this implementation method *transform()* of the *AdapterImpl* class sequentially passes XML elements to the *MessageGraphComposer*, which in turn adds statements to an RDF graph. When all messages are processed, Adapter calls *getGraph()* method and receives a Graph reference, which is passed to the object which has instantiated the Adapter and called *transform()* method. After a Graph reference is obtained, the Graph data can be passed to the *storeTransformedData()* method of the Adapter, and consequently will be stored to the storage.

3.5 Message Browser (a user guide for a GUI tool)

The start window of the application is shown in Figure 7. The user selects parameters for filtering by checking the appropriate checkbox. To execute a filtering query, user presses button "*Query*" and receives a table of messages. The user has a possibility to see the original SOAP/XML message by clicking the link in a last column of the table.

Figure 7 – a Message Filtering Window

When user decides to annotate a set of messages, he/she marks relevant message strings in a checkboxes in a previous to last column of the message table and presses "*Annotate selected messages*" button. A message annotation interface appears in a new window with messages selected on a previous step (see Figure 8).

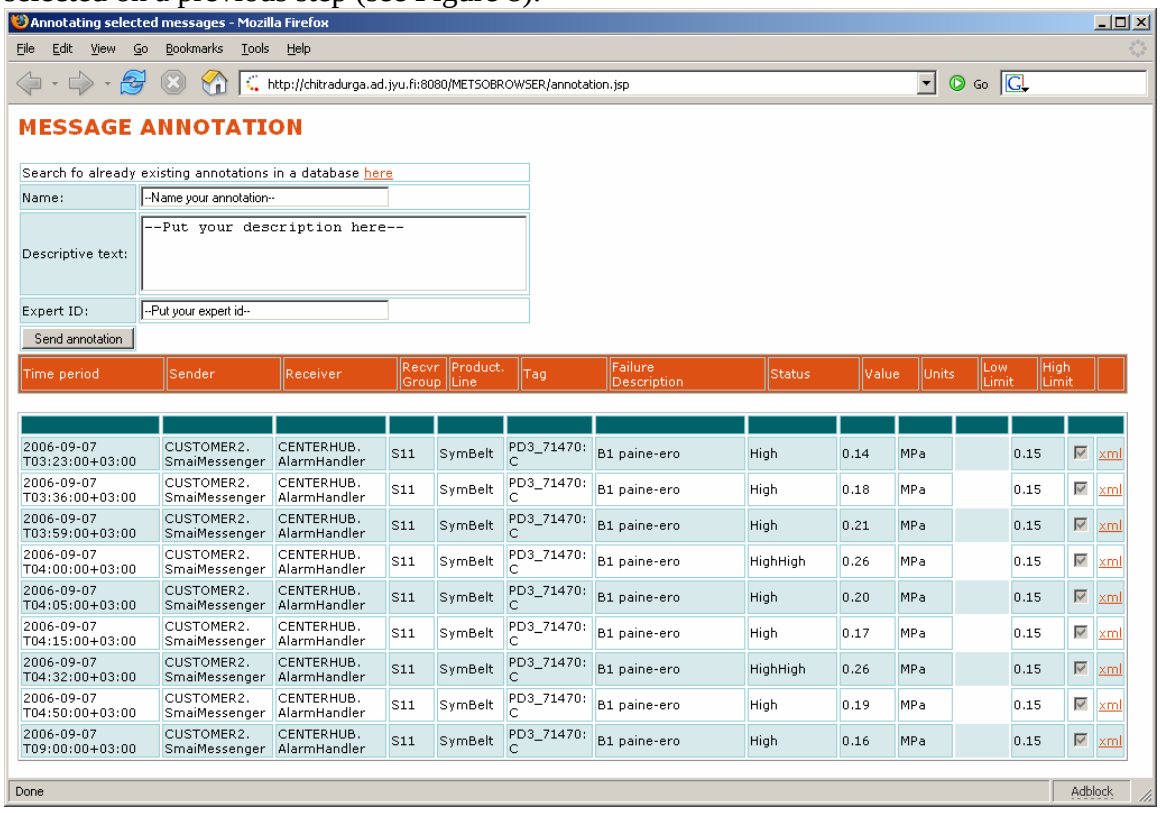

Figure 8 – Message Annotation window

The user is prompted to type name and description of a new annotation. Next, he/she fills Expert ID field with his/her unique ID and presses the button "*Send Annotation*". The information that annotation was stored/not stored appears right next to the button.

User can also browse annotations by clicking on the link on the top of the *Message Annotation* window described above, or by typing in a browser "http://host:port/METSOBROWSER/browseannotations.jsp".
By pressing button "*Query*" in *Annotation Browser* window (see Figure 9) user receives a list of annotations in a table form. The filtering functionality is planned but not yet implemented.

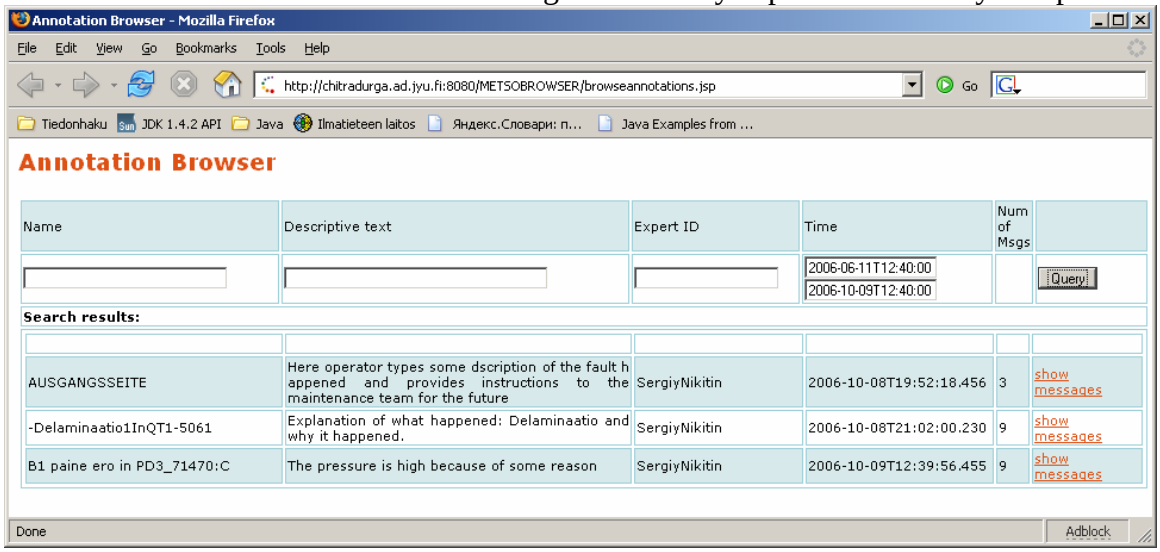

Figure 9 – Annotation Browser window

By clicking on a "*show messages*" link in the rightmost column of the table of annotations in *Annotation Browser*, a window with a set of messages of the selected annotation appears (see Figure 10).

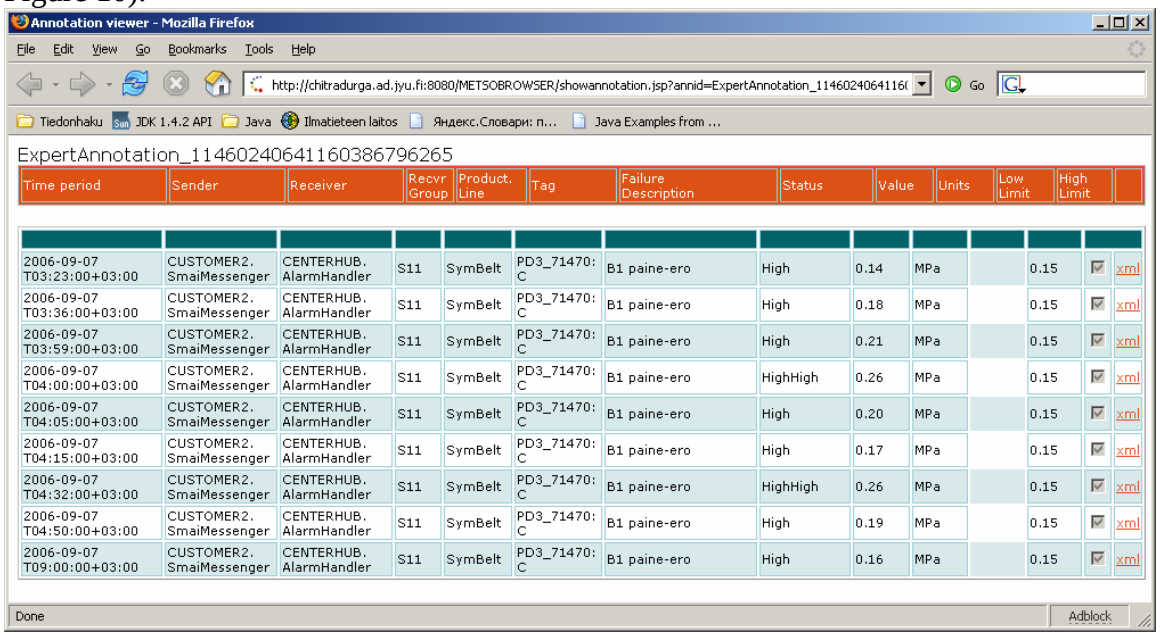

Figure 10 – Messages of the selected annotation

3.6 Message Browser Class diagram

The class diagram on Figure 11 depicts main classes of the *METSOBROWSER* component. *ActionController* is a main control servlet of the application. On a browser side a JavaScript [JS] AJAX [AJAX] function sends HTTP [HTTP] requests to the *ActionController* servlet.

The request contains XML string which is parsed and depending on a root element, is further processed by different handlers in a *doPost()* servlet method.

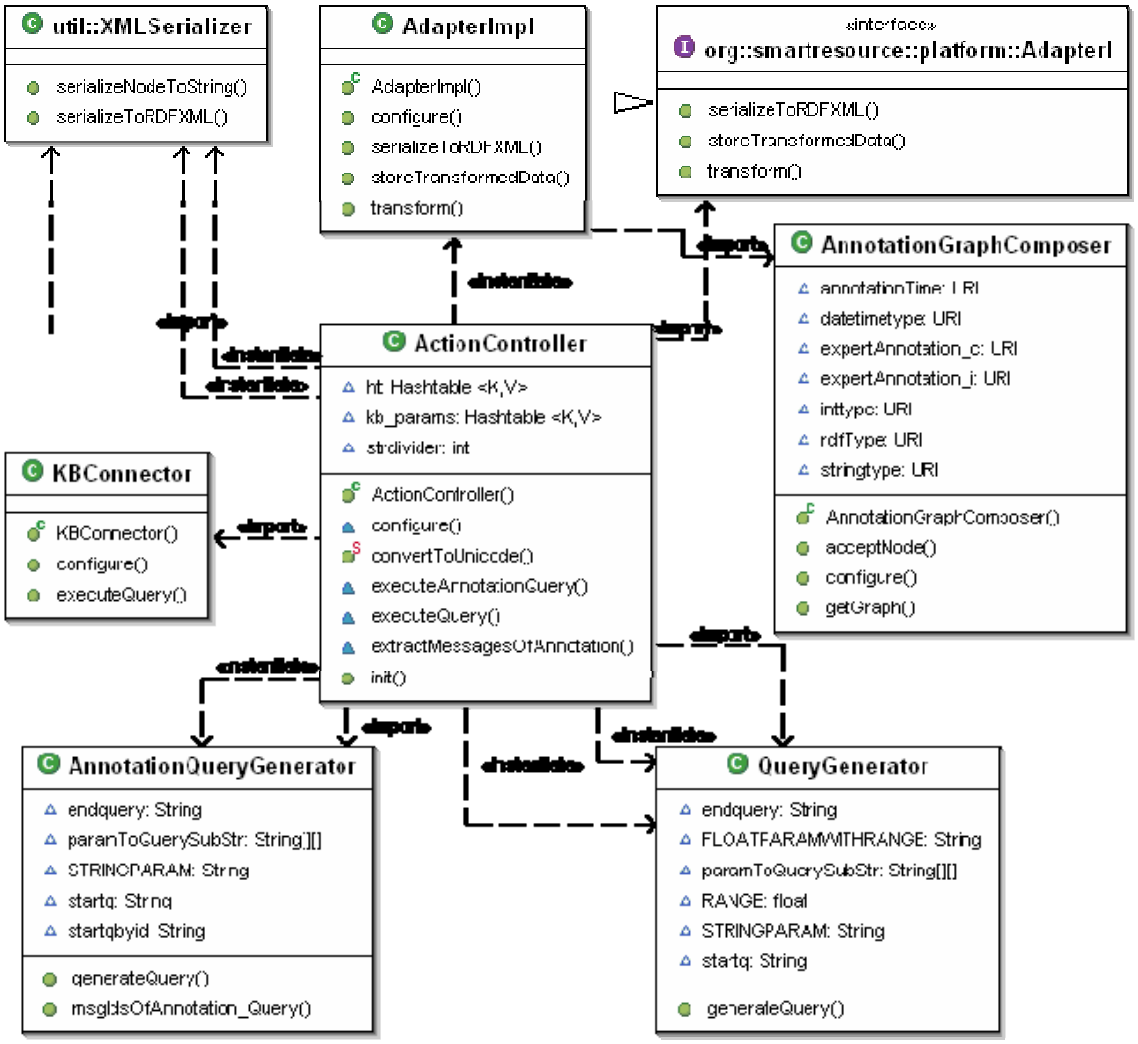

Figure 11 – Class diagram of the METSOBROWSER

There are two query generators which dynamically compose a SeRQL query [SeRQL] based on an XML query received from the browser. *QueryGenerator* class produces a SeRQL query for messages extraction based on the XML query received from a user. *AnnotationQueryGenerator* class generates two types of queries – first one is produced by *generateQuery()* method and provides query for extraction of all available expert (user) annotations. Second query is produced by *msgIdsOfAnnotation_Query()* method and provides a query for extracting messages of one particular annotation. *ActionController* class also instantiates an *AdapterImpl* class which is used for transformation of the annotation received from the browser interface in an XML format to the RDF graph using *AnnotationGraphComposer* class. *KBConnector* class is used to obtain a connection to the storage instance and execute queries.

3.7 Dynamic query generation

When a user selects filtering parameters and presses "*Query*" button in a message browser, the script in a browser generates an XML document of a format like:

```
<query>
  <messageSender>value1</messageSender>
  <status>value2</status>
  …
</query>
```

The document is then passed as a parameter of the HTTP request to *ActionController* servlet, where the XML document is processed depending on a root node. All <query> nodes are meant to be filtering queries for fault messages. The node is passed to a *QueryGenereator*, which processes child nodes of the document and combines a query from a predefined header, a dynamic part and an enclosing tail. The header part is static and looks like:

```
SELECT time, messengername, maintcentername, msgtypeid, recgroupname,
productionlinename, tag, failuredesc, statusname, value, lowlimit,
highlimit, message
FROM
{message} ns:time {time}
{message} ns:hasMsgType {} ns:msgId {msgtypeid},
{message} ns:messageSender {} ns:messengerName {messengername},
{message} ns:messageReceiver {} ns:maintCenterName {maintcentername},
{message} ns:receiverGroup {} ns:receiverGroupName {recgroupname},
{message} ns:hasAlarm {alarm},
{alarm} ns:failureDescription  {failuredesc};
        ns:value {value};
        [ns:tag {tag}];
        [ns:lowLimit {lowlimit}];
        [ns:highLimit {highlimit}];
        ns:productionLine {} ns:productionLineName {productionlinename},
{alarm} ns:status {} ns:statusName {statusname}
```

The dynamic part has an algorithm of a composition depending on the XML document content. Referring to the XML, given above, a SeRQL query part would look like:

```
WHERE messengername LIKE “value1*” AND statusname LIKE “value2”
```

The star (*) sign in a query means that a search criterion for a given variable must start with the given string (value1) but the end of the string may vary.
The query string is appended by the enclosing tail which contains definition of the namespace:

```
USING NAMESPACE
ns=<http://org.smartresource.app.metsocase#>
```

3.8 Integration with SmartResource Agent platform

Agent-based extension of the system introduces new possibilities in communication with the human. In the example shown on Figure 12, the expert has a real-time monitoring tool which is updated by the Metso Expert Agent and provides an online fault data bound to geographical location.

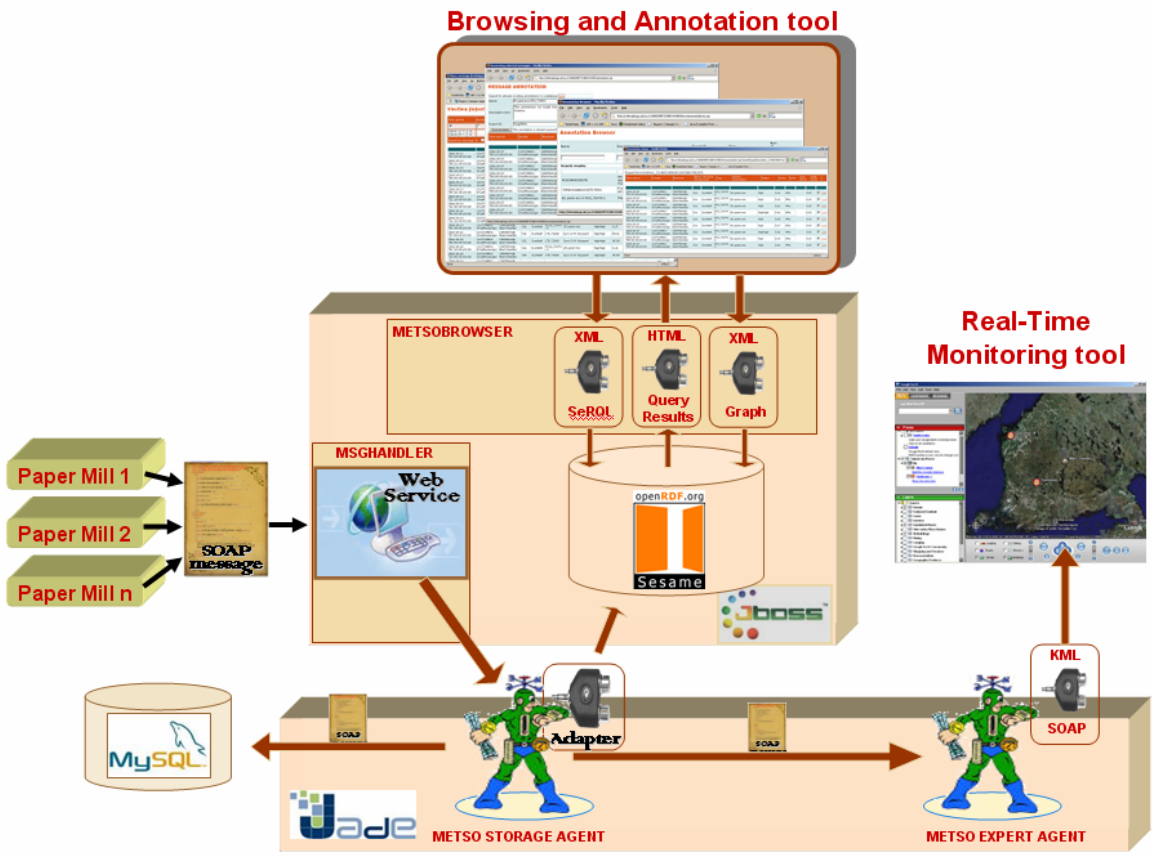

Figure 12 – Architecture of the system enhanced with the SmartResource Agent platform

The main architectural change is that SOAP message from the customer goes via web service to *METSO Storage Agent* and the transformation is performed on the Agent side. Further, the agent executes the same operations as *Message Handler* but in addition, sends a message to *METSO Expert Agent* with the fault information. *METSO Expert Agent* dynamically updates the situation on the Real-Time monitoring tool by adding a new object on the map. An expert can view the fault message by clicking on the link provided in a context menu of the newly created object. The link opens a message browser in a new window with the particular message selected.

3.9 System Configurability

The configurability element was introduced to the system on the later stage and continues to evolve. The idea behind was to simplify deployment of the system and make system parts adjustable (tunable) in a runtime. There is an interface named *Configurable* which must be implemented by all components in order to allow runtime configuration of their parameters. The interface has one method *configure(Hashtable ht)*, which accepts a *Hashtable* object with the configuration parameters and their values. The set of configurable parameters is defined by every object. When object is instantiated, it inevitably calls *configure()* method and passes either *null* as an argument, which means that object must be configured by default, or a *Hashtable* with new configuration settings. An object can be configured later in runtime as well. At this point we did not provide a security policy for configurability of objects yet but the research towards different configurability aspects continues to evolve.

3.10 Performance and Scalability

The software we have used in the implementation of the system has shown a stable and predictive behaviour. The Sesame storage demonstrates very promising performance, although we had to use in-memory storage mechanism instead of database backend because of CPU-time consuming operations in between Sesame and MySQL database. Up to day the storage contains over 350 messages and extraction time of all the messages in a table format by sesame querying tool takes around 600 milliseconds. The hardware we use is pretty basic – it is a Pentium III 800 MHz with 512 Mb of RAM. However, due to non-optimal server-side implementation of the browsing tool, we have some delays, when the number of extracted messages is big. But it is rather a problem of a particular pilot implementation, than of architecture as such.

4 Conclusions and Future Work

The system presented here demonstrates the applicability of Semantic Web technology on a real industrial case. The system performs transformation from XML formats to RDF and provides browsing and annotation facilities for stored data. The system development process has shown that it is possible to combine different Semantic Web tools in a different modeling and execution tasks such as ontology modeling in the Protégé tool and ontology insertion into the Sesame storage. It is pretty simple to add new classes or properties in a Protégé tool and then just copy-paste the updated model in RDF format to Sesame without any restarts. The model is updated in a couple of seconds while the system continues to run. We can say that ontology-based approach is extensible, although we did not test what happens when we make major changes to the ontological model such as changing domain and range of properties or deleting classes. Adapters are the most sensitive components to changes in the structure of the incoming formats, messages and ontology. In our opinion the adapter transformation function should be tied together with the ontology and react immediately on changes which lead to inconsistency of the data. We see one of the future research challenges in an elaboration of a user interface development process for ontology-based applications. In the implementation of a browsing tool we have extensively used AJAX technology and XML-based messaging. We have realized a need for a script-based engine-like client side visualization library for ontology based applications.

5 References

[AJAX] AJAX technology http://en.wikipedia.org/wiki/AJAX

[HTTP] Hypertext Transfer Protocol http://www.w3.org/Protocols/

[JBOSS] A JBoss application server http://www.jboss.com/

[JS] JavaScript – a scripting programming language http://en.wikipedia.org/wiki/JavaScript

[METSO] Metso Automation – a leading provider of software and maintenance solutions in paper industry

[OWL] Ontology Web Language, a W3C Recommendation on 10 February 2004, http://www.w3.org/TR/owl-semantics/

[Protégé] Protégé – a free, open source ontology editor, http://protege.stanford.edu/

[RDBMS] Relational Database Management System http://en.wikipedia.org/wiki/Relational_database_management_system

[RDF] Resource Description Framework, a W3C Recommendation on 10 February 2004, http://www.w3.org/RDF/

[SOAP] Simple Object Access Protocol, W3C Recommendation 24 June 2003, http://www.w3.org/TR/soap12/

[Sesame] Sesame – an open source RDF database with support for RDF Schema inferencing and querying. http://www.openrdf.org/

[SW] Berners-Lee, T., Hendler, J., and Lassila, O. (2001) The Semantic Web, *Scientific American*, Vol. 284, No. 5, pp. 34-43.