University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor Thesis

Semantic Web in EEG/ERP Portal

Pilsen, 2012

Jakub Krauz

Declaration

I hereby declare that this bachelor thesis is completely my own work and that I used only the cited sources.

Pilsen, May 5, 2012, Jakub Krauz

Acknowledgement

I would like to thank to my thesis supervisor Ing. Petr Ježek and to Ing. Roman Mouček, Ph.D. for their assistance, time and valuable remarks.

Abstract

This thesis is focused on the usage of Semantic Web technologies, especially the Web Ontology Language, within the neuroscience research. It deals with concepts of mapping between object-oriented programming and Semantic Web languages. The goal is to propose and implement a transformation tool for automation of this process. Solving of semantic gaps between object-oriented programming and Semantic Web technologies is investigated and discussed. This effort resulted in the development of an extension of current object code based on Java annotations. The transformation of these annotations is proposed and implemented as well. The tool is used in the EEG/ERP Portal, which manages a database of EEG/ERP experiments. An integration of this tool into the EEG/ERP Portal is also presented.

Contents

1 Introduction **2**

2 Semantic Web **3**

2.1 Resource Description Framework (RDF) ... 4
2.2 RDF Schema (RDFS) ... 7
2.3 Web Ontology Language (OWL) ... 8

3 EEG/ERP Research **11**

3.1 EEG/ERP Experiments ... 11
3.2 The EEG/ERP Portal ... 12

4 OOP to OWL Mapping Concepts **14**

4.1 OOM Compared with OWL ... 15
4.2 Jena ... 16
4.3 The OWL API ... 17
4.4 JenaBean ... 17

5 JenaBeanExtension **18**

5.1 Implemented Mapping ... 19
5.2 Java Annotations ... 21
5.3 Implemented OWL Language Elements ... 22
5.4 Serialization Syntaxes ... 25
5.5 Provided API ... 26

6 Integration into the Portal **28**

6.1 Getting the Ontology Document ... 28
6.2 Jena Models Difficulties ... 29
6.3 RDF/XML Serialization Syntax ... 31
6.4 Optimization of the Transformation Process ... 31
6.5 Invalid Characters in XML ... 32
6.6 Proxy Objects in the OOM ... 33

7 Conclusion **34**

1 Introduction

The neuroscience research has made great progress in last decades thanks to technical advance. Electroencephalography (EEG) enables researching the brain activity and disorders. The derived technique (event-related potentials, ERP) is used in the research of brain responses to various stimuli. A need to share knowledge arose with the growth of the EEG/ERP research. Various Web portals emerged in order to enable researchers to share their knowledge, data from experiments and other information. One of possible ways of sharing information is represented by the Semantic Web technologies.

The goal of this work is to propose and implement a transformation tool for the EEG/ERP Portal, which manages data from EEG/ERP experiments. The transformation tool is expected to provide gathered data using languages of the Semantic Web, namely the Web Ontology Language (OWL). Because of existing semantic gaps between an object-oriented code (used in the Portal) and an OWL ontology it is desirable to enrich the object code with additional information. Java annotations were chosen for this purposes.

The first two sections deal with the theoretical background. The Semantic Web is introduced and its languages essential for this thesis are described in section 2. Basic concepts of the Resource Description Framework, as the base of all Semantic Web technologies, are discussed and some illustrative examples are given. Section 3 deals with the neuroscience background. It briefly outlines EEG/ERP experiments and describes the purpose and goals of the EEG/ERP Portal.

The next part is focused on the proposal and implementation of the transformation library, which provides an automated mapping from the object-oriented code to Semantic Web languages. Basic mapping concepts as well as existing tools are introduced and discussed in section 4. Section 5 provides description of the developed tool. Implemented mapping and Java annotations as well as provided API are presented. Section 6 describes an integration of the tool into the EEG/ERP Portal, discussing difficulties that arose and describing their solution.

2 Semantic Web

Semantic Web is a term that refers to World Wide Web Consortium's (W3C)[1] efforts to standardize structure of information. Its goal is to change the current unstructured web into the web of structured data. Its advantage consists in sharing information across different applications since data can be processed automatically by machines. More information can be found in [1] and [2].

Today's World Wide Web (hereinafter the Web) became part of our daily life. It is used for work, business, education, free time activities, searching information or simply browsing, and many others. Thanks to its success the Web contains a huge amount of data comprising all thinkable domains. These data can be transferred and accessed via various transfer formats. They are human-readable, but they are not machine-processable in the sense of understanding their meaning, so called semantics.

The problem is that the current Web contains too little information about data structure, so called metadata. All information is intended for a human reader who understands the meaning. But machines do not recognize which piece of information is related with another one, they cannot distinguish relevant linkage among various data sources. Searching for information is therefore very difficult. The text search gives very inaccurate or even unrelated results.

These reasons led to W3C's Semantic Web activity. The beginnings of this effort go back to the early 1990s. The activity resulted in creating the Resource Description Framework (RDF) as the basic standard for describing data structure. Many other similar activities were in motion in the USA and Europe in the same time. This lead to formation of various standards, languages and syntaxes for those purposes. Nowadays all resulting languages, frameworks and syntaxes are integrated and standardized under W3C's Semantic Web.

The vision is to transform the Web of unstructured data into the Web of linked data. Such data will be provided with metadata describing their semantics and linkage. Machines will be able to read and process them automatically, which includes automated reasoning (making logical deductions from given axioms) and querying. The Semantic Web should be accessible as a huge relational database in the outcome. The central importance for these efforts are languages suitable for representing data and their semantics.

1 World Wide Web Consortium (W3C), URL: <http://www.w3.org>

2.1 Resource Description Framework (RDF)

Semantic Web technologies are based on the Resource Description Framework (RDF). It is a model for representing structured information rather than a language. Its original purpose was describing metadata on Web pages. The best source of information about RDF is W3C's RDF Primer [3], also [1] gives a good explanation and many related information can be found on W3C's official site [4].

W3C describes RDF as a language for representing information about resources in the World Wide Web. A resource can be anything – a Web page, an institution, a person etc. These resources do not have to be directly retrieved from the Web, but they have to be unequivocally identified. RDF uses Uniform Resource Identifiers (URIs) for this purpose.

A special subset of URIs are Uniform Resource Locators (URLs). URLs are used on the Web to identify Web pages. Web pages are resources that are directly retrievable from the Web. URIs are generalization of this concept. A URI can be created by anyone who needs to refer some resource. It can look like a URL, but the resource does not have to be accessible from the given address. RDF uses URI references (URIrefs) to be precise. A URIref consists of a URI and a fragment identifier. A URIref can be for example:

```
http://eegdatabase.kiv.zcu.cz/home.html#article5
```

where `http://eegdatabase.kiv.zcu.cz/home.html` is a URI (as well as URL in this case) and `article5` is a fragment identifier.

The mainstay of RDF are so called statements about resources. A statement adds a piece of information about a resource. This concept is based on the idea that things are described via their properties, which have some values. Such a statement consists of three parts (that is why it is also called the triple) – a subject, a predicate, and an object. Subject is the resource being described, predicate is the property and object is the value.

Every part of the triple can be considered a resource on its own. That is why it should be identified by a URIref, so as any statement about this resource can be added if needed. Only those objects that represent concrete values (e.g. textual or numerical) are identified directly by the value itself (so called literals). RDF allows also existence of empty nodes, i. e. nodes that are identified neither by any URIref nor a value.

RDF makes a graph model of those statements (a graph is a set of triples). Both the subject and the object represent nodes and the predicate represents an arc (oriented from the subject to the object). The graph model is the core of RDF. An example of such a graph is shown in Figure 1. It describes a person named Jan Novák, who is 35 years old and has some colleague. Nodes identified by URIrefs (persons) are drawn as ellipses while literals (names, age) are drawn in square nodes.

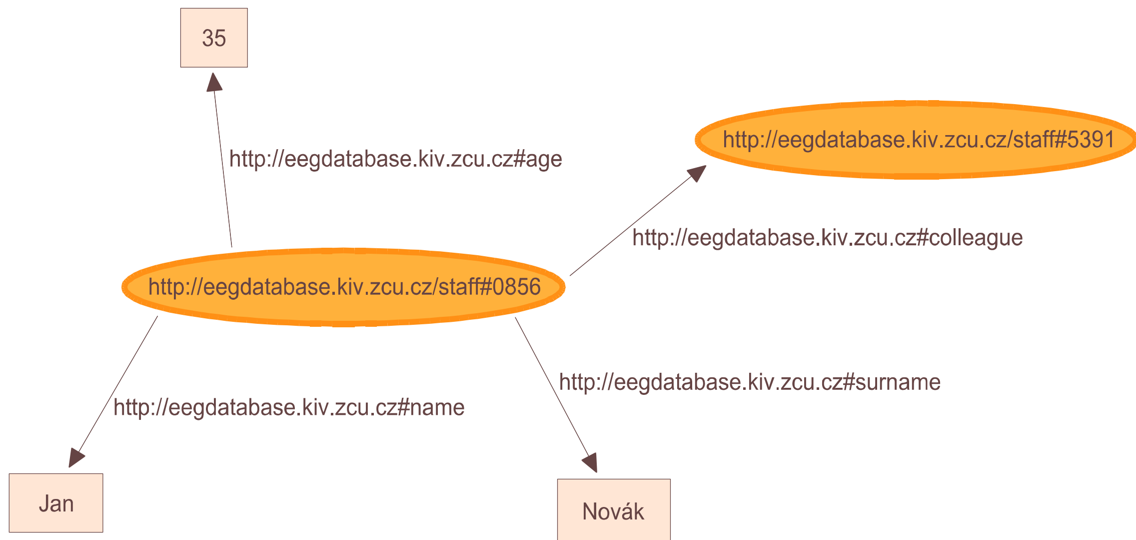

Figure 1: Example of an RDF graph.

An equivalent non-graphical way to express statements is the Triple Notation (N-Triple), which writes out all the triples in a simple text form, each triple ended by a full-stop. RDF uses XML qualified names (QNames) in order to abbreviate long URIrefs. These are comprised of a prefix and a local name. The prefix is a declared shorthand for the URI (called namespace) and the local name corresponds to the fragment identifier from the URIref.

An example of statements in the Triple Notation that expresses the same information as the graph in Figure 1 follows:

```
@prefix staff:   <http://eegdatabase.kiv.zcu.cz/staff#>
@prefix kiv:     <http://eegdatabase.kiv.zcu.cz#>

staff:0856       kiv:name          "Jan"       .
staff:0856       kiv:surname       "Novák"     .
staff:0856       kiv:age           "35"        .
staff:0856       kiv:colleague     staff:5391  .
```

However, the Triple Notation is only an alternative to the drawn graph, but RDF defines a XML-based syntax for writing and exchanging data. This normative syntax is called RDF/XML and is described in [5].

RDF/XML uses common XML features such as namespaces and QNames, datatypes defined by XML Schema [6], entities and others. All tags belonging to RDF have a fixed well-known namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, which is usually abbreviated as `rdf`. The root element is `rdf:RDF`. Every RDF/XML document is a valid XML document, of course.

An example of a RDF/XML document that describes the graph from Figure 1 follows:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY staff "http://eegdatabase.kiv.zcu.cz/staff#">
]>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:kiv="http://eegdatabase.kiv.zcu.cz#" >

    <rdf:Description rdf:about="&staff;0856">
        <kiv:name>Jan</kiv:name>
    </rdf:Description>

    <rdf:Description rdf:about="&staff;0856">
        <kiv:surname>Novák</kiv:surname>
    </rdf:Description>

    <rdf:Description rdf:about="&staff;0856">
        <kiv:age>35</kiv:age>
    </rdf:Description>

    <rdf:Description rdf:about="&staff;0856">
        <kiv:colleague rdf:resource="&staff;5391"/>
    </rdf:Description>

</rdf:RDF>
```

RDF defines many other features like typed literals, blank nodes, containers and collections, statement reification and others. Many useful documents describing RDF and related issues can be found on the official RDF site [4].

RDF is the basic standard for representing information on the Web. Other Semantic Web technologies, languages and vocabularies are built upon this concept. Languages such as RDF Schema (described in 2.2), Web Ontology Language (described in 2.3), or Simple Knowledge Organization System are primarily vocabularies upon RDF. In addition they bring some special features such as alternative serialization syntaxes besides RDF/XML.

2.2 RDF Schema (RDFS)

RDF defines how facts about various resources should be expressed. Expressing some generic knowledge (also called schema knowledge), sorting resources into classes or expressing common properties of those classes is also necessary. RDF defines a vocabulary called RDF Schema (RDFS) for these purposes. RDF Schema specification [7] is a part of W3C's RDF specification. Many useful information is in [1] and RDF Primer [3].

Vocabulary is a set of terms that are used to describe resources. Every term in the vocabulary has a defined meaning. Usually, terms from one vocabulary have a common namespace. A vocabulary consists of classes, properties and their semantics. Classes are used to identify categories of things, while properties are used to describe those things (more in section 4.1).

RDFS is a special kind of generic vocabulary that is used to define other user vocabularies. It contains only generic terms for describing classes, properties and their relations. The namespace of RDFS terms is `http://www.w3.org/2000/01/rdf-schema#`, usually abbreviated by the `rdfs` prefix.

Classes are defined as instances of `rdfs:Class`. Resources can be declared instances of some class using `rdf:type` property. RDF Schema allows creating class hierarchy using inheritance, which can be expressed using the `rdfs:subClassOf` property. RDF Schema defines all classes to be subclasses of the generic class `rdfs:Resource` (because all classes are resources). The following example defines two classes, `Person` and `Man`. `Man` is declared a subclass of `Person`:

```
@prefix kiv: <http://www.kiv.zcu.cz/>

kiv:Person      rdf:type         rdfs:Class  .
kiv:Man         rdf:type         rdfs:Class  .
kiv:Man         rdfs:subClassOf  kiv:Person  .
```

Properties are defined as instances of `rdf:Property`. Properties have domains and ranges. The domain specifies classes of subjects that can own the property and is declared using the `rdfs:domain` property. The range specifies classes instances of which can be values of the property and is declared using the `rdfs:range` property. RDF Schema also allows defining property hierarchy similarly to classes using the `rdfs:subPropertyOf` property. The following example defines two properties, `address` and `permanentAddress`. `PermanentAddress` is declared as a subproperty of `address`, both domain and range are inherited:

```
@prefix kiv: <http://www.kiv.zcu.cz/>

kiv:address            rdf:type    rdf:Property.
kiv:address            rdfs:domain kiv:Person  .
kiv:address            rdfs:range  kiv:Address .

kiv:permanentAddress   rdf:type           rdf:Property .
kiv:permanentAddress   rdfs:subPropertyOf kiv:address  .
```

The RDFS vocabulary provides a number of other properties that can be used in the schema, for example for adding some comments or additional information to declared classes and properties. They can be found in the RDFS specification [7].

RDF Schema is the basic language for defining RDF vocabularies. However, its capabilities are limited. For example RDFS is not able to add constraints on defined terms. Therefore ontology languages such as the DARPA Agent Markup Language (DAML), Ontology Interchange Language (OIL), or the Web Ontology Language (described in section 2.3) were developed. These languages build upon RDF and RDFS and add new vocabularies to express even more semantics.

2.3 Web Ontology Language (OWL)

The Web Ontology Language is proposed by the W3C for defining so called ontologies. An ontology can be described as a set of knowledge of some particular domain, such as the neuroscience research. The term is borrowed from philosophy where it denotes study of nature of being. The basic explanation of OWL is in the OWL Guide [8] and in [1], W3C's official site [9] offers many specification documents and other related sources.

OWL builds upon RDF and RDFS. It defines vocabulary that extends the RDF(S) capabilities and also defines rules and restrictions for its usage. Currently, the OWL language has two versions – OWL 1 and OWL 2. The second version was introduced by the W3C group in 2009 and it enriches the OWL 1 version with some new language features. However, version 2 is completely backward compatible with the version 1. Because tools used in my work currently support only OWL 1, I decided to study and work with OWL 1. Therefore features of OWL 1 will be discussed if the language version is not explicitly stated.

The Web Ontology Language can be divided into three sublanguages:

- OWL Lite
- OWL DL
- OWL Full

Their mutual relations are depicted on the next diagram (Figure 2). OWL Lite is a subset of OWL DL, which is a subset of OWL Full. Their difference lies in their expressiveness and syntactical restrictions. OWL Lite is the most restricted one, OWL DL eliminates some of the restrictions and OWL Full offers the most freedom. However, with the increasing expressiveness and syntactical freedom come computational difficulties. In fact, OWL Full has no computational guarantees for a processing software such as reasoners. That is why I work with OWL DL, which is expressive enough and is guaranteed to be computed in finite time by a reasoning software.

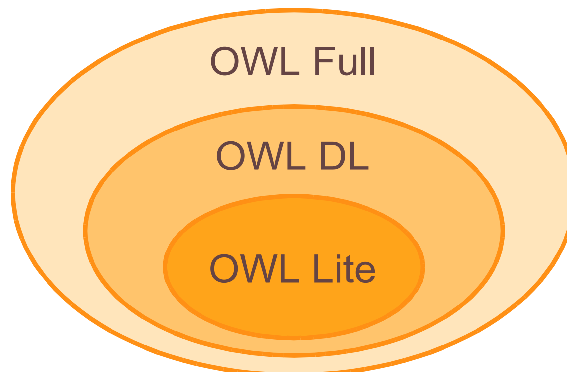

Figure 2: Relations among three OWL sublanguages.

Basic elements of every ontology are classes, properties and individuals. All these elements are resources within the meaning of RDF. OWL comes with its own class type `owl:Class`. A class describes a set of objects (individuals) that have similar characteristics. All classes of a particular ontology creates its core, so called taxonomy.

Properties are used to add statements about resources. OWL works with the `rdf:Property` type divided into several subtypes like `owl:DatatypeProperty` or `owl:ObjectProperty` according to their extension.

Individuals are instances of classes, they represent concrete objects in the domain. A set of individuals that belong to some class is called the class extension. Every individual in OWL is an instance of the generic class `owl:Thing`, all classes are subclasses of this one.

Ontologies are designated for sharing knowledge of some domain. OWL takes advantage of number of serialization syntaxes. The serialization (also called an ontology document) can be simply written to a file or stream. It opens the way for exchanging ontologies among users and applications or simply storing them. The most important syntaxes used for OWL ontologies are:

- RDF/XML
- OWL/XML
- Turtle
- Functional-Style
- Manchester

The first two are XML-based, which means that the ontology document is a valid XML document, the others are not. The RDF/XML syntax is adopted from the RDF definition itself. It is the main syntax which must be supported by all tools for processing ontologies. The support of other ones is optional.

3 EEG/ERP Research

3.1 EEG/ERP Experiments

The electroencephalography is a technique based on measuring and interpreting brain activity. The brain produces ionic flows specific to its functions and activities. They show on the head surface as changes of electrical potential. It can be measured by a set of electrodes connected to a device called electroencephalograph. The electroencephalogram (the measured signal) shows variation of the electrical potential with time. Analysing of these waves can examine functions of the brain and evaluate brain disorders.

Event-related potentials is a technique closely related to EEG. The main topic of interest of this technique are responses of the brain to specific stimuli. The tested person is exposed to some predefined stimuli (e.g. audio or video) and the resulting brain waves are measured and analysed.

The Department of Computer Science and Engineering has a laboratory for EEG/ERP experiments. The research is specialized in attention of drivers or seriously injured people. Collaborators of this research are for example Skoda Auto Inc, University Hospital in Pilsen or Czech Technical University in Prague.

The EEG/ERP experiments are time-consuming and produce a lot of data. There is a lot of information concerning the experiment, such as data related to tested subjects, scenarios (length of the experiment, course etc.), used laboratory equipment, surrounding conditions like weather and temperature, except the measured signals. All this information need to be stored and managed efficiently. That was the reason of development of the EEG/ERP Portal.

3.2 The EEG/ERP Portal

The EEG/ERP Portal[2], hereinafter the Portal, is being developed at the University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering[3] (KIV). Its purpose is to store and manage data gathered from EEG/ERP experiments. Basic description of the Portal is in [10], [11] and [12].

The goal is to enable neuroscience researchers and other interested people to share and interchange data from experiments. Registered users can store, update and download data and metadata from EEG/ERP experiments (described in section 3.1). It is accessible through a Web interface (Figure 3). Data are divided into several semantic groups (e.g. experiments, scenarios, people), a logged-in user can switch among them.

Figure 3: Web interface of the EEG/ERP Portal.

2 EEGbase, URL: <http://eegdatabase.kiv.zcu.cz/home.html>
3 University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering, URL: <http://www.kiv.zcu.cz>

Since the Portal stores personal data about tested subjects it is necessary to protect them from an unauthorized access. That is why a registration is required and user roles are introduced. Users are divided into groups with different access privileges. This and other security issues in the Portal are described in [13].

The Portal is an open-source project built on common technologies like Java or XML. It is based on the three layer architecture. The data layer uses Oracle database for data storage and Hibernate framework for object-relational mapping (ORM). The application and presentation layers are implemented using Spring framework and other Java EE technologies.

The Portal is registered within the Neuroscience Information Framework[4] (NIF), which is a dynamic inventory of Web-based neuroscience resources. NIF advances the neuroscience research by enabling discovery and access to public research data and tools through an open source, networked environment. It enables sharing data from neuroscience experiments among researchers and other interested persons.

Sharing information from neuroscience research poses a problem, because any structure of the shared knowledge within the neuroscience research is not defined. The answer could lie in taking advantage of the Semantic Web technologies. As described in section 2, the Semantic Web offers languages designed for representing knowledge of some domain. The domain is the neuroscience research in this case.

Hence one of the goals of the Portal development is providing the data and metadata from EEG/ERP experiments in the form of an ontology. OWL was chosen as the most suitable language for this purpose. Because the data itself is stored in the relational database, it is necessary to propose and integrate into the Portal a proper transformation tool. This tool should be able to automatically generate the ontology from the stored data. A manual ontology creation is totally unsuitable for this purpose since the data are dynamically changed.

4 Neuroscience Information Framework (NIF), URL: <http://www.neuinfo.org>

4 OOP to OWL Mapping Concepts

Data are stored in a relational database in the Portal. When generating the OWL ontology two basic approaches can be used:

- relational database to OWL ontology mapping
- object-oriented model (OOM) to OWL ontology mapping

Thorough analysis of these possibilities and an overview of available software tools is described in [12]. Both approaches were tested within the Portal environment and the second one was chosen subsequently as the more suitable one [14].

The OOM to OWL approach does not access the database directly, but takes advantage of Hibernate framework used in the Portal. This framework provides an abstraction of the database model in the form of the OOM. The object-relational mapping (ORM) is ensured automatically by the framework and the transformation tool can work with the OOM only. The transformation process is depicted in Figure 4.

The OOM is based on Plain Old Java Objects (POJOs). They are ordinary Java objects used to persist the data. Individual objects corresponds with tables in the relational database. They contain fields corresponding to columns in these tables. Fields are accessible via appropriate getters and setters.

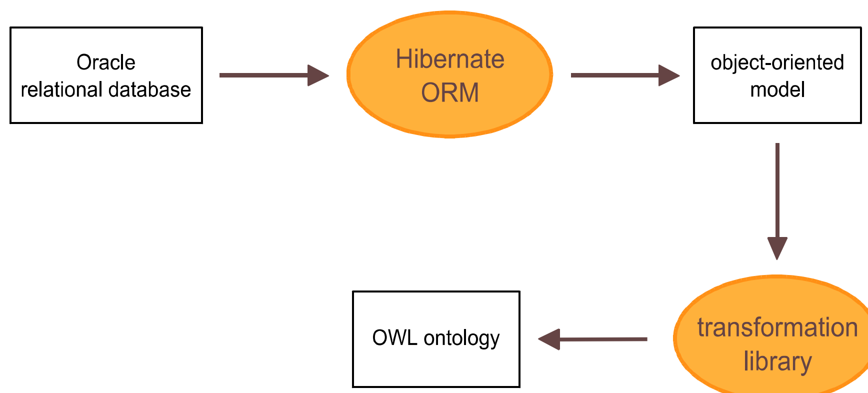

Figure 4: Transformation process in the Portal.

4.1 OOM Compared with OWL

There is a resemblance between data models known from OOP and RDF-based models. Both consist of classes, properties and instances. Properties in OOP are called fields or attributes. Instances in RDF are also called individuals. Both models can create class hierarchy using inheritance. Properties (fields) can take primitive values or other objects as their values. Instances can be classified into classes.

This comparison leads to a basic scheme of the mapping as shown in Table 1.

OOP elements	**OWL elements**
class, interface	class
field (attribute), method	property
instance	individual
primitive datatype	literal

Table 1: Basic scheme of OOP to OWL mapping.

But there are differences that must be taken into consideration. First difference is the conception of a class. While classes (and interfaces) are understood as types in OOP, in OWL classes are considered sets of individuals. This fact is closely related to assigning instances to classes. While in OOP every instance must belong to one class (except its superclasses), an individual in OOP can belong to several diverse classes at the same time. The class membership of an individual can even change at runtime in OWL.

Another important difference consists in approach to properties. While in OOP fields are defined in a class and are connected with this class exclusively (except its subclasses), properties in OWL are stand-alone entities defined outside any class. Individuals can be assigned any property, while instances in OOP can have properties declared in their type only. Therefore encapsulation of class fields known from OOP does not exist in OWL, everything is public and it can be accessed from anywhere.

Objects in OOP are not assigned any global identifier, since they are used locally in an application. They cannot be referenced from outside this application. OWL builds on referencing everything from anywhere that is why it uses URIrefs.

There are many other issues, but the above mentioned ones are essential for the mapping process. The conclusion is that OWL gives more possibilities and expresivity than OOP (at data-modelling level of course). Data from OOM should be mapped into OWL without loss of information. The OOM is poorer in contained semantic information, therefore an adding of this information is needed.

4.2 Jena

Jena[5] is a Java framework for building Semantic Web applications. Its API enable users to create, load and manage RDF data programatically. Jena provides API for RDFS and OWL ontologies, including support for the RDF/XML, Turtle or N-Triple serialization formats. It comprises many other capabilities, such as inferencing or querying, but the above mentioned ones are essential for the Portal. The Jena library is very well documented (including tutorials, Javadoc and a forum support), its basic deployment is described in a clearly arranged tutorial [15].

Jena started as an open source project under the HP Labs Semantic Web Programme around 2000. Lately the development was in decline and seemed to be ended. But currently (January 2012) Jena was adopted by The Apache Software Foundation[6] and is in the so called incubation period (newly accepted projects that are not stabilized yet). Its name changed to Apache Jena and the development was renewed.

The main disadvantage of this library is missing support for OWL 2. It is caused by the decline period, the OWL 2 standard was published at the same time. However, the library contains definition of OWL 2 vocabulary and therefore I suppose adding this feature in future versions.

5 Apache Jena, URL: <http://incubator.apache.org/jena>
6 The Apache Software Foundation, URL: <http://www.apache.org>

4.3 The OWL API

The OWL API[7] is a Java library for creating and manipulating OWL ontologies (similarly to Jena, but this library is specialized strictly in OWL). It supports RDF/XML, OWL/XML, OWL Functional Style, Turtle and partially other serialization formats. This tool is focused towards the OWL 2 standard. It is also very well documented (including code examples, Javadoc and tutorials).

The OWL API is an open source project under either LGPL or Apache Licences. The current version is being developed at the University of Manchester. OWL API is currently the leading tool for processing OWL ontologies.

4.4 JenaBean

JenaBean[8] is a Java library for persisting JavaBeans to RDF. It uses the Jena library. JenaBean works as an additional interface for Jena. It enables automatic transformation of JavaBeans (or POJOs) to the RDF model in Jena. That is useful when mapping the OOM to RDF/OWL, because Jena's API itself does not provide such functionality. JenaBean uses an annotation-based approach to give the object code necessary semantics.

It is an open source project under the Apache License 2.0[9]. In contrast to the above mentioned tools this project is much worse documented. There are not any tutorials as well as Javadoc comments are mostly missing. Its development seems to be ended although it is not written on the project Web page.

---

7 The OWL API, URL: <http://owlapi.sourceforge.net/index.html>
8 JenaBean, URL: <http://code.google.com/p/jenabean>
9 Apache License, Version 2.0, URL: <http://www.apache.org/licenses/LICENSE-2.0.html>

5 JenaBeanExtension

JenaBeanExtension is a Java library that I have been developing. It is used in the Portal as the transformation library. Its input is the object-oriented model in the form of POJOs. Its output is the OWL ontology. The library is intended primarily for the Portal, but it is created as a tool that could be used universally.

The library is based on JenaBean and the project is a continuation of JenaBean Annotation Extension [16] and Java2SemanticWeb [17] projects. The first one started to define and implement Java annotations that can be used to add more semantics into the OOM. The second one was a simple library that created an interface for the Portal to control JenaBean and OWL API. I decided to include this interface in the library itself.

The attached CD contains all source files, a distributable JAR file, an example of use, as well as the Javadoc documentation.

The transformation process within the JenaBeanExtension library is depicted in Figure 5. The library gets a list of data objects (POJOs) that create the data model. These objects are parsed and mapped to the ontology model in Jena, which is a high-level abstraction of the RDF graph. When finished, the model can be serialized in a specified syntax. The serialization could be provided either directly from Jena or alternatively using the OWL API. In the second case the serialization from Jena is loaded by the OWL API and a new serialization is created. The reason is that the OWL API supports some other syntaxes, e.g. OWL/XML.

The parsing process uses Java reflection. The JenaBeanExtension library goes recursively through all provided objects and their fields and creates appropriate resources and statements in the Jena model. There is loaded on the one hand the static structure of data (classes, properties and their relations) and on the other data itself (individuals).

The library keeps a list of already defined classes. If a new class appears it is written to the model (together with metadata provided by present annotations, described below). The same procedure is applied to all its fields. Then the contained data (fields' values) are written to the model as well (as instances of defined classes).

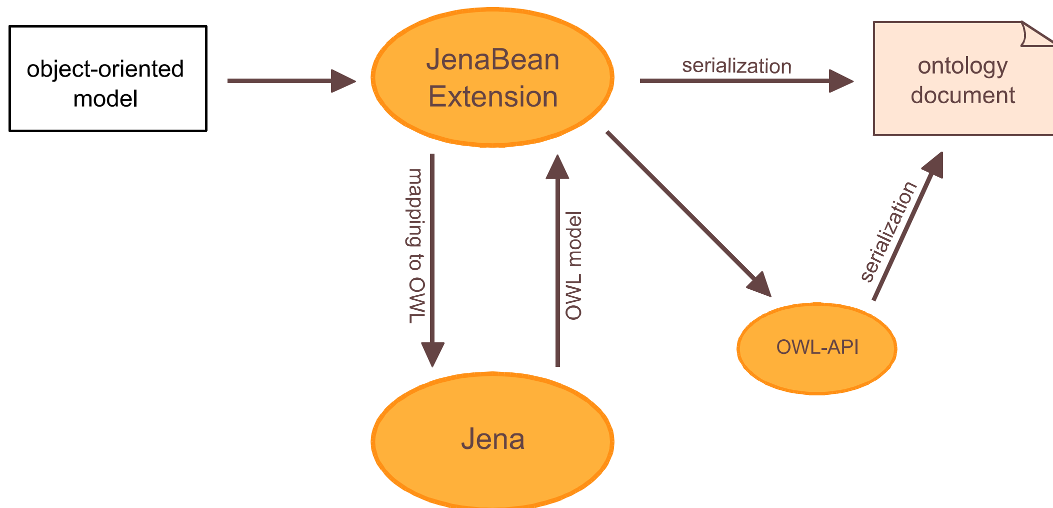

Figure 5: Scheme of the OOP to OWL transformation process.

5.1 Implemented Mapping

Because the required language for the output ontology is OWL, I changed the mapping used in original JenaBean. It used only RDFS vocabulary. Of course, RDF(S) elements are valid in OWL, but OWL provides more expressiveness. Individual language terms are referred by their QNames in the following text. Table 2 gives an overview of used namespaces and their prefixes.

Namespace	Prefix	URI
OWL	`owl`	http://www.w3.org/2002/07/owl#
RDF	`rdf`	http://www.w3.org/1999/02/22-rdf-syntax-ns#
RDF Schema	`rdfs`	http://www.w3.org/2000/01/rdf-schema#
XML Schema	`xsd`	http://www.w3.org/2001/XMLSchema#

Table 2: Used namespaces and their prefixes.

Java classes are mapped into instances of `owl:Class`. Class attributes are mapped into instances of `owl:DatatypeProperty` or `owl:ObjectProperty` (depending on their declared types). Individual Java instances are mapped into individuals containing concrete data values. This is summarized in Table 3.

Java	OWL
class	`owl:Class`
class attribute	`owl:DatatypeProperty` or `owl:ObjectProperty`
instance of class X	individual of type X

Table 3: Mapping of objects.

The naming pattern preserves original names from Java, if they are not changed explicitly by an annotation (more in section 5.2). The namespace is adopted primarily from the Java package name, if it is not changed explicitly by an annotation again. However, this proved to be too restrictive, I implemented other ways to change the namespace for the whole ontology in a simple way (see 5.5). Individuals are named after their runtime class type and their hash code (as a unique key). It can be changed again, by `@Id`, which sets the annotated attribute as the unique key instead of the hash code.

An example of mapping follows. If we consider this POJO class (getters and setters are omitted):

```
package cz.zcu.kiv.pojo;
public class Person {
    private String name;
    private Person friend;
    ...
}
```

It is mapped to (using RDF/XML syntax):

```
<owl:Class rdf:about="http://cz.zcu.kiv.pojo#Person" />
<owl:DatatypeProperty rdf:about="http://cz.zcu.kiv.pojo#name">
  <rdfs:domain rdf:resource="http://cz.zcu.kiv.pojo#Person" />
  <rdfs:range
    rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:about="http://cz.zcu.kiv.pojo#friend">
  <rdfs:domain rdf:resource="http://cz.zcu.kiv.pojo#Person" />
  <rdfs:range rdf:resource="http://cz.zcu.kiv.pojo#Person" />
</owl:ObjectProperty>
```

This example shows the basic naming pattern as well as other features. Properties were divided into object properties and datatype properties according to their declared type in the POJO class. Object properties in OWL are those with an individual as their value. In contrast, datatype properties take literal values. Literal in OWL is similar to a primitive datatype in Java, but it is not the same. Strings or dates are also literals. That is why some Java objects are mapped to literals. Mapping to OWL datatypes used in current JenaBeanExtension shows Table 4. I implemented this mapping because of OWL 2 QL language profile validity. This profile was chosen to be supported by the Portal.

All properties are also implicitly set their domain and range (according to their class membership, resp. their declared type). However, the range of datatypes (in terms of OWL) can be changed by `@DataRange`. If using the triple terminology, the domain defines the class of individuals that can be subjects of the property. Suchlike the range specifies the class of individuals, or the datatype of literals, which can be objects.

Java datatypes and objects	**OWL datatypes**
all integer datatypes (and their wrapper classes)	`xsd:integer`
all floating-point datatypes (and their wrapper classes)	`owl:real`
`boolean` (and its wrapper class)	`xsd:integer`
`char` (and its wrapper class)	`xsd:string`
`java.lang.String`	`xsd:string`
`java.util.Date`	`xsd:dateTime`
`java.util.Calendar`	`xsd:dateTime`

Table 4: Mapping of datatypes.

5.2 Java Annotations

As said in section 4.1, OWL provides more expresivity than the object-oriented model. JenaBeanExtension (like JenaBean) uses an annotation-based approach to add the OOM more semantics. Annotations are standard part of the Java API. Their usage is very clearly described in [18] (chapter 7).

Defined annotations are used in POJO classes. Depending on their meaning they can be used for classes, fields or methods. All these annotations are preserved in the objects at runtime (ensured by their definitions). JenaBeanExtension checks objects on contained annotations and for every annotation adds required information in the OWL model.

Original JenaBean provided several annotations, their detailed description is in [16]. Four of them are still used in the current JenaBeanExtension library. Implementation of the others I found unsuitable and changed it (described in the next section). The four used ones are:

- `@Namespace` – sets namespace for annotated class and its members
- `@RdfType` – sets (different) name for annotated class
- `@RdfProperty` – sets (different) name and namespace for annotated field
- `@Id` – annotated field is used as a unique key for identifying instances

The goal is to implement more annotations to be able to provide POJOs with further semantics (according to expressing abilities of OWL).

5.3 Implemented OWL Language Elements

JenaBeanExtension implements most of OWL 1 vocabulary currently. Some language elements are implemented for implicit transformation (basic ones), most of them are implemented in the form of Java annotations. These annotations must be added to definitions of POJO classes so as the required information is written into the output ontology. Language constructs describing created ontology itself (ontology header) are implemented separately within a special object. In addition, JenaBeanExtension provides way to load statements created by an external tool such as Protége.

Semantics that is created automatically includes primarily classes, properties and their hierarchy. This creates the basic structure of data. It is sometimes called taxonomy, because it enables classifying individuals. This information is gathered automatically from the class structure of the OOM, even when no annotations are present in POJO classes.

All implemented annotations can be found in the library package named `thewebsemantic.annotations`. Names of individual annotations are equivalent to the OWL language construct which it implements. Some annotations are parameterless, some have one or more parameters. Their meaning, usage and some example is provided within their Javadoc documentation and in Appendix A.

Metadata describing the ontology itself (ontology header) was not suitable to be implemented with the annotation-style, because it does not concern any individual class or attribute in the OOM. Therefore I created a special class named `Ontology`, which is part of JenaBeanExtension API (described in 5.5). This class encapsulates ontology properties, which can be set using proper setters. Instance of this class can be created programatically and passed to JenaBeanExtension afterwards. However, this solution can be sometimes unsuitable, that is why the ontology header (as well as any other statements) can be loaded from an auxiliary RDF/XML document (described in section 5.5).

Table 6 gives an overview of all OWL-specific language elements implemented in the current version of JenaBeanExtension. The second column named "Implementation" says how the element is implemented. JenaBeanExtension also implements some RDFS elements. Table 5 gives their overview.

Appendix A provides more detailed description of individual elements, describes their meaning (as defined in OWL reference [19]) and gives examples of use.

RDFS	Implementation
`rdfs:comment`	`@Comment` and `Ontology.setComment()`
`rdfs:domain`	implicit mapping
`rdfs:isDefinedBy`	`@IsDefinedBy`
`rdfs:label`	`@Label` and `Ontology.setLabel()`
`rdfs:range`	implicit mapping
`rdfs:seeAlso`	`@SeeAlso` and `Ontology.setSeeAlso()`
`rdfs:subClassOf`	implicit mapping

Table 5: Overview of implemented RDFS-specific language elements.

OWL element	Implementation
owl:AllDifferent	@AllDifferent
owl:allValuesFrom	@AllValuesFrom
owl:backwardCompatibleWith	Ontology.setBackwardCompatibleWith()
owl:cardinality	@Cardinality
owl:Class	implicit mapping
owl:complementOf	@ComplementOf
owl:DatatypeProperty	implicit mapping
owl:DeprecatedClass	java.lang.@Deprecated
owl:DeprecatedProperty	java.lang.@Deprecated
owl:differentFrom	@DifferentFrom
owl:disjointWith	@DisjointWith
owl:equivalentClass	@EquivalentClass
owl:equivalentProperty	@EquivalentProperty
owl:FunctionalProperty	@FunctionalProperty
owl:hasValue	@HasValue
owl:imports	Ontology.setImports()
owl:incompatibleWith	Ontology.setIncompatibleWith()
owl:InverseFunctionalProperty	@InverseFunctionalProperty
owl:inverseOf	@InverseOf
owl:maxCardinality	@MaxCardinality
owl:minCardinality	@MinCardinality
owl:ObjectProperty	implicit mapping
owl:onProperty	implicit when restriction used
owl:Ontology	Ontology
owl:priorVersion	Ontology.setPriorVersion()
owl:Restriction	implicit when restriction used
owl:sameAs	@SameAs
owl:someValuesFrom	@SomeValuesFrom
owl:SymmetricProperty	@SymmetricProperty
owl:TransitiveProperty	@TransitiveProperty
owl:versionInfo	@VersionInfo and Ontology.setVersionInfo()

Table 6: Overview of all implemented OWL-specific language elements.

5.4 Serialization Syntaxes

Since JenaBeanExtension uses Jena to provide the ontology model, serialization syntaxes available are those supported by Jena. They are:

- RDF/XML (two variants)
- Turtle
- N-Triple (simple Triple Notation)
- N3 (more variants, Turtle is one them, in fact)

Jena offers two variants of the RDF/XML syntax. Both implement W3C's specification and are equivalent, the difference lies in their understandability for a human reader and efficiency of their implementation. They are:

a) RDF/XML – not well human-readable, but the serialization is very efficient

b) RDF/XML-ABBREV – very well human-readable, but the serialization is not very efficient (performance problems for large models are possible)

JenaBeanExtension uses the OWL API in order to provide more serialization syntaxes. The OWL API is able to convert Jena's output into other two syntaxes. Since the OWL API supports OWL 2, it offers syntaxes that came together with the OWL 2 specification, namely:

- OWL/XML
- OWL Functional-Style

It also supports the older ones:

- RDF/XML
- Turtle

5.5 Provided API

JenaBeanExtension provides a simple API for controlling the transformation process. All needed interfaces and classes are located in the package `tools`. The complete API is depicted in UML diagram in Figure 6.

The basic interface of the API is `JenaBeanExtension`. Its method `loadOOM(List<Object> dataList)` runs the transformation. Its argument is a list of POJOs. The ontology can be obtained subsequently in an input stream (`java.io.InputStream`) using `getOntology(String syntax)`, which returns the serialization from Jena. Available serialization syntaxes are defined in the class `Syntax`. The interface also offers static structure of the ontology (i.e. defined classes and properties), using `getOntologySchema(String syntax)`. All these basic methods are provided in several variants. Javadoc is on the attached CD.

This interface also enables to load a serialization document using `loadStatements(InputStream document, String syntax)`. It can be used to load either an whole ontology (e.g. from a previous serialization), or several statements only which will be added to the model created from POJOs.

This approach offers a more convenient way to add the ontology header (the other one using the `Ontology` class was described in 5.3). Required information, such as the ontology header, can be stored in an external RDF/XML file. This file can be easily edited with a graphical tool such as Protége. When the transformation starts this file is loaded and contained information is added to the model created from POJOs. Moreover, if the ontology header is loaded before the transformation (method `loadOOM()`), the ontology namespace is used as a default namespace for all the resources created during the tranformation (classes, properties, individuals).

An example of typical usage of the library follows:

```
List<Object> dataList;       // list of POJOs
InputStream ontologyHeader; // RDF/XML
InputStream ontology;       // generated ontology

     ...  // initializing dataList and ontologyHeader

JenaBeanExtension jbe = new JenaBeanExtensionTool();
jbe.loadStatements(ontologyHeader, Syntax.RDF_XML);
jbe.loadOOM(dataList);

ontology = jbe.getOntology(Syntax.RDF_XML);
```

Because the ontology serialization is obtained from Jena, which does not support OWL 2 and the OWL/XML syntax, JenaBeanExtension provides tool to simply convert the serialization from RDF/XML to OWL/XML using the OWL API. This tool implements interface `OwlApi`. Its constructor takes the serialization in RDF/XML from Jena and loads the ontology into the OWL API. Method `getOntologyDocument()` can be used afterwards to obtain the serialization.

Although the serialization obtained in OWL/XML looks like OWL 2, it is still the OWL 1 ontology that was created by Jena.

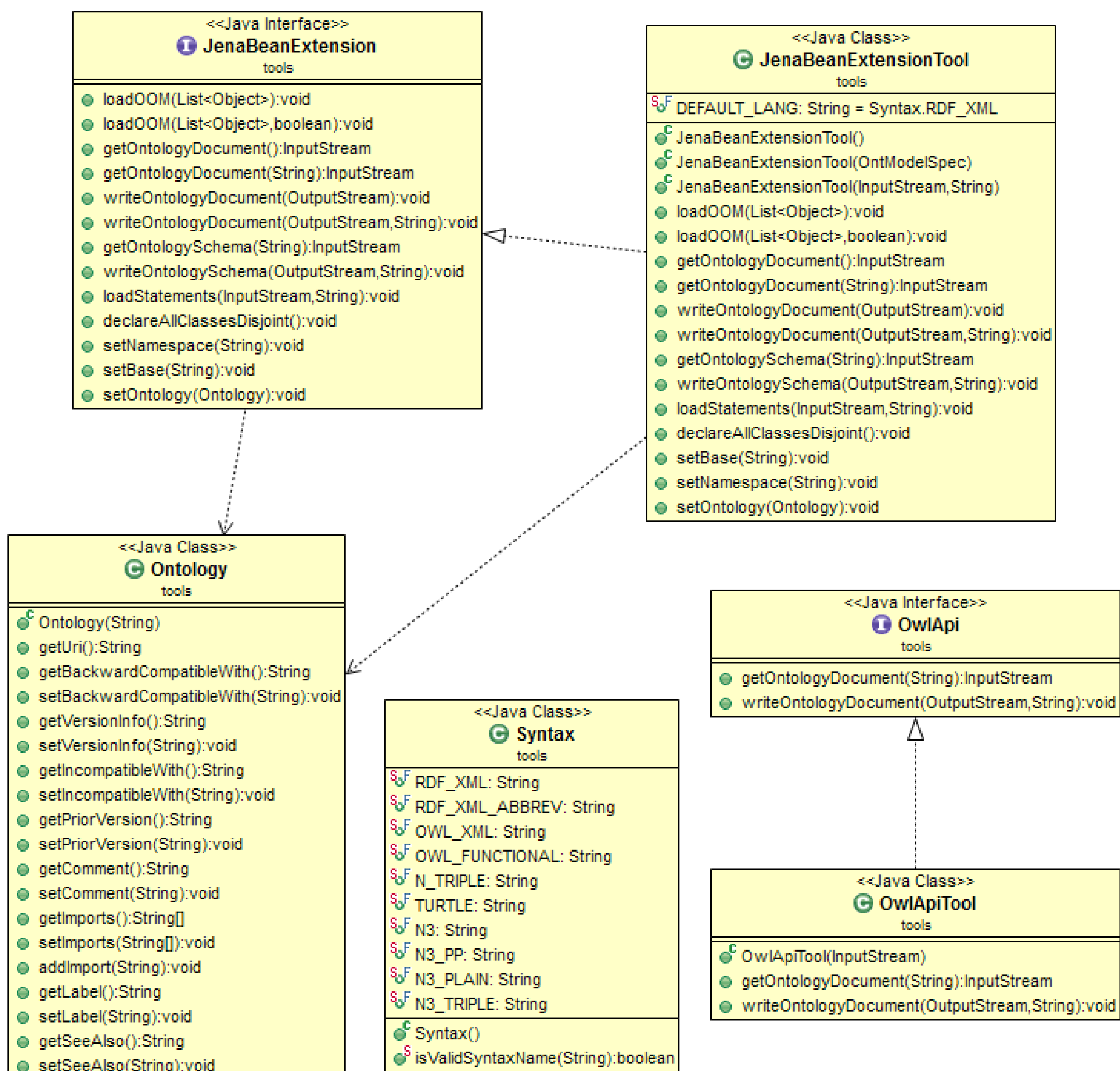

Figure 6: UML diagram of JenaBeanExtension API.

6 Integration into the Portal

The JenaBeanExtension library has been used in the testing version of the Portal during the development. Therefore I have also ensured its proper function and regular upgrades. This has brought some advantages. The library was tested in the real environment and I was able to respond to actual problems and needs.

6.1 Getting the Ontology Document

The ontology document is provided on the Portal's Web interface from defined URLs. So as the URLs express their meaning, I changed them as follows:

`/semantic/getOntology.html`

gets the default RDF/XML serialization from Jena. The syntax can be changed using parameter `type`. Its values can be `rdf/xml`, `n-triple`, `turtle` or `n3`. For example `/semantic/getOntology.html?type=turtle` gets the ontology in the Turtle syntax.

`/semantic/getOntologyOwlApi.html`

gets the OWL/XML serialization from the OWL API. The syntax can be changed using parameter type. Its values can be `rdf/xml`, `owl/xml`, `turtle` or `owl-functional`. For example `/semantic/getOntologyOwlApi.html?type=owl-functional` gets the ontology in the OWL Functional-Style syntax.

`/semantic/getOntologyStructure.html`

gets the ontology structure (or schema) in RDF/XML (abbreviated version) from Jena. The ontology does not contain any data, only definitions of classes and properties.

Sample fragments of ontology documents from the Portal are enclosed in Appendix B (in RDF/XML). They comprise a very small part of the ontology and are stated by way of illustration only. The whole ontology document is on the attached CD.

6.2 Jena Models Difficulties

Model is a structure in Jena that represents the RDF graph. The model is a high-level abstraction of the RDF graph, which provides high-level operations, such as creating resources, adding statements, querying for resources with given properties or combining several models. Jena offers number of model interfaces with different qualities. The Jena API provides `ModelFactory` for creating standard types of models.

The original JenaBean used a default model provided by the `ModelFactory.createDefaultModel()` method. This kind of model was not adequate for JenaBeanExtension, because it does not support adding OWL statements. An OWL-compatible model is called `OntologyModel` in Jena. `ModelFactory` provides a default one:

```
Model m = ModelFactory.createOntologyModel();
```

This one was used in JenaBeanExtension at first, but problems occurred during integration into the Portal. The computational time was unacceptable. While the previous version (original JenaBean with the `DefaultModel`) computed a few minutes, the extended version seemed to be working many hours or even a few days (tested on Asus with Duo T6500 CPU, 4 GB RAM).

The reason is that the default `OntologyModel` included a RDFS-level inference, which imposes a computational cost. Because we did not use reasoning in the project so far, the simplest solution was to disable the reasoner. Jena's class `OntModelSpec` provides constants for defining `OntologyModel` specification. An `OWL_MEM` specification creates model without reasoning:

```
Model m = ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM);
```

I measured the computational time depending on the amount of processed data objects. The graph in Figure 7 shows results for the `OntologyModel` with reasoning compared with the `OntologyModel` without reasoning. While the dependence is linear without reasoning, the reasoner causes at least quadratic dependence. Because there is processed a large amount of data objects in the Portal, I found the reasoner included in the Jena library unfit for our project.

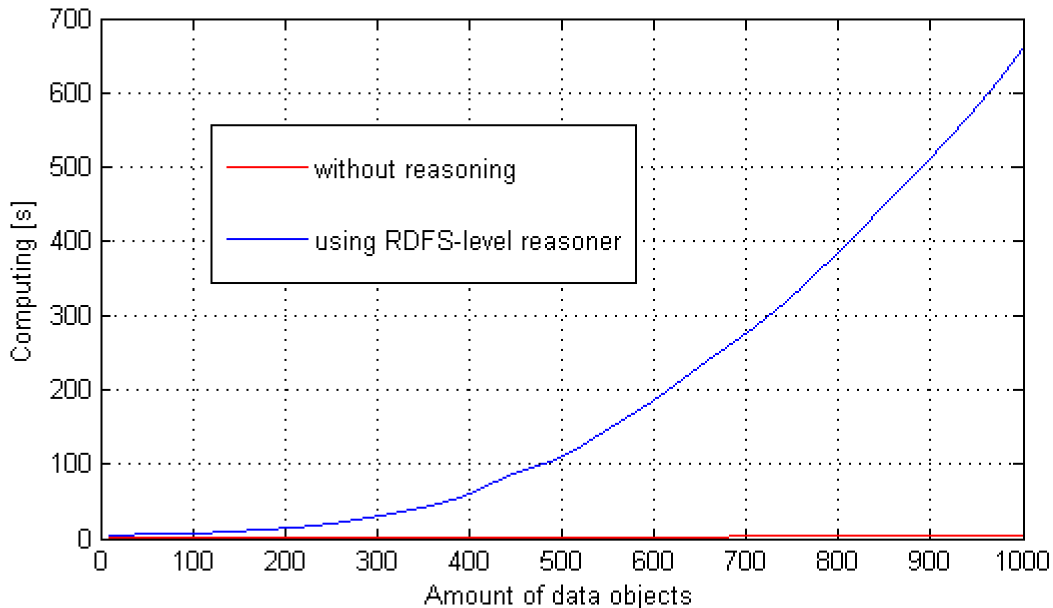

Figure 7: Computational demands of the Ontology Model.

The reasoner comes in useful primarily when querying the model. I suppose that would not be useful in the Portal, because so far the model serves as an intermediate stage during the transformation process. Its only usage after loading the data is providing the serialization in a specified syntax (RDF/XML, N-Triple, Turtle or N3 currently available). But this output ontology document can be affected by the reasoner as well.

The `OntologyModel` provides two groups of methods for getting its serialization (they can be used with various parameters, which are not important now). The first ones are named `write(...)` and they do not cooperate with the reasoner. That means the output contains only the asserted data. But the second group named `writeAll(...)` includes inferred statements in the serialization. I think that could be useful, since it would enrich the output ontology document. Unfortunately, it cannot be used because of the above mentioned performance problem.

6.3 RDF/XML Serialization Syntax

JenaBeanExtension allows users to get the ontology in several serialization syntaxes (described in 5.4). Since RDF/XML is the basic one (according to W3C's recommendations) I have implement it as the default one if the `type` parameter is not set properly (getting the ontology is described in 6.1).

There is a choice in Jena if the RDF/XML should be abbreviated or not. Since the abbreviated version is much better human-readable, I wanted to prefer it. But the implementation of Jena's RDF/XML-ABBREV serializer is not very efficient. This matter is pointed out in Jena's documentation, too. The problem is that the Portal manages a lot of data and the generated RDF graph is very large. Currently the ontology in the Portal (testing version) contains about 40,000 statements. It is about 60,000 lines in the RDF/XML serialization.

For that reasons the abbreviated version of the ontology is not possible. In the past I succeeded in getting the abbreviated version, but the time of creating the serialization ran into hours (Duo T6500 CPU, 4 GB RAM). Moreover the database contained less data then. Currently the serialization process for RDF/XML-ABBREV falls on lack of memory (4 GB RAM).

The only provided version of RDF/XML is the raw one. However, this restriction does not concern the ontology schema document (describing structure of classes and properties). Since this RDF graph is much smaller then the whole ontology, the abbreviated version does not pose any problem. RDF/XML-ABBREV is the default syntax for the ontology schema because of its human-readability.

6.4 Optimization of the Transformation Process

The transformation process was initially invoked whenever any user requested the ontology document from the Web interface using defined URLs. This approach brought two problems:

- The user had to wait until the transformation is finished and the output is produced. The processing time is quite significant and for the user very unpleasant.
- Every transformation entails appreciable workload for the database, because all the stored data are loaded.

For that reason I have introduced another solution – the ontology is generated automatically at regular intervals and its serialization is stored in a temporary file.

I have used `java.util.Timer` to schedule regular transformation tasks. The transformation is activated after the system starts and then regularly once a day. The ontology is stored in a temporary file (provided by `java.io.File.createTempFile()`) in RDF/XML. When a user requires the ontology, the serialization is simply read from the temporary file. If the required syntax is other than RDF/XML, the serialization is loaded back to JenaBeanExtension and the required serialization is created afterwards. This is much less time consuming, the waiting time for user is a few seconds. Moreover the database server is not burdened at all.

All the code that is responsible for the transformation process is located in `cz.zcu.kiv.eegdatabase.logic.semantic.SimpleSemanticFactory`. This bean is initialized by the Spring framework when the Portal starts. It provides methods for getting the ontology document which are used by the web-controller beans.

6.5 Invalid Characters in XML

During the testing period there occurred problems with encoding some characters in XML. The data in OOM contained characters with the code point `0x00` (NULL). Since this character is invalid in XML and the ontology is serialized in RDF/XML, the transformation fell on an exception.

Neither JenaBeanExtension nor Jena check characters when creating the model. That could lower the performance and moreover these characters are not invalid generally, but only in the context of XML. For example the Turtle serialization accepts them without any problem. Jena uses XML 1.0 for the RDF/XML serialization, therefore I focused on XML 1.0 valid characters.

Valid XML 1.0 characters are defined in the XML specification [20]. They are described by a set of valid Unicode code points:

```
#x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] |
[#x10000-#x10FFFF]
```

It follows that invalid characters are:

- most of C0 control characters (`0x00 - 0x1F`)
- all surrogates (`0xD800 - 0xDFFF`)
- non-characters `0xFFFE` and `0xFFFF`

Although it comprises quite large amount of invalid characters, they should not ordinarily appear in the data. The only invalid character that appeared in the data so far was the `0x00` code point. It was held in a field of type `char`. Since this value has a meaning of NULL, JenaBeanExtension ignores it as if the field was not set.

6.6 Proxy Objects in the OOM

I stated that Hibernate provides a list of POJOs in section 4. To be exact, Hibernate creates proxy objects representing requested POJOs. The framework uses lazy initialization, data are not loaded from the database until they are accessed, which occurs just during the parsing process in JenaBeanExtension.

Since JenaBeanExtension uses reflection, proxy objects caused problems. Ascertainment of the class name of a proxy object does not work as expected. For example calling the method `getName()` over a proxy for class `Person` returns something like “Person_$$_javassist_26” instead of “Person” (Hibernate uses Javassist proxy classes). Also iterating over all class members results in defining objects like initializers or handlers in the output ontology (except the requested data).

The solution is based on the fact that Hibernate proxy classes are subclasses of original POJO classes and implement the interface `org.hibernate.proxy.HibernateProxy`. The proxy class can be simply detected before parsing and its superclass used instead. Following code fragment shows the principle of this solution:

```
Object bean;  // POJO object
...
Class<?> cls = (bean instanceof HibernateProxy) ?
           bean.getClass().getSuperClass() : bean.getClass();
```

```
parse(cls);  // parsing class members
```

7 Conclusion

The goal of this work was to propose and implement a proper transformation tool for the EEG/ERP Portal, which is written in Java and works with the object-oriented data model. The EEG/ERP Portal is being registered in the Neuroscience Information Framework that enables sharing data using Semantic Web technologies. The transformation should automatically transform the object-oriented model into an OWL ontology.

At first I had to familiarise myself with the Portal and Semantic Web technologies. Semantic Web is quite a new and not widely used technology so far. This fact is closely related to lack of literature about this topic. Specifications and recommendations proposed by the World Wide Web Consortium are the main source of information.

The transformation tool that I have developed is based on the open-source library JenaBean. Its code has been modified so as the tool meets Portal's requirements. It was necessary to adjust the mapping from the object model to the OWL ontology. The output complies with the OWL 2 QL language profile currently. I have also implemented most of OWL 1 language elements. They can be used in the form of Java annotations to enrich the object model with more semantics.

The tool was tested in the Portal environment. The large amount of contained data revealed a few problems, mostly concerning performance of the transformation. They were all successfully resolved. However, there remains one future issue – definitions of POJO classes need adding Java annotations provided by JenaBeanExtension in order to take full advantage of its capabilities. I have added a few annotations as a demonstration example only.

The main deficiency of the tool is the missing support for OWL 2. Since the ontology model is created in Jena, which supports only OWL 1 so far, it is unavoidable to wait until Jena adds support for OWL 2. An alternative solution could consist in a replacement of Jena by the OWL API, which provides similar interface for creating OWL ontologies and supports OWL 2. That would mean a complete reworking of the transformation library, but in my opinion it is feasible and worth considering.

Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
EEG	Electroencephalography
ERP	Event-Related Potentials
JAR	Java Archive
NIF	Neuroscience Information Framework
OOM	Object-Oriented Model
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
OWL	Web Ontology Language
POJO	Plain Old Java Objects
RDF	Resource Description Framework
RDFS	RDF Schema
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Used Software

Software	Description
Eclipse	Integrated development environment, free licence.
Inkscape	Vector graphics editor, free licence.
IntelliJ Idea	Integrated development environment.
Matlab	Environment for technical computing.
Notepad++	Source code editor, free licence.
OpenOffice	Office productivity software suite, free licence.
Protége	Ontology editor, free licence.
TortoiseSVN	Subversion client, free licence.
Windows 7	Operating system.

References

[1] HITZLER, Pascal, KRÖTZSCH, Markus, RUDOLPH, Sebastian. *Foundations of Semantic Web Technologies*. Boca Raton: Chapman & Hall / CRC, 2009. ISBN 978-1-4200-9050-5.

[2] World Wide Web Consortium (W3C). *Semantic Web* [online]. [cit. 2012-01-05]. URL: <http://www.w3.org/standards/semanticweb>

[3] World Wide Web Consortium (W3C). *RDF Primer* [online]. [cit. 2012-02-06]. URL: <http://www.w3.org/TR/2004/REC-rdf-primer-20040210>

[4] World Wide Web Consortium (W3C). *Resource Description Framework (RDF)* [online]. [cit. 2012-01-15]. URL: <http://www.w3.org/RDF>

[5] World Wide Web Consortium (W3C). *RDF/XML Syntax Specification (Revised), W3C Recommendation* [online]. [cit. 2012-03-10]. URL: <http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210>

[6] World Wide Web Consortium (W3C). *XML Schema Part 2: Datatypes, W3C Recommendation* [online]. [cit. 2012-03-10]. URL: <http://www.w3.org/TR/2001/REC-xmlschema-2-20010502>

[7] World Wide Web Consortium (W3C). *RDF Vocabulary Description Language 1.0: RDF Schema* [online]. [cit. 2012-01-26]. URL: <http://www.w3.org/TR/2004/REC-rdf-schema-20040210>

[8] World Wide Web Consortium (W3C). *OWL Web Ontology Language Guide* [online]. [cit. 2012-01-14]. URL: <http://www.w3.org/TR/owl-guide>

[9] World Wide Web Consortium (W3C). *Web Ontology Language (OWL)* [online]. [cit. 2011-11-02]. URL: <http://www.w3.org/2004/OWL>

[10] JEŽEK, Petr, MOUČEK, Roman. *Database of EEG/ERP experiments*. Third International Conference on Health Informatics, 2010, Valencia, Spain.

[11] JEŽEK, Petr, MOUČEK, Roman. *EEG/ERP Portal – Semantic Web Extension, Generating Ontology from Object Oriented Model*. Second Global Congress on Intelligent Systems, 2010, Wuhan, China. ISBN 978-1-4244-9247-3.

[12] ČERNÝ, Jaroslav. *Neuroinformatics Database and Semantic Web Resources*. Pilsen, 2011. Diploma thesis. University of West Bohemia, Department of Computer Science and Engineering.

[13] VLAŠIMSKÝ, Jiří. *Access privileges in EEG/ERP portal (Systém oprávnění v EEG/ERP portálu)*. Pilsen, 2011. Diploma thesis. University of West Bohemia, Department of Computer Science and Engineering.

[14] BRŮHA, Petr. *EEG/ERP Portal and Resources of Semantic Web (EEG/ERP portál a prostředky sémantického webu)*. Pilsen, 2011. Diploma thesis. University of West Bohemia, Department of Computer Science and Engineering.

[15] Jena. *An Introduction to RDF and the Jena RDF API* [online]. [cit. 2011-11-23]. URL: <http://incubator.apache.org/jena/tutorials/rdf_api.html>

[16] MARKVART, Filip. *EEG/ERP portal - transformation to semantic web (EEG/ERP portál - transformace do sémantického webu)*. Pilsen, 2011. Bachelor thesis. University of West Bohemia, Department of Computer Science and Engineering.

[17] ŠMÍD, Dominik. *Database of EEG/ERP experiments and semantic web (Databáze EEG/ERR experimentů a sémantický web)*. Pilsen, 2010. Bachelor thesis. University of West Bohemia, Department of Computer Science and Engineering.

[18] PECINOVSKÝ, Rudolf. *Java 5.0 - Language innovations and upgrade of applications (Java 5.0 - Novinky jazyka a upgrade aplikací)*. Brno: CP Books, a. s., 2005. ISBN 80-251-0615-2.

[19] BECHHOFER, S., HARMELEN, F. van, HENDLER, J., HORROCKS, I., MCGUINESS, D. L., PATEL-SCHNEIDER, P., STEIN, L. A.. *OWL Web Ontology Language Reference* [online]. [cit. 2012-03-10]. URL: <http://www.w3.org/TR/owl-ref>

[20] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online]. [cit. 2012-04-15]. URL: <http://www.w3.org/TR/xml>

Appendix A: Implemented OWL language elements

A.1 Classes

owl:Class

A class in OWL is defined as a group of individuals that have some common characteristics. They do not have to share all properties. `Owl:Class` is a subclass of `rdfs:Class`. These two elements are equivalent in OWL Full, but in OWL Lite or DL `owl:Class` must meet some restrictions.

Implementation: Java classes from the object model are mapped into `owl:Class` instances. There is no need to use any annotation in the Java source code, every class is mapped implicitly. Its name is preserved from the Java class name and its namespace is implicitly adopted from the Java package name. However, this can be changed by using the `@Namespace` annotation for the Java class. Another way is to set the default namespace in the JenaBeanExtension constructor (but this will set the namespace for all classes in the model).

For example:

```
package data.pojo;
public class Person { ... }
```

This class is mapped into:

```
<owl:Class rdf:about="http://data.pojo#Person" />
```

owl:complementOf

If class A is a complement of class B, then every individual that does not belong to B must belong to A and vice versa. Classes have no common individual. This relation can be expressed as a logical negation.

Implementation: Implemented as the class annotation `@ComplementOf`. Its value must be a well-formed URI (referencing the complement class).

Example of use:

```
@ComplementOf("http://some.ontology#EverythingExceptPerson")
public class Person {
  ...
}
```

This class is mapped into:

```
<owl:Class rdf:about="#Person">
  ...
  <owl:complementOf>
    <owl:Class
      rdf:about="http://some.ontology#EverythingExceptPerson"/>
  </owl:complementOf>
</owl:Class>
```

owl:disjointWith

This property expresses that two classes have no common individuals.

Implementation: Implemented as the class annotation `@DisjointWith`. Its value must be a well-formed URI (referencing the disjoint class).

Example of use:

```
@DisjointWith("http://some.ontology#Animal")
public class Person {
  ...
}
```

This class is mapped into:

```
<owl:Class rdf:about="#Person">
  ...
  <owl:disjointWith>
    <owl:Class rdf:about="http://some.ontology#Animal"/>
  </owl:disjointWith>
</owl:Class>
```

owl:equivalentClass

This property expresses that two classes have the same class extension (the same set of individuals), but not necessarily the same concepts. That means although two equivalent classes have the same instances, they does not have to be equal. In OWL Full class equality can be expressed using below mentioned `owl:sameAs`. OWL Lite or DL cannot express class equality.

Implementation: Implemented as the class annotation `@EquivalentClass`. Its value must be a well-formed URI (referencing the equivalent class).

Example of use:

```
@EquivalentClass("http://some.ontology#Man")
public class Person {

  ...

}
```

This class is mapped into:

```
<owl:Class rdf:about="#Person">

  ...

  <owl:equivalentClass>
    <owl:Class rdf:about="http://some.ontology#Man"/>
  </owl:equivalentClass>

</owl:Class>
```

A.2 Ontology Header

Following properties are used to create statements about individual ontologies. Ontology itself is also a resource (an instance of `owl:Ontology`), that is why it can be described the same way as classes or properties. Statements describing the ontology comprise so called ontology header.

An ontology can be described using common datatype or object properties. OWL also defines ontology properties as instances of `owl:OntologyProperty`. These properties are in fact object properties that both domain and range of which are instances of `owl:Ontology`. They can link mutually compatible ontologies, import other ontologies and so on.

I implemented `owl:Ontology` and all its properties as a Java class within the JenaBeanExtension API. I could not take advantage of the annotation-based approach, because these statements does not concern any individual classes or fields, but the whole model. The object-oriented model cannot contain this information, it must be passed on to the JenaBean Extension additionally. For these purposes can be used an instance of the `Ontology` class. Following example shows the way of setting the ontology header:

```
JenaBeanExtension jbe;
Ontology ontology;
...  // creating the model

ontology = new Ontology("http://kiv.zcu.cz/eegdatabase");
...(set ontology properties)...
jbe.setOntology(ontology);
```

owl:Ontology

Ontology itself is a resource defined by this element, referred to as an ontology header. It should be defined near the beginning of the ontology document. Ontology header contains information about the ontology itself. Information about ontology can be stated using properties as well as for classes and other resources. There is also a special kind of properties in OWL that links an ontology to an ontology (`owl:OntologyProperty`).

Implementation: Implemented as the Java class `Ontology` within the JenaBean Extension API. Its instance is used to set required ontology properties. It is

passed to JenaBeanExtension afterwards using the `setOntology()` method and the `owl:Ontology` element is added to the ontology document.

Example of use:

```
JenaBeanExtension jbe;
jbe = new JenaBeanExtensionTool(datalist); // creating owl model
jbe.setOntology(new Ontology("kiv.zcu.cz/eegdatabase"));
...
```

The third line adds to the ontology document:

```
<owl:Ontology rdf:about="http://kiv.zcu.cz/eegdatabase" />
```

owl:backwardCompatibleWith

This is an ontology property. It defines the referenced ontology as a prior version of the containing ontology and asserts that the new version is backward compatible with the prior one. This can be useful when importing some third party's ontology into our ontology. If the third party's ontology has a new version and it is declared as backward compatible, we can safely change the import statement to the new version.

Implementation: Implemented within the `Ontology` class, which is one of the JenaBean Extension interface classes. The `owl:backwardCompatibleWith` element can be set using the `setBackwardCompatibleWith()` method upon this class. Its argument must be a well-formed URI referencing the prior version of the ontology.

Example of use:

```
Ontology ontology = new Ontology("http://kiv.zcu.cz/eegbase/2.0");
ontology.setBackwardCompatibleWith("http://kiv.zcu.cz/eegbase/1");
...
```

This adds following ontology header:

```
<owl:Ontology rdf:about="http://kiv.zcu.cz/eegbase/2.0">

  <owl:backwardCompatibleWith
    rdf:resource="http://kiv.zcu.cz/eegbase/1" />
  ...

</owl:Ontology>
```

owl:priorVersion

This is an ontology property. It defines the referenced ontology as a prior version of the containing ontology like `owl:backwardCompatibleWith`, but this statement does not say anything about their compatibility. It can be used to manage ontology versions only.

Implementation: Implemented within the `Ontology` class as its method `setPriorVersion()`. Its argument must be a well-formed URI referencing the prior version.

Example of use:

```
Ontology ontology = new Ontology("http://kiv.zcu.cz/eegbase/2.0");
ontology.setPriorVersion("http://kiv.zcu.cz/eegbase/1.0");
...
```

This adds following ontology header:

```
<owl:Ontology rdf:about="http://kiv.zcu.cz/eegbase/2.0">

  <owl:priorVersion
    rdf:resource="http://kiv.zcu.cz/eegbase/1.0" />
  ...

</owl:Ontology>
```

owl:incompatibleWith

This ontology property is the opposite of `owl:backwardCompatibleWith`. It defines the referenced ontology as a prior version of the containing ontology and asserts that they are not compatible each other. This fact should be assumed automatically whenever the `owl:backwardCompatibleWith` element is not present. The `owl:incompatibleWith` element can be used to explicitly emphasize this fact.

Implementation: Implemented within the `Ontology` class as its method `setIncompatibleWith()`. Its argument must be a well-formed URI referencing the incompatible prior version.

Example of use:

```
Ontology ontology = new Ontology("http://kiv.zcu.cz/eegbase/2.0");
ontology.setIncompatibleWith("http://kiv.zcu.cz/eegdatabase");
...
```

This adds following ontology header:

```
<owl:Ontology rdf:about="http://kiv.zcu.cz/eegbase/2.0">
  <owl:incompatibleWith
    rdf:resource="http://kiv.zcu.cz/eegdatabase" />
  ...
</owl:Ontology>
```

owl:imports

This is an ontology property that is used to import another ontology into our ontology. The statement contains URI that reference the imported ontology. It can be a third party ontology. It contains definitions that we want to use in the importing ontology. If the imported ontology contains the `owl:imports` statement as well, they are imported both.

Implementation: Implemented within the `Ontology` class as its methods `addImport()` or `setImports()`. The first one adds reference to one ontology and can be used several times. The second one sets all imports at once, its argument is an array of references. References must be well-formed URIs.

Example of use:

```
Ontology ontology = new Ontology("http://kiv.zcu.cz/eegbase/2.0");
ontology.addImport("http://some.address/ontology");
ontology.addImport("http://another.address/ontology");
...
```

This adds following ontology header:

```
<owl:Ontology rdf:about="http://kiv.zcu.cz/eegbase/2.0">
  <owl:imports rdf:resource="http://some.address/ontology" />
  <owl:imports
    rdf:resource="http://another.address/ontology" />
  ...
</owl:Ontology>
```

A.3 Properties

owl:DatatypeProperty

This class is used in OWL to define datatype properties. Datatype properties link individuals to data values. `Owl:DatatypeProperty` is a subclass of `rdf:Property`.

Implementation: Class fields from the Java source code are mapped into properties in the ontology model. If the field's declared type is equivalent with some datatype used in OWL (for example integer, string, date) then the field is mapped into an `owl:DatatypeProperty` instance. There is no need to use any annotation in the Java source code, every field is mapped implicitly. Its name is preserved from the Java field's name and its namespace is borrowed from the Java class that declares this field.

The property is also set the `rdfs:domain` and `rdfs:range` axioms. Domain specifies the subject that can own this property and is set to the `owl:Class` instance that represents the Java class declaring this field. Range specifies possible values of the property (object from the triple). It is set to a datatype equivalent to the field's declared type.

For example:

```
public class Person {
  ...
  private String name;
}
```

The `name` field is mapped into:

```
<owl:DatatypeProperty rdf:about="#name">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range
    rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

owl:ObjectProperty

This class is used in OWL to define object properties. Object properties link individuals to individuals. `Owl:ObjectProperty` is a subclass of `rdf:Property`.

Implementation: Class fields from the Java source code are mapped into properties in the ontology model. If the field's declared type is not equivalent with any datatype used in OWL then the field is mapped into an `owl:ObjectProperty` instance. There is no need to use any annotation in the Java source code, every field is mapped implicitly. Its name is preserved from the Java field's name and its namespace is borrowed from the Java class that declares this field.

The property is also set the `rdfs:domain` and `rdfs:range` axioms. Domain specifies the subject that can own this property and is set to the `owl:Class` instance that represents the Java class declaring this field. Range specifies possible values of the property (object from the triple). It is set to the `owl:Class` instance that represents the field's declared type.

For example:

```
public class Person {
  ...
  private Profession profession;
}
```

The `profession` field is mapped into:

```
<owl:ObjectProperty rdf:about="#profession">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="#Profession"/>
</owl:ObjectProperty>
```

owl:FunctionalProperty

This built-in class is used to specify its instances to be functional. A functional property can have only one value for a given subject. A functional property can be either a datatype property or an object property.

Implementation: Implemented as the field annotation `@FunctionalProperty`. It is parameterless.

Example of use:

```
public class Person {
  ...
  @FunctionalProperty
  private String surname;
}
```

The `surname` field is mapped into:

```
<owl:FunctionalProperty rdf:about="#surname">
  ...
</owl:FunctionalProperty>
```

owl:InverseFunctionalProperty

This built-in class specifies its instances to be inverse-functional. It is a subclass of `owl:ObjectProperty`. An inverse-functional property can have only one subject for a given object. It means that the object of the property unequivocally determines the subject. No two subjects can have the same object for the property.

Implementation: Implemented as the field annotation `@InverseFunctionalProperty`. It is parameterless.

Example of use:

```
public class Person {
  ...
  @InverseFunctionalProperty
  private BirthNumber birthNumber;
}
```

The `birthNumber` field is mapped into:

```
<owl:InverseFunctionalProperty rdf:about="#birthNumber">
  ...
</owl:InverseFunctionalProperty>
```

owl:SymmetricProperty

This element is used to specify a property to be symmetric. Symmetric property means that the subject and the object from the triple can be interchanged and the statement is true as well. In other words, if the pair (X, Y) is an instance of a symmetric property, then the pair (Y, X) is an instance of this property, too. It follows that its domain and range must be the same. `Owl:SymmetricProperty` is a subclass of `owl:ObjectProperty`.

Implementation: Implemented as the field annotation `@Symmetric` without arguments. If a field is marked by this annotation, the resulting property will be specified as an `owl:SymmetricProperty` instance. JenaBeanExtension does not check if the annotation is used properly (domain and range must be the same), it must be arranged by a programmer.

Example of use:

```
public class Person {
  ...
  @Symmetric
  private Person friend;
}
```

The `friend` field is mapped into:

```
<owl:SymmetricProperty rdf:about="#friend">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Person" />
</owl:SymmetricProperty>
```

owl:TransitiveProperty

This element is used to specify a property to be transitive. It is useful primarily for inferencing. Transitive property means that if we have two pairs (X, Y) and (Y, Z) as instances of a transitive property, then the pair (X, Z) is also an instance of this property. `Owl:TransitiveProperty` is a subclass of `owl:ObjectProperty`.

Implementation: Implemented as the field annotation `@Transitive` without arguments. If a field is marked by this annotation, the resulting property will be specified as an `owl:TransitiveProperty` instance.

Example of use:

```
public class Person {
  ...
  @Transitive
  private Person classmate;
}
```

The `classmate` field is mapped into:

```
<owl:TransitiveProperty rdf:about="#classmate">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource="#Person" />
</owl:TransitiveProperty>
```

owl:equivalentProperty

This statement links two properties to be equivalent, which means they have the same property extension (the same values). It does not necessarily mean that they are equal (have the same meaning). In OWL Full property equality can be expressed using below mentioned `owl:sameAs`. OWL Lite or DL cannot express class equality.

Implementation: Implemented as the field annotation `@EquivalentProperty`. Its value must be a well-formed URI (referencing the equivalent property).

Example of use:

```
public class Person {
  ...
  @EquivalentProperty("http://some.ontology#givenName")
  private String name;
}
```

The `name` field is mapped into:

```
<owl:DatatypeProperty rdf:about="#name">
  ...
  <owl:equivalentProperty
    rdf:resource="http://some.ontology#givenName" />
</owl:DatatypeProperty>
```

owl:inverseOf

This statement links two properties that are inverse each other. It means they describe the same relation from the other side (some parent has a child, this child has that parent).

Implementation: Implemented as the field annotation `@Inverse`. Its value must be a well-formed URI (referencing the inverse property).

Example of use:

```
public class Person {
  ...
  @Inverse("data.pojo#child")
  private Person parent;
}
```

The `parent` field is mapped into:

```
<owl:ObjectProperty rdf:about="#parent">
  ...
  <owl:inverseOf rdf:resource="http://data.pojo#child">
</owl:ObjectProperty>
```

A.4 Property Restrictions

These properties are used to describe classes of all individuals that satisfy some restriction on their property. There are two types of restrictions in OWL – value constraints and cardinality constraints. Value constraints define range of a property, they specify possible values the property can acquire. Cardinality constraints specify number of occurrences of the property within the restriction class.

These statements define constraints on properties in the context of the restriction class only. The property concerned has no constraints outside this class. There are also some global property restrictions, like `rdfs:range` or `owl:FunctionalProperty`. By way of contrast, these restrictions are applied wherever the property concerned is used.

Property restrictions are defined inside a restriction class (`owl:Restriction`) which is usually anonymous. This class contains also the `owl:onProperty` element which determines the restricted property. The anonymous restriction class can be used afterwards e.g. as a superclass of another class for which we want to use the restriction.

owl:hasValue

This property is a value constraint. It says that at least one value of the property concerned must be semantically equal to value V. The value V can be either an individual or a data value.

Implementation: Implemented as field annotation `@HasValue`. Its argument is either well-formed URI referencing the range class, or a simple data value. In the generated ontology the Java class containing the annotated field inherits from an anonymous restriction.

For example:

```
public class Person {
  ...
  @HasValue(stringValue="Jakub")
  private String givenname;
}
```

This class is mapped into:

```
<owl:Class rdf:about="#Person">
  ...
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#givenname" />
      <owl:hasValue
          rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >Jakub</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This class describes only those persons whose name is Jakub (or one of their names, if they have more than one).

owl:allValuesFrom, owl:someValuesFrom

These properties are value constraints, they give ranges to the property under consideration, but unlike the `rdfs:range` property these ones concern only the restriction class. `Owl:allValuesFrom` says that all values of the property concerned must belong to a defined range. `Owl:someValuesFrom` is less restrictive – it states that at least one value of the property under consideration has the defined range. The range itself can be either a class or a data range.

Implementation: Implemented as field annotations `@AllValuesFrom` and `@SomeValuesFrom`. Their arguments are either well-formed URIs referencing the range class, or enumerations of simple data values. In the generated ontology document the Java class containing the annotated field inherits from an anonymous restriction.

Example of value constraint usage follows:

```
public class Person {
  ...
  @AllValuesFrom(charValues={'M', 'F'})
  private char gender;

  @SomeValuesFrom("http://an.ontology#Dog")
  private Set<Animal> pets;
}
```

This class is mapped into:

```
<owl:Class rdf:about="#Person">
  ...
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#gender" />
      <owl:allValuesFrom>
        <owl:DataRange>
          <owl:oneOf rdf:parseType="Resource">
            <rdf:rest rdf:parseType="Resource">
              <rdf:rest rdf:resource="&rdf;nil"/>
              <rdf:first rdf:datatype="&xsd;string">F</rdf:first>
            </rdf:rest>
            <rdf:first rdf:datatype="&xsd;string">M</rdf:first>
          </owl:oneOf>
        </owl:DataRange>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#pets" />
      <owl:someValuesFrom rdf:resource="http://an.ontology#Dog"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This class describes those persons who have at least one dog (and can have other animals). Their gender can be only 'M' (male) or 'F' (female).

owl:cardinality, owl:maxCardinality, owl:minCardinality

These elements are cardinality constraints. `Owl:cardinality` indicates that all individuals of the restriction class have exactly N different values of the property concerned. `Owl:maxCardinality` restricts the count of different values from above, `owl:minCardinality` from below. If `owl:maxCardinality` is used in combination with `owl:minCardinality`, it defines an interval to which the number of the property's different values must belong. `Owl:cardinality` has the same meaning as using both `owl:maxCardinality` and `owl:minCardinality` with the same value of N.

Implementation: Implemented as field annotations `@Cardinality`, `@MaxCardinality` and `@MinCardinality` with integer arguments.

Problem of this implementation consists in the difference between an object code and a RDF-based ontology. We can say that some property's cardinality is 5 using `@Cardinality(5)` for the field under consideration. This statement will appear in the ontology document after the transformation process. But we can't create instances in Java that correspond to this statement (i.e. every instance has exactly 5 different values of that field at the same time). That is why these annotations should be used only for collections or arrays. They restricts the "number of elements" of the collection. If used for other types, the only meaningful values are 0 and 1.

For example:

```
public class Person {
  ...
  @MaxCardinality(4)
  @MinCardinality(2)
  private Set<Person> children;
}
```

This class is mapped into:

```
<owl:Class rdf:about="Person">
  ...
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#children" />
      <owl:maxCardinality
        rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >4</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#children" />
      <owl:minCardinality
        rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >2</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This class describes those persons that have 2, 3 or 4 children.

A.5 Annotation Properties

Properties from this group can be used for any OWL resource. Mostly they provide some additional information about the resource. Except for `owl:versionInfo` they are not used by machines, but they are intended for a human reader. They are implemented for classes and properties in the form of Java annotations. They can be used in the ontology header as well using appropriate methods in the `Ontology` class.

owl:versionInfo

This property is used for versioning. Its object is a literal that gives some information about its subject's version. It is used primarily for ontologies.

Implementation: For an ontology header implemented within the `Ontology` class as its method `setVersionInfo()`. For classes and properties implemented as the `@VersionInfo` annotation. Both method and annotation have a string argument that describes the version.

Example of use:

```
Ontology ontology = new Ontology("http://kiv.zcu.cz/eegbase");
ontology.setVersionInfo("v 2.0 - 5 Feb 2012");
...
```

This adds following ontology header:

```
<owl:Ontology rdf:about="http://kiv.zcu.cz/eegbase">
  <owl:versionInfo>v 2.0 – 5 Feb 2012</owl:versionInfo>
  ...
</owl:Ontology>
```

rdfs:comment

This element is used to provide a human-readable description of its subject.

Implementation: For an ontology header implemented as the `setComment()` method within the `Ontology` class. For classes and properties can be used the `@Comment` annotation. Both method and annotation have a string value with a textual information about the resource.

Example of use:

```
@Comment(value="Class of persons registered in the Portal."
        lang="en")
public class Person { ... }
```

This class is mapped into:

```
<owl:Class rdf:about="#Person">
  <rdfs:comment xml:lang="en">Class of persons registered in
    the Portal.</rdfs:comment>
</owl:Class>
```

rdfs:isDefinedBy

This property is used to set a reference to a resource that defines the resource concerned. It is a subproperty of `rdfs:seeAlso`.

Implementation: For an ontology header implemented as the `setIsDefinedBy()` method within the `Ontology` class. For classes and properties can be used the `@IsDefinedBy` annotation. The argument is a URI referencing some resource.

Example of use:

```
@IsDefinedBy("http://kiv.zcu.cz/eegbase")
public class Person { ... }
```

This class is mapped into:

```
<owl:Class rdf:about="#Person">
  <rdfs:isDefinedBy rdf:resource="http://kiv.zcu.cz/eegbase"/>
  ...
</owl:Class>
```

rdfs:label

This element is used to provide a human-readable name of its subject. It is useful for a human reader to understand its meaning better, especially if some resource is not very transparently named.

Implementation: For an ontology header implemented as the `setLabel()` method within the `Ontology` class. For classes and properties can be used the `@Label` annotation. Both method and annotation have a string argument that names the resource.

Example of use:

```
@Label("Parameters of measuration")
public class MeasurationAdditionalParams { ... }
```

This class is mapped into:

```
<owl:Class rdf:about="#MeasurationAdditionalParams">
  <rdfs:label>Parameters of measuration</rdfs:label>
  ...
</owl:Class>
```

rdfs:seeAlso

This element gives a reference to a resource that can provide some relevant information. It can be a Web page for example.

Implementation: For an ontology header implemented as the `setSeeAlso()` method within the `Ontology` class. For classes and properties can be used the `@SeeAlso` annotation. Their argument is a URI referencing some relevant resource.

Example of use:

```
Ontology ontology = new Ontology("http://kiv.zcu.cz/eegbase");
ontology.setSeeAlso("http://eegdatabase.kiv.zcu.cz");
...
```

This adds following ontology header:

```
<owl:Ontology rdf:about="http://kiv.zcu.cz/eegbase">
  <rdfs:seeAlso rdf:resource="http://eegdatabase.kiv.zcu.cz"/>
  ...
</owl:Ontology>
```

A.6 Individuals

Following properties are used to describe individuals. This poses a problem for the annotation-based approach, because in the static code we can annotate only static structures like classes, fields or methods. Individual Java instances, which are mapped into OWL individuals, cannot be provided with different annotation values.

Classes and properties can be treated as individuals in OWL Full, that is why we can use this group of properties for classes and properties. However, that implies that the resulting ontology document will be in OWL Full, which has no computational guarantees. That means for example that a further processing by a reasoning software can be very demanding or problematic. Therefore OWL Full is not supported in the Portal, but OWL DL. However, I have implemented this group of properties in the JenaBean Extension library as class or field annotations, but use of them is deprecated until the Portal supports OWL Full.

owl:AllDifferent, owl:differentFrom

These elements indicate that given individuals are not the same. `Owl:differentFrom` is a property, it describes the relation for two individuals. `Owl:AllDifferent` is a built-in class that defines a group of mutually different individuals. It is a more convenient way for a number of different individuals.

Implementation: Implemented as the class and field annotations `@AllDifferent` and `@DifferentFrom`. Their argument is a URI or an array of URIs that refer to different individuals.

owl:sameAs

This property states that two individuals are equal. It is the opposite of `owl:differentFrom`. We can use it to express that two different URIs refer to the same thing.

Implementation: Implemented as the class and field annotation `@SameAs`. Its argument is a URI of the equal individual.

Appendix B: Listings

```xml
<?xml version="1.0"?>
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:semantic="http://thewebsemantic.com#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xml:base="http://kiv.zcu.cz/eegbase#">
    <owl:Ontology rdf:about="http://kiv.zcu.cz/eegbase">
        <owl:versionInfo>10 April 2012</owl:versionInfo>
        <owl:backwardCompatibleWith rdf:resource="http://kiv.zcu.cz/eegdatabase"/>
        <owl:incompatibleWith rdf:resource="http://kiv.zcu.cz/ontology"/>
        <owl:priorVersion rdf:resource="http://kiv.zcu.cz/eegdatabase"/>
        <rdfs:comment>This ontology contains data from EEG/ERP experiments.</rdfs:comment>
        <rdfs:label>EEG/ERP Database</rdfs:label>
    </owl:Ontology>
    <owl:Class rdf:ID="Person">
        <rdfs:label xml:lang="cs">Osoba</rdfs:label>
        <rdfs:comment xml:lang="en">Class of persons registered in the Portal.</rdfs:comment>
        <semantic:javaclass>cz.zcu.kiv.eegdatabase.data.pojo.Person</semantic:javaclass>
    </owl:Class>
    <owl:Class rdf:ID="Experiment">
        <semantic:javaclass>cz.zcu.kiv.eegdatabase.data.pojo.Experiment</semantic:javaclass>
    </owl:Class>
    <owl:Class rdf:ID="Disease">
        <semantic:javaclass>cz.zcu.kiv.eegdatabase.data.pojo.Disease</semantic:javaclass>
    </owl:Class>
    <owl:Class rdf:ID="ScenarioType">
        <semantic:javaclass>cz.zcu.kiv.eegdatabase.data.pojo.ScenarioType</semantic:javaclass>
        <rdfs:subClassOf rdf:resource="http://kiv.zcu.cz/eegbase#IScenarioType"/>
    </owl:Class>

    ...

    <owl:ObjectProperty rdf:ID="experimentsForOwnerId">
        <rdfs:domain rdf:resource="http://kiv.zcu.cz/eegbase#Person"/>
        <rdfs:range rdf:resource="http://kiv.zcu.cz/eegbase#Experiment"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="electrodeTypes">
        <rdfs:domain rdf:resource="http://kiv.zcu.cz/eegbase#ResearchGroup"/>
        <rdfs:range rdf:resource="http://kiv.zcu.cz/eegbase#ElectrodeType"/>
    </owl:ObjectProperty>

    ...

    <owl:DatatypeProperty rdf:ID="startTime">
        <rdfs:domain rdf:resource="http://kiv.zcu.cz/eegbase#Experiment"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#dateTime"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:ID="givenname">
        <rdfs:domain rdf:resource="http://kiv.zcu.cz/eegbase#Person"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </owl:DatatypeProperty>

    ...

    <owl:AnnotationProperty rdf:about="http://thewebsemantic.com#javaclass">
        <rdfs:comment>
            This property determines the Java class that was mapped into declaring resource.
        </rdfs:comment>
        <rdfs:label>Java class</rdfs:label>
    </owl:AnnotationProperty>
</rdf:RDF>
```

Listing B.1: Illustration fragment of the ontology schema from the Portal (in RDF/XML-ABBREV).

```xml
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY this 'http://kiv.zcu.cz/eegbase#'>
                   <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
                   <!ENTITY xsd 'http://www.w3.org/2001/XMLSchema#'>
                   <!ENTITY owl 'http://www.w3.org/2002/07/owl#'>
                   <!ENTITY semantic 'http://thewebsemantic.com#'>
                   <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>]>
<rdf:RDF
      xmlns:rdf="&rdf;"
      xmlns:semantic="&semantic;"
      xmlns:owl="&owl;"
      xmlns:xsd="&xsd;"
      xmlns="&this;"
      xmlns:rdfs="&rdfs;"
    xml:base="&this;" >
    <rdf:Description rdf:about="&this;Experiment_1477909621">
        <subjectGroup rdf:resource="&this;SubjectGroup_0"/>
        <privateExperiment rdf:datatype="&xsd;integer">0</privateExperiment>
        <digitization rdf:resource="&this;Digitization_0"/>
        <rdf:type rdf:resource="&this;Experiment"/>
        <startTime rdf:datatype="&xsd;dateTime">2010-11-27T12:00:00Z</startTime>
        <personByOwnerId rdf:resource="&this;Person_0"/>
        <weather rdf:resource="&this;Weather_0"/>
        <endTime rdf:datatype="&xsd;dateTime">2010-11-27T12:30:00Z</endTime>
        <researchGroup rdf:resource="&this;ResearchGroup_1589011725"/>
        <experimentId rdf:datatype="&xsd;integer">122</experimentId>
        <personBySubjectPersonId rdf:resource="&this;Person_0"/>
        <scenario rdf:resource="&this;Scenario_0"/>
        <temperature rdf:datatype="&xsd;integer">25</temperature>
        <histories rdf:resource="&this;History_2142899520"/>
    </rdf:Description>
    <rdf:Description rdf:about="&this;History_269925795">
        <scenario rdf:resource="&this;Scenario_0"/>
        <person rdf:resource="&this;Person_929761472"/>
        <historyId rdf:datatype="&xsd;integer">310</historyId>
        <dateOfDownload rdf:datatype="&xsd;dateTime">2011-04-21T12:57:05Z</dateOfDownload>
        <rdf:type rdf:resource="&this;History"/>
    </rdf:Description>
    <rdf:Description rdf:about="&this;Weather_0">
        <defaultNumber rdf:datatype="&xsd;integer">0</defaultNumber>
        <weatherId rdf:datatype="&xsd;integer">0</weatherId>
        <scn rdf:datatype="&xsd;integer">0</scn>
        <rdf:type rdf:resource="&this;Weather"/>
    </rdf:Description>
    <rdf:Description rdf:about="&this;Disease_983945598">
        <title rdf:datatype="&xsd;string">blindnesss</title>
        <experiments rdf:resource="&this;Experiment_376897125"/>
        <diseaseId rdf:datatype="&xsd;integer">31</diseaseId>
        <description rdf:datatype="&xsd;string">blindnesss</description>
        <rdf:type rdf:resource="&this;Disease"/>
    </rdf:Description>
    <rdf:Description rdf:about="&this;Hardware_2009202871">
        <description rdf:datatype="&xsd;string">Usporny procesor</description>
        <hardwareId rdf:datatype="&xsd;integer">71</hardwareId>
        <experiments rdf:resource="&this;Experiment_1322182163"/>
        <type rdf:datatype="&xsd;string">Sandy Bridge</type>
        <defaultNumber rdf:datatype="&xsd;integer">1</defaultNumber>
        <title rdf:datatype="&xsd;string">Intel 2100 T</title>
        <researchGroups rdf:resource="&this;ResearchGroup_1072752492"/>
        <hardwareGroupRels rdf:resource="&this;Hardware_2009202871"/>
        <researchGroups rdf:resource="&this;ResearchGroup_54927975"/>
        <experiments rdf:resource="&this;Experiment_966434611"/>
        <rdf:type rdf:resource="&this;Hardware"/>
    </rdf:Description>

    ...

</rdf:RDF>
```

Listing B.2: Illustration fragment of the ontology document from the Portal (in RDF/XML).