Assignment VI

Creating a simple terminology knowledge base using SKOS

The goal of this assignment is manifold.

- We will use Protegé for knowledge modelling. If you are not familiar with it, it will be an occasion to practice it
- We will use the OWL APIs to manipulate an ontology programmatically
- We will create a "toy" knowledge base, learning some principles in the process, which will be useful for further exercises.

Task 0 – Setup

Prepare a maven module in your github repo. Use the usual namespace, and "homework6" as the artifact id.

Task I – SKOS++

SKOS provides a convenient framework to describe concepts (as opposed to classes). However, it only defines very general semantic relationships. This is done on purpose, so we will extend it.
For example, it does not define the common relationships "subTypeOf" and "partOf", so we will do it as an exercise.

Using Protegé (version 4 or 5, downloadable from here), let's create a new ontology. A new ontology requires a URL – use this pattern:
http://edu.asu.bmi/cad591/[your name]/skos/ext

Then, import the standard SKOS ontology using the "direct import" capability of Protegé. Import the ontology from a URL on the web. The URL of the SKOS ontology is this:
http://www.w3.org/2009/08/skos-reference/skos.rdf

Now you should see the concepts and relationships in SKOS.
Go to the object property tab and locate the relationship "broaderTransitive". Our relationships, "subTypeOf" and "partOf", are sub-properties of that. They are transitive, and they imply a "broaderThan" relationships. That is:
A subTypeOf B => A broaderTransitive B
A partOf B => A broaderTransitive B
Not the other way around : broaderTransitive is more generic, so it could mean other things.

- Add the two relationships as sub-properties of broaderTransitive, set the domain and range appropriately (skos:Concept) and mark the properties as "transitive".

Now, we'll take a chance and create an instance of skos:ConceptScheme. Go to the "Individuals" panel and:

- Create a new individual. Call it "bmiVocabulary", set its type to skos:ConceptScheme and set the data property "skos:notation" to "bmi".

Notice that now the individual has a URI – its unique identifier – and a "label / display name" - "bmi".

This scheme will be our “terminology system”, like SNOMED or LOINC...

Now save the ontology in the src/main/resources folder of your project.

Task II – OWL API basics

The OWL API allow to manipulate an OWL ontology from Java. In order to use them, add these dependencies to your project

```
<dependency>
 <groupId>net.sourceforge.owlapi</groupId>
 <artifactId>owlapi-apibinding</artifactId>
 <version>3.5.0</version>
</dependency>
<dependency>
 <groupId>net.sourceforge.owlapi</groupId>
 <artifactId>owlapi-api</artifactId>
 <version>3.5.0</version>
</dependency>
```

We will also need a DL reasoner, so add HermiT too:

```
<dependency>
 <groupId>com.hermit-reasoner</groupId>
 <artifactId>org.semanticweb.hermit</artifactId>
 <version>1.3.8.2</version>
</dependency>
```

The OWL API use three main classes: the OWLOntology itself, the OWLOntologyManger – which is responsible for loading, updating and saving an OWLOntology – and the OWLDataFactory – which allows to create new assertions to add to the OWLOntology.
The documentation on how to use the OWL APIs, complete with several code examples, can be found here: http://owlapi.sourceforge.net/documentation.html

To begin with, try the current snippet

```
OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
OWLOntology ontology = manager.loadOntologyFromOntologyDocument(
        // the relative path to your file, where "/" corresponds to
           src/main/resources
        this.getClass().getResourceAsStream( "/test.owl" )
);
OWLDataFactory factory = manager.getOWLDataFactory();
manager.saveOntology( ontology, System.err );
```

Once an ontology has been loaded, we can start manipulating it. We will add “individuals” (instances) to the ontology – in particular, instances of skos:Concept.
Notice that everything (individuals, properties, classes, etc..) are denoted using IRIs – internationalized

resource identifiers. These are special symbols (http://en.wikipedia.org/wiki/Internationalized_resource_identifier) that identify entities on the web. An IRI has a namespace and a local part or "fragment". The namespace will be the same as your ontology URI, while the local name will vary.

So, the IRI http://edu.asu.bmi/cad591/davide/skos/ext#mySomething has a namespace http://edu.asu.bmi/cad591/davide/skos/ext# and a fragment mySomething.

Notice that namespaces can be associated to prefixes, e.g. "bmi591" and so IRIs can be represented in the more compact form bmi591:mySomething. See the OWLAPI docs on how to configure prefixes programmatically. Notice also that, usually, an ontology will use multiple prefixes to denote entities/classes/properties coming from multiple sources. For example, our ontology is already using the namespace http://www.w3.org/2004/02/skos/core, with (usual) prefix "skos"

We can now populate our ontology using the OWL APIs:

```
static final String BMI591 = "http://edu.asu.bmi/cad591/davide/skos/ext#";

// instantiates OR looks up an individual with the given IRI
OWLNamedIndividual ind = factory.getOWLNamedIndividual(
      IRI.create( BMI591 + name )
);

// instantiates a relationship between two individuals
OWLObjectPropertyAssertionAxiom opa = factory.getOWLObjectPropertyAssertionAxiom(
      factory.getOWLObjectProperty( IRI.create(  ) ),  // the property
      factory.getOWLNamedIndividual( IRI.create(  ) ), // the subject
      factory.getOWLNamedIndividual( IRI.create(  ) )  // the object
);

// instantiates a data property ("attribute") of an individual
OWLDataPropertyAssertionAxiom oda = factory.getOWLDataPropertyAssertionAxiom(
        factory.getOWLDataProperty( IRI.create(  ) ),
        factory.getOWLNamedIndividual( IRI.create(  ) ),
        value
);
```

The individuals, as well as the object/data property assertions, do not get added directly to the ontology. We need to add them explicitly:

```
manager.addAxiom( ontology, factory.getOWLDeclarationAxiom( ind ) );
manager.addAxiom( ontology, opa );
manager.addAxiom( ontology, oda );
```

As you see, the code is quite verbose and repetitive. Try building your own helper functions to simplify the authoring.

Task III – Modelling a taxonomy

After familiarizing with the tools, we will now assert some coded concepts in the BMI591 vocabulary. To this end, let's remember that a coded concept needs

- a unique IRI in the BMI591 namespace

- to be part of the BMI591 vocabulary
  factory.getOWLNamedIndividual(BMI591 + "bmiVocabulary") will refer to the individual that represents the vocabulary. This is possible because we are using the same IRI.
  We will need to assert the object property skos:inScheme between each of our concepts and this vocabulary
- a "code", which can be asserted using the data property skos:notation
- one or more labels (optional, use the OWL Annotation Properties skos:prefLabel and skos:altLabel)
- one or more super-concepts. Use the object property skos:broaderTransitive or bmi591:subConceptOf
- one or more domain-specific relationships to other attributes (e.g. bmi591:partOf)

To this end, you will create helper methods such as:

```
public void addCodedConcept(
 IRI id,
 String code,
 List<String> labels,
 List<IRI> parents,
 OWLOntology ontology
)

public void addRelationship(
 IRI subject,
 IRI property,
 IRI object,
 OWLOntology ontology
)
```

Now use these methods to populate your ontology and save it. Mimic a small piece of the SNOMED-CT ontology: go to http://browser.ihtsdotools.org/ and pick a "root" concept. Model it appropriately using the helper methods you have just created. Notice that we are "importing" these concepts in the bmi591 vocabulary by changing the namespace and the containing vocabulary. In a real system, we would preserve the SNOMED-CT namespace, and we would model SNOMED itself as a skos:CodingScheme.

Now pick up to 5 of its descendants of your root concept and model them as well. Assert the "isA" relationships and make sure that the hierarchy has at least 2 levels.
Finally, make sure that at least one of the concepts has at least one "attribute" - which in SNOMED is a relationship to another concept. These related concepts will have to modelled (code, labels), but ignore their hierarchy and second-degree attributes, if any.

Task III – Asserted vs Inferred Ontology

At this point, you have enriched the original ontology with new concepts. This is called the "asserted" ontology. These assertions may imply additional knowledge, which can be inferred by a "Reasoner", creating the "inferred" ontology. To do so, a Reasoner must be invoked. You can read how to do it in the OWL API docs. There are many variations. The simplest way is to use the helper provided here:

https://github.com/BMI-591-CAD-2015/MockStudent/blob/master/terms-demo/src/main/java/edu/asu/

bmi/cad591/ReasonerHelper.java

Task III – Putting it all together

Now we will create a Junit test case to prove the necessity of the inference mechanism.

Create a method getPopulatedOntology() that returns the ontology up to the end of Task II.

Go back to your taxonomy and pick two concepts: one must be a descendant of the other, but not an immediate child. Let's call “ancestor” the former and “descendant” the latter.

We need to prove that the “bmi591:subTypeOf(descendant, ancestor)” relationship (or skos:broaderTransitive) is not (yet) part of the ontology. You can adapt the method checkRelationship here:

https://github.com/BMI-591-CAD-2015/MockStudent/blob/master/terms-demo/src/test/java/edu/asu/bmi/cad591/AxiomTester.java

passing a boolean argument that will control whether assertTrue or assertFalse is checked at the end.

At this point, checkRelationship should succeed on assertFalse.

Now invoke the reasoner, then check for the presence of the relationship. It should be there.
That is, the following should succeed:

```
// the code may vary slightly depending on how you adapt the example classes
axiomTester.checkRelationship( descendant, “subTypeOf”, ancestor, false );
new ReasonerHelper().makeInferences( ontology );
axiomTester.checkRelationship( descendant, “subTypeOf”, ancestor, true );
```

Now that you are familiar with SKOS and the OWL API, you can look at the implementation here: http://skosapi.sourceforge.net/

Now consider this: a real world terminology contains thousands of codes. If not directly represented in OWL (or SKOS), it would probably distributed in a spreadsheet format – like LOINC, for example. In that case, you would parse the spreadsheet and invoke your helper methods from within a loop, rather than manually passing constant values.
XLS files can be conveniently parsed using the POI library (http://poi.apache.org/). Of course, you are not expected to parse LOINC in this assignment, but remember that this is an option...