Léo POTIERS

Paul SERONIE-VIVIEN

December 2020

5AE - ISS

Semantic Data:
Lab Report

Introduction

In the framework of this lab, we will concretely understand and manipulate the notion of ontology. For this, we will use data from weather sensor observations from the city of Aarhus in Denmark. In the first part of this practical work, we will create an ontology that will aim to semantically describe the data so that it can determine the meteorological phenomenon (rain, storm, ...), the associated measurable parameters (temperature, humidity, ...) or even which sensor has made this observation. To create the ontology, we will use the Protégé software which has an integrated reasoner "Hermit".

Ontology creation

1. Lightweight ontology

a. Conception

In this first case, we will use the ontology to list concepts of the meteorological field. This is what we call a lightweight ontology.

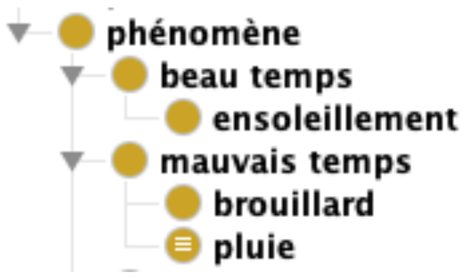
First, we are going to express various information by translating the ***next sentences*** into the suited concepts:

1. Le beau temps et le mauvais temps sont deux types de phénomènes.

Under <owl: Thing>, we created a class "Phénomène" with two subclasses "beau temps" and "mauvais temps".

2. La pluie et le brouillard sont des types de phénomènes de mauvais temps, l'ensoleillement est un type de phénomène de beau temps.

As we just did, we are going to create two subclasses of "mauvais temps" which are "pluie" and "brouillard" and one subclass of "beau temps" which is "ensoleillement".

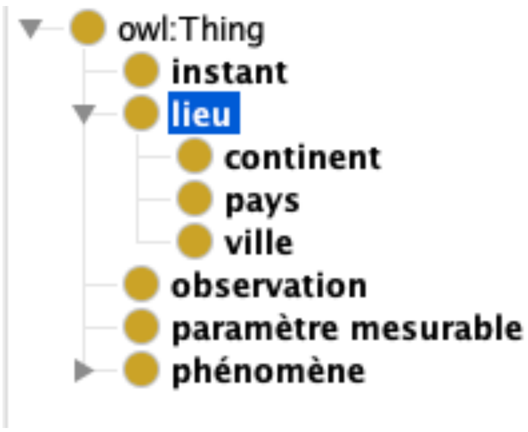


3. Les paramètres mesurables sont une classe de concept, ainsi que les instants et les observations

Here we need to create three more classes "paramètres mesurables", "instants" and "observations".

4. Une ville, un pays et un continent sont des types de lieux

Here we created another class "lieu" with three subclasses "continent", "pays" and "ville".

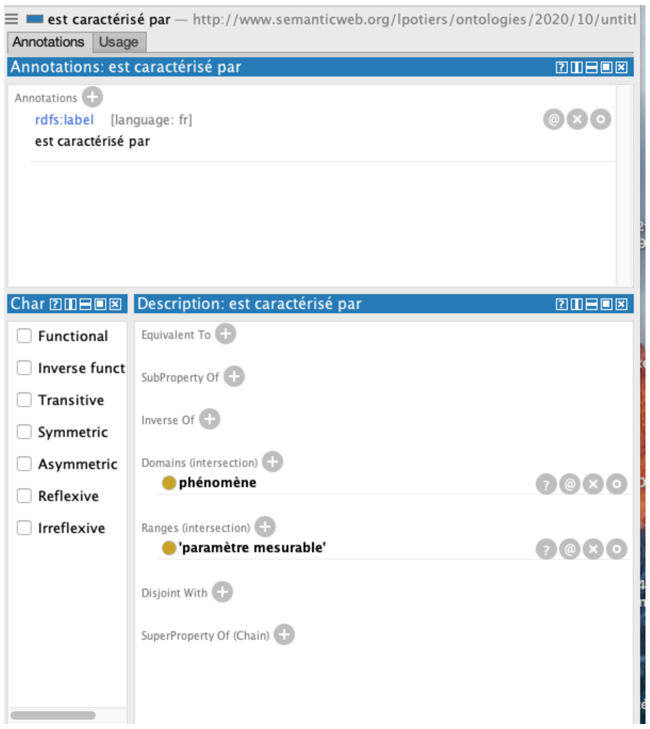


- owl:Thing
  - instant
  - lieu
    - continent
    - pays
    - ville
  - observation
  - paramètre mesurable
  - phénomène

Now that we have our classes, we are going to add proprieties linking these classes to each other. It is important to introduce here the concepts of Object and Data properties. An Object Property is a property which links two classes whereas a Data Property links a class and a value (with a specified type).

1. *Un phénomène est caractérisé par des paramètres mesurables*

Here we created the object property « est caractérisé par » which links the class "phénomène" to the class "paramètre mesurable".



est caractérisé par — http://www.semanticweb.org/lpotiers/ontologies/2020/10/untitl

Annotations | Usage

Annotations: est caractérisé par

Annotations
rdfs:label [language: fr]
est caractérisé par

Char

- [ ] Functional
- [ ] Inverse funct
- [ ] Transitive
- [ ] Symmetric
- [ ] Asymmetric
- [ ] Reflexive
- [ ] Irreflexive

Description: est caractérisé par

Equivalent To

SubProperty Of

Inverse Of

Domains (intersection)
phénomène

Ranges (intersection)
'paramètre mesurable'

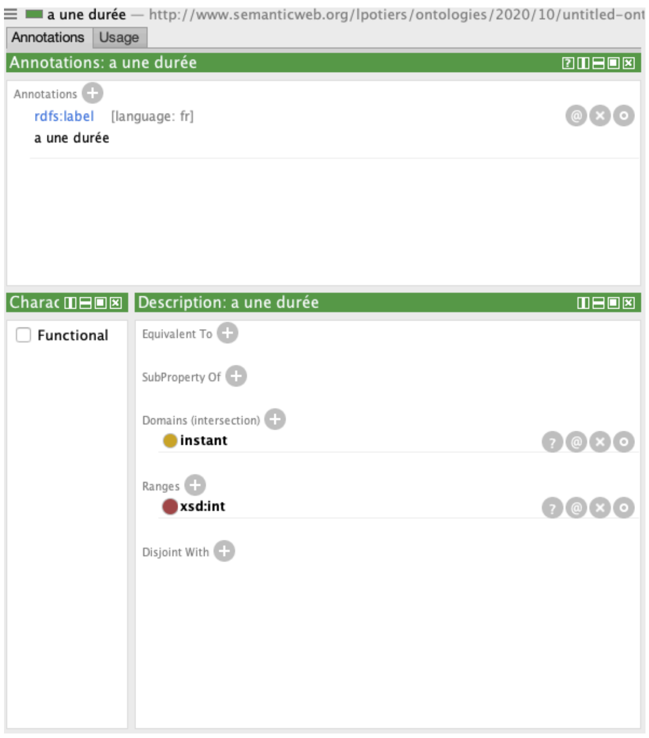
Disjoint With

SuperProperty Of (Chain)

The next sentences express object properties as well: ***3. Un phénomène débute à un instant - 4. Un phénomène finit à un instant - 6. Un phénomène a pour symptôme une observation - 7. Une observation météo mesure un paramètre mesurable. - 9. Une observation météo a pour localisation un lieu. - 10. Une observation météo a pour date un instant - 11. Un lieu peut être inclus dans un autre lieu - 12. Un lieu peut inclure un autre lieu - 13. Un pays a pour capitale une ville***

2. ***Un phénomène a une durée en minutes***

As explained just before, this is data property because with this information, we are going to indicate that an instance of the class "phénomène" will be linked to an integer thanks to the label "a une durée".


a une durée — http://www.semanticweb.org/lpotiers/ontologies/2020/10/untitled-ont

Annotations: a une durée

rdfs:label [language: fr]
a une durée

Description: a une durée

Functional: ☐

Domains (intersection): instant

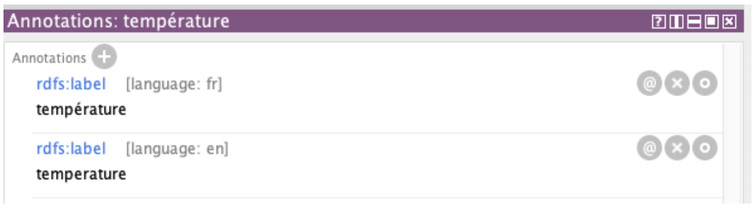
Ranges: xsd:int

The next sentences express as well data properties : ***5. Un instant a un timestamp, de type xsd :dateTimeStamp - 8. Une observation météo a une valeur pour laquelle vous ne représenterez pas l'unité***

b. Settlement

The settlement process enables to understand what the raisoner can deduct from the ontology we settled. In this part we are going to create instances of the classes and data properties we firstly have been creating.

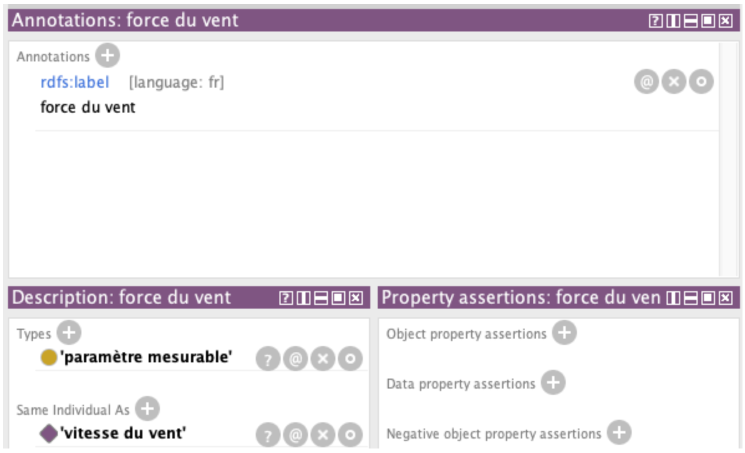
We implemented a translation of an instance by adding an annotation in another language:



Annotations: température

Annotations
rdfs:label [language: fr]
température
rdfs:label [language: en]
temperature

If two instances have the same meaning in the same language it is also possible to implement it into Procédé with the mention “Same individual as” :



Annotations: force du vent

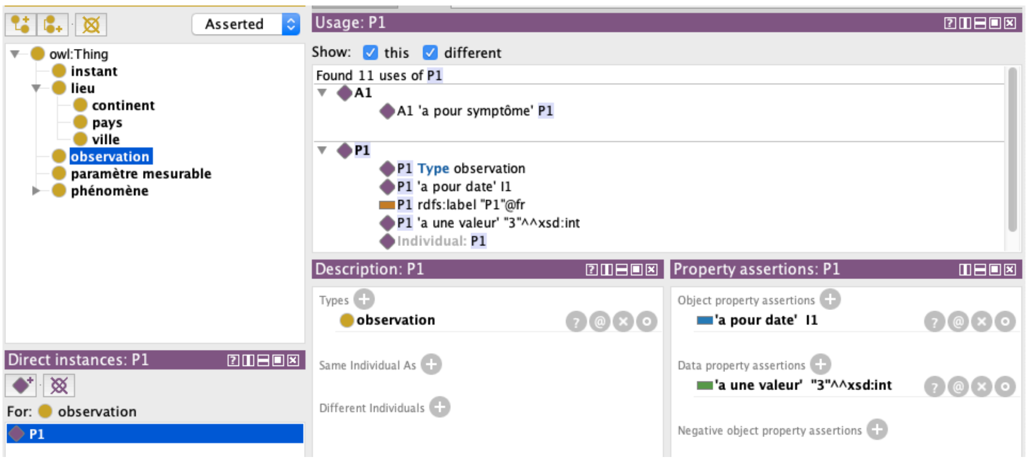
Annotations
rdfs:label [language: fr]
force du vent

Description: force du vent
Types
'paramètre mesurable'
Same Individual As
'vitesse du vent'

Property assertions: force du ven
Object property assertions
Data property assertions
Negative object property assertions

At this point, if we create a “ville” and a “pays” without specifying their class and we use the object property “être inclus dans” so that the city is included to the country (if so it is), then the reasoner will automatically set their type to “lieu” because it knows that the chosen object property links a “lieu” to another. We notice the very same behavior if we create Paris as the capital of France, there is no need to say its type because the reasoner will understand it alone thanks to the parameters of the object property used.

To simulate the measurement of a sensor we can implement a data property value and an object property relating to the date of the measure into the property assertions of the object “observation”:



Asserted
owl:Thing
instant
lieu
continent
pays
ville
observation
paramètre mesurable
phénomène

Usage: P1
Show: ☑ this ☑ different
Found 11 uses of P1
A1
A1 'a pour symptôme' P1
P1
P1 Type observation
P1 'a pour date' I1
P1 rdfs:label "P1"@fr
P1 'a une valeur' "3"^^xsd:int
Individual: P1

Description: P1
Types
observation
Same Individual As
Different Individuals

Property assertions: P1
Object property assertions
'a pour date' I1
Data property assertions
'a une valeur' "3"^^xsd:int
Negative object property assertions

Direct instances: P1
For: observation
P1

2. Heavyweight ontology

a. Conception

When the ontology needs to have some more complex reasoning, it will deal with more relations and properties to describe the concepts on the ones we want to reason, this is what we call a heavyweight ontology.

1. Toute instance de ville ne peut pas être un pays
In order to implemented this concept, we disjointed the class “ville” with the class “pays”.

2. Un phénomène court est un phénomène dont la durée est de moins de 15 minutes
We created « phénomène court » as an equivalent to a ”phénomène” with a data property “a une durée” valued under 15 minutes. We did implemented the same equivalency (but >15min) for a “phénomène long” and specified a disjunction between these both “phénomène”.

5. La propriété indiquant qu’un lieu est inclus dans un autre a pour propriété inverse la propriété indiquant qu’un lieu en inclue un autre.
To enable the reasoner to understand that some object properties could be reciprocally the same, we used the label “Inverse Of”. This way, it can deduce that is a “ville” is included into a “pays”, then we can also say that this “pays” includes the “ville”.

6. Si un lieu A est situé dans un lieu B et que ce lieu B est situé dans un lieu C, alors le lieu A est situé dans le lieu C
In this sentence we tackle the transitivity characteristic of an object property. We added it to “include” and to “être inclus dans”.

7. A tout pays correspond une et une seule capitale
If a property is injective, we need to check “Functional” into the object property characteristics. Just to know, the characteristic “Inverse Functional” means that the propriety is surjective.

8. Si un pays a pour capitale une ville, alors ce pays contient cette ville
Here we introduce the concept of “SubProperty of”, “a pour capitale” is a subproperty of “inclure”. Reciprocally, we can say that “inclure” is a SuperProperty Of “a pour capitale”.

9. La Pluie est un Phénomène ayant pour symptôme une Observation de Pluviométrie dont la valeur est supérieure à 0.
As explained before to distinguish a “phénomène court” or a “phénomène long”, we used the Manchester syntax to write “phénomène that 'a pour symptôme' some (observation that (mesure value pluviométrie) and ('a une valeur' some xsd:float [>0]))” into the “Equivalent To” part of the description of the Subclass “pluie”.

b. Settlement

In this part, as the following as the settlement of the lightweight ontology, we can see that now the reasoner understands and can deduce even more concepts. For example, that A1 belongs to the class “pluie” and that a country can not have more than one capital. If the creation of an individual goes against a specification, the reasoner returns an error.

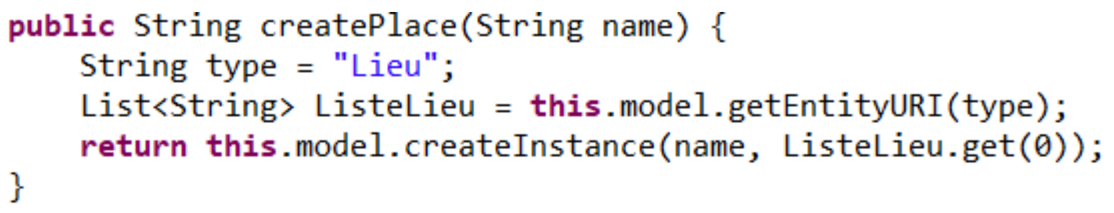
Exploitation of the ontology

In this second part, once our ontology was created, we started to implement functions that can be used for this ontology. In fact, the objective was to use the different properties that we defined earlier for all the classes to create a model that will represent some individuals from our ontology. The different functions that we were asked to implement are the following:

- ***createPlace()*** that creates an instance of the class « Lieu » in our ontologie and return the URI of this new individual. The only parameter this function is taking is the name of the place.
- ***createInstant()*** is really similar to the previous one but it creates an “Instant”. This function has also only one parameter that is of type “TimestampEntity”. It is important to notice that the new individual will not be created if there already is an “Instant” associated to the given timestamp.
- ***getInstantURI()*** returns the URI of the « Instant » given in parameter of the function, if it exists.
- ***getInstantTimestamp()*** returns the timestamp associated to the “Instant” that has been given as a parameter of the function. If it does not exist, the function returns null.
- ***createObs()*** allows to create an observation from the different elements that characterize it : its value, its « Instant » and the type of the parameter that is being observed. The function returns the URI of the created observation.

To help us implementing all these functions, several methods that have been listed in the ***IConvenienceInterface*** class were at our disposal to do everything we needed to manipulate the ontology (create instances, add some properties to an individual or get its URI). Every time in the code when we used ***this.model.****, we are referring to one of the function described in this class.

Before starting to treat the coding part of the previously described functions, it is important to notice that there is an important difference between the individuals that we created and their URIs. A URI can be associated to anything in the ontology, the properties, the classes, or the individuals. This has been extremely useful for the implementation of our functions to get access to the different URIs contained in the model. To do that, the function ***getEntityURI()*** was really helpful because it allowed us to get access to the URI of anything by just giving the label of the entity we were looking for. However, there is a little trick, you need to use ***get(0)*** after this function because it returns a list of String and the URI can be accessed doing that. Here is an example in the implementation of the function ***createPlace().***



```
public String createPlace(String name) {
    String type = "Lieu";
    List<String> ListeLieu = this.model.getEntityURI(type);
    return this.model.createInstance(name, ListeLieu.get(0));
}
```

The code above is a good example of the use of the function ***createInstance()*** that creates an individual of the provided type. The type is provided by giving the URI of the type as a parameter of the function. It returns the URI of the new individual and was perfect to implement the functions ***createPlace()*** and ***createInstant()***. Here is the code of our second function.

```java
public String createInstant(TimestampEntity instant) {

    String timeStamp = instant.getTimeStamp();
    String type = "Instant";
    List<String> ListeInstant = this.model.getEntityURI(type);
    String instantURI = this.model.createInstance("instant "+ timeStamp , ListeInstant.get(0));
    String propertyURI = this.model.getEntityURI("a pour timestamp").get(0);
    this.model.addDataPropertyToIndividual(instantURI, propertyURI, timeStamp);

    return instantURI;
}
```

In the code of the ***createInstant()*** function above, a new notion is introduced : the properties. In this case, we created an instance of the class “Instant”. However, this class is characterized by a timestamp and a property is defined to show that and is called “a pour timestamp”. Therefore we used the function ***addDataPropertyToIndividual()*** to “link” the timestamp associated to the new “Instant”. It was the case of a data property because a timestamp is only a String and not an entirely other class.

In the following function ***getInstantURI()***, we manipulated the previous property “a pour timestamp”. The function is looking at all the “Instant” to find the one that has the timestamp that has been given as a parameter of the function. If the property is correctly defined and has the good value, the function returns the URI of the correct “Instant”. The function ***hasDataPropertyValue()*** is used to do the checking. The code of the function is shown below.

```java
public String getInstantURI(TimestampEntity instant) {

    String timeStamp = instant.getTimeStamp();
    List<String> ListeinstantClassURI = this.model.getEntityURI("Instant");
    List<String> ListePropertyURI = this.model.getEntityURI("a pour timestamp");

    for (String Listeinstant : this.model.getInstancesURI(ListeinstantClassURI.get(0))) {
        if (this.model.hasDataPropertyValue(Listeinstant, ListePropertyURI.get(0), timeStamp)) {
            return Listeinstant;
        }
    }

    return null;
}
```

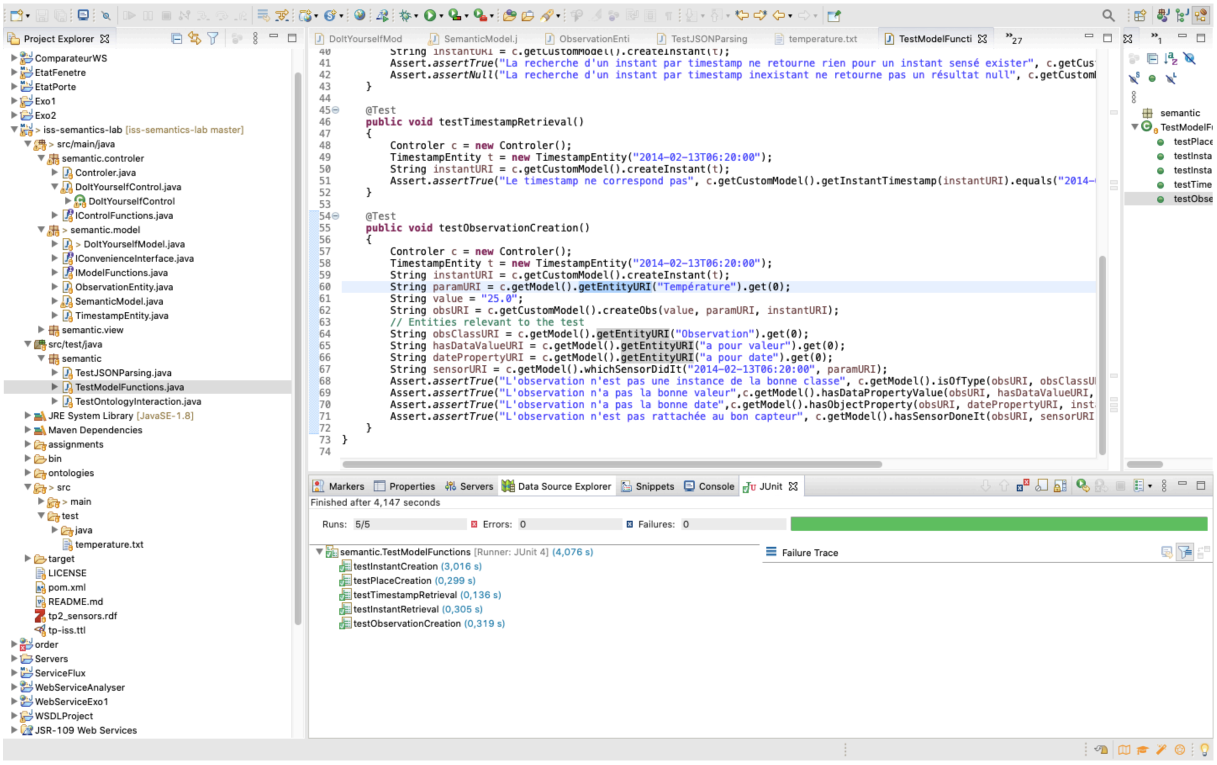
The next function ***getInstantTimestamp()*** is the opposite of the previous one, giving the timestamp associated to a given “Instant”. The new function that is used in this case is ***listProperties()*** that gives all the properties related to a certain entity. The code is shown down below, using a for loop to check if the property “a pour timestamp” exists and then returns the value of the timestamp.

```java
public String getInstantTimestamp(String instantURI)
{
    List<String> ListetimestampPropertyURI = this.model.getEntityURI("a pour timestamp");
    for(List<String> ListeProperties : this.model.listProperties(instantURI))
    {
        if(ListeProperties.get(0).equals(ListetimestampPropertyURI.get(0)))
        {
            return ListeProperties.get(1);
        }
    }
    return null;

}
```

The last function we implemented, ***createObs()***, returns the URI of the observation that has been created by the function. The main difference with the previous “creator” functions is that an observation is way more complex due to the several properties it has. Then, the function is divided in two parts. The first one is made to get access to all the URIs that are needed to create an observation by using the ***getEntityURI()*** function and the ***getInstantTimestamp()*** function that we just created. The second part is where the links between all these URIs are created for the new observation (created at the beginning of the function). There is also a sensor that is obtained using the ***whichSensorDidIt()*** function so we can link the new operation to this sensor with the ***addObservationToSensor()*** function. The code is shown in the figure below.

```java
public String createObs(String value, String paramURI, String instantURI) {

    String observationClassURI = this.model.getEntityURI("Observation").get(0);
    String uri_instance = this.model.createInstance("obs_"+instantURI, observationClassURI);
    String uri_dataValue = this.model.getEntityURI("a pour valeur").get(0);
    String uri_instantProp = this.model.getEntityURI("a pour date").get(0);
    String uri_paramProp = this.model.getEntityURI("mesure").get(0);
    String timestamp = getInstantTimestamp(instantURI);
    String sensor = this.model.whichSensorDidIt(timestamp, paramURI);
    this.model.addObservationToSensor(uri_instance, sensor);
    this.model.addDataPropertyToIndividual(uri_instance, uri_dataValue, value);
    this.model.addObjectPropertyToIndividual(uri_instance, uri_instantProp, instantURI);
    this.model.addObjectPropertyToIndividual(uri_instance, uri_paramProp, paramURI);
    return uri_instance;

}
```

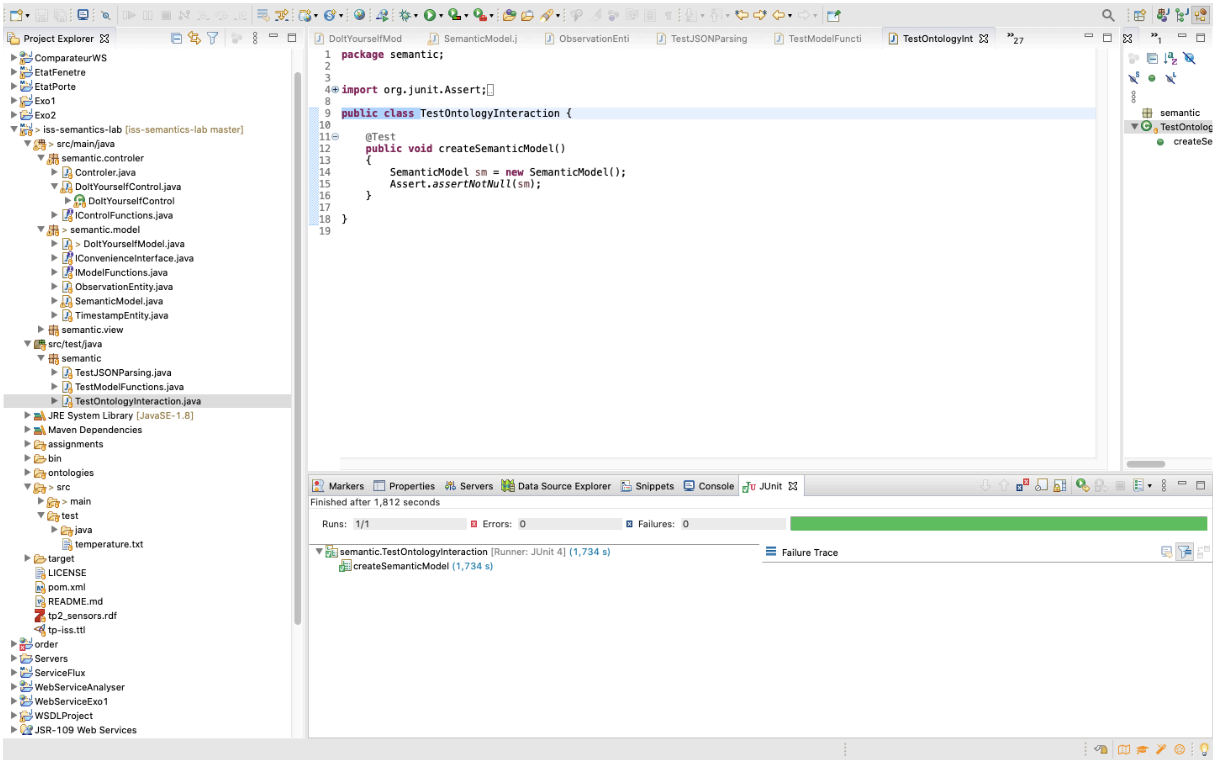
Once our five functions were coded, we ran the tests that were included in the project. The objective was to see whether our model was able to create properly different individuals of different classes. The results of the tests were all okay and are detailed in the figures below.



```
        String instantURI = c.getCustomModel().createInstant(t);
        Assert.assertTrue("La recherche d'un instant par timestamp ne retourne rien pour un instant sensé exister", c.getCus
        Assert.assertNull("La recherche d'un instant par timestamp inexistant ne retourne pas un résultat null", c.getCustom
    }

    @Test
    public void testTimestampRetrieval()
    {
        Controler c = new Controler();
        TimestampEntity t = new TimestampEntity("2014-02-13T06:20:00");
        String instantURI = c.getCustomModel().createInstant(t);
        Assert.assertTrue("Le timestamp ne correspond pas", c.getCustomModel().getInstantTimestamp(instantURI).equals("2014-
    }

    @Test
    public void testObservationCreation()
    {
        Controler c = new Controler();
        TimestampEntity t = new TimestampEntity("2014-02-13T06:20:00");
        String instantURI = c.getCustomModel().createInstant(t);
        String paramURI = c.getModel().getEntityURI("Température").get(0);
        String value = "25.0";
        String obsURI = c.getCustomModel().createObs(value, paramURI, instantURI);
        // Entities relevant to the test
        String obsClassURI = c.getModel().getEntityURI("Observation").get(0);
        String hasDataValueURI = c.getModel().getEntityURI("a pour valeur").get(0);
        String datePropertyURI = c.getModel().getEntityURI("a pour date").get(0);
        String sensorURI = c.getModel().whichSensorDidIt("2014-02-13T06:20:00", paramURI);
        Assert.assertTrue("L'observation n'est pas une instance de la bonne classe", c.getModel().isOfType(obsURI, obsClassU
        Assert.assertTrue("L'observation n'a pas la bonne valeur",c.getModel().hasDataPropertyValue(obsURI, hasDataValueURI,
        Assert.assertTrue("L'observation n'a pas la bonne date",c.getModel().hasObjectProperty(obsURI, datePropertyURI, inst
        Assert.assertTrue("L'observation n'est pas rattachée au bon capteur", c.getModel().hasSensorDoneIt(obsURI, sensorURI
    }
}
```

```
Finished after 4,147 seconds

Runs: 5/5    Errors: 0    Failures: 0

semantic.TestModelFunctions [Runner: JUnit 4] (4,076 s)
    testInstantCreation (3,016 s)
    testPlaceCreation (0,299 s)
    testTimestampRetrieval (0,136 s)
    testInstantRetrieval (0,305 s)
    testObservationCreation (0,319 s)
```

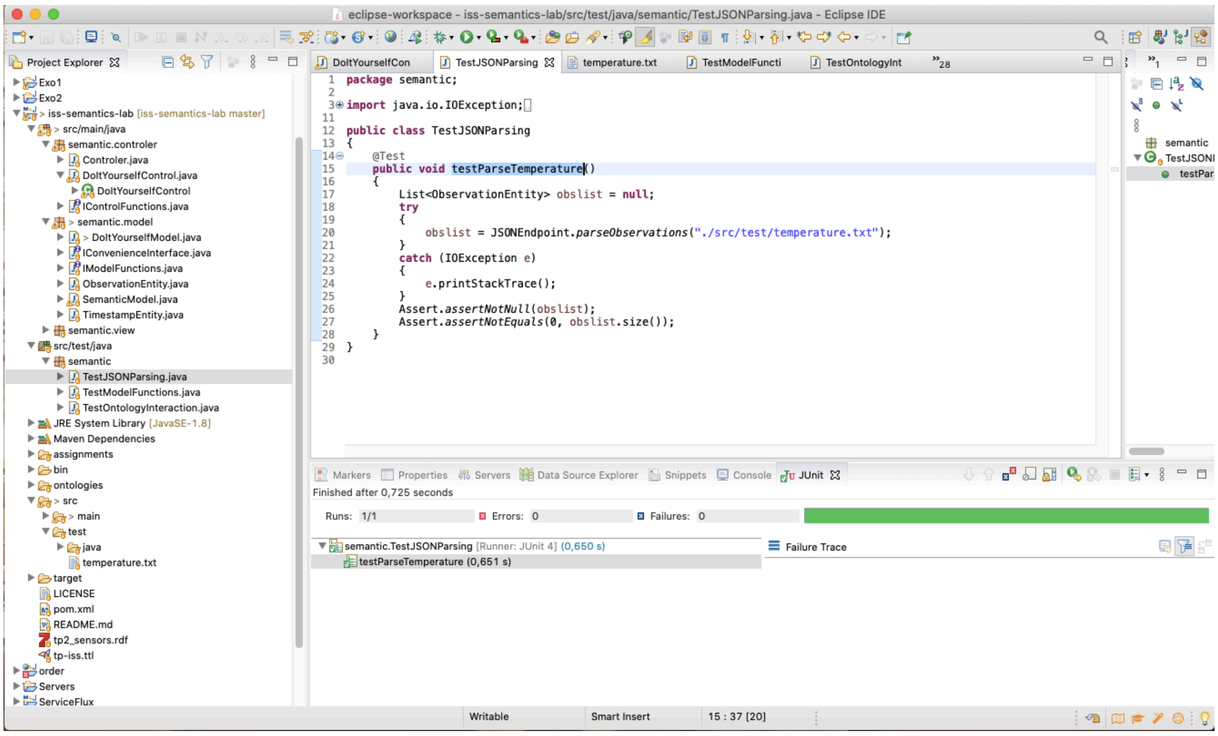
Figure 1 : Test of the functions of the model



```
package semantic;

import org.junit.Assert;

public class TestOntologyInteraction {

    @Test
    public void createSemanticModel()
    {
        SemanticModel sm = new SemanticModel();
        Assert.assertNotNull(sm);
    }
}
```

```
Finished after 1,812 seconds

Runs: 1/1    Errors: 0    Failures: 0

semantic.TestOntologyInteraction [Runner: JUnit 4] (1,734 s)
    createSemanticModel (1,734 s)
```

Figure 2 : Test of the interactions within the ontology



```java
package semantic;

import java.io.IOException;

public class TestJSONParsing
{
    @Test
    public void testParseTemperature()
    {
        List<ObservationEntity> obslist = null;
        try
        {
            obslist = JSONEndpoint.parseObservations("./src/test/temperature.txt");
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        Assert.assertNotNull(obslist);
        Assert.assertNotEquals(0, obslist.size());
    }
}
```

```
Finished after 0,725 seconds
Runs: 1/1    Errors: 0    Failures: 0
semantic.TestJSONParsing [Runner: JUnit 4] (0,650 s)
    testParseTemperature (0,651 s)
```

Figure 3 : Test of the JSON parsing

Conclusion

During this lab, we were able to understand the creation, use and specificities of an ontology. The use of the Protege software allowed us to understand the interaction that the different properties can cause within an ontology. It was also possible for us to understand the reasoning of a "reasoner" and to understand what certain definitions on individuals allow to deduce from other individuals.

The creation of functions that we performed during the second part of this lab was also very enriching to realize what is necessary to create the model of an ontology. Unfortunately, due to several technical issues while using and initializing the second part of this lab, we couldn't go beyond creating functions.