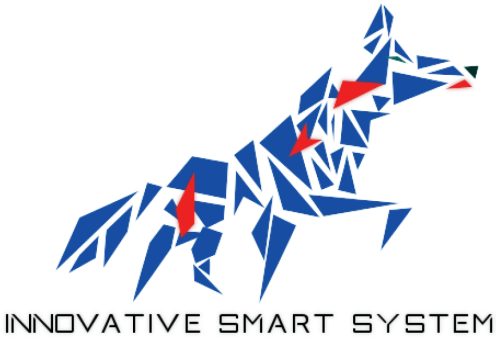
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

PTP Innovative Smart Systems - Semantic Data

Alex Noize - Mathieu Raynaud

Rapport TPs - Web sémantique



19 janvier 2019

Table des matières

1 Introduction

Dans le cadre de notre 5ème année de formation à l'INSA de Toulouse, spécialité Innovative Smart Systems (ISS), nous avons suivi le cours de Traitement des Données Sémantiques de l'Unité de Formation Analyse et traitement des données, applications métier.

C'est à la suite des deux Travaux Pratiques effectués au sein de ce cours que nous avons produit le présent rapport, faisant office de compte-rendu pour le travail que nous avons produit. L'objectif de ces TPs est d'implémenter une ontologie à l'aide de *Protégé* constituant un modèle de données dans le domaine des observations météorologiques.

La deuxième partie portera sur le développement d'un outil basé sur la libraire Java **Jena** permettant d'intégrer à notre ontologie des données CSV issues d'un open data set mis en ligne par la ville d'Aarhus, au Danemark.

2 Création de l'ontologie

Cette partie a pour but de définir de manière très simple une ontologie de la météo comprenant une notion de capteur. Pour cela, nous utiliserons l'éditeur *Protégé* qui nous permettra de mettre en évidences les différents constituants d'une ontologie. De plus, nous nous servirons du raisonneur *Hermit* pour voir les déductions qu'il est possible de faire à partir de la base de connaissance que nous avons créée.

2.1 L'ontologie légère

2.1.1 Conception

Nous avons représenté les connaissances décrites à la section 2.2.1 par la création des classes :

1. *Phénomène* contenant deux sous-classes *Beau_temps* et *Mauvais_temps*.
2. Deux sous-classes *Pluie* et *Brouillard* dans la classe *Mauvais_temps* et la création de la sous-classe *Ensoleillement* dans la classe *Beau_temps*.
3. Trois classes *Paramètres_mesurables*, *Instants* et *Observations*.
4. *Lieu* et de ses sous-classes *Ville*, *Pays* et *Continent*.

Et par l'ajout des propriétés :

1. *est_caractérisé_par*(*Phénomène* → *Paramètres_mesurables*) dans **Object Properties**
2. *a_une_durée*(*Phénomène*) de type **xsd :float** dans **Data Properties**
3. *débute*(*Phénomène* → *Instant*) dans **Object Properties**
4. *finit*(*Phénomène* → *Instant*) dans **Object Properties**
5. *timestamp* de type **xsd :dateTimeStamp** dans **Data Properties**
6. *a_pour_symptôme*(*Phénomène* → *Observation*) dans **Object Properties**
7. *mesure*(*Observation* → *Paramètre_mesurable*) dans **Object Properties**
8. *a_une_valeur*(*Observation*) dans **Data Properties**
9. *a_pour_localisation*(*Observation* → *Lieu*) dans **Object Properties**
10. *a_pour_date*(*Observation* → *Instant*) dans **Object Properties**

11. *est_inclus_dans*(*Lieu* → *Lieu*) dans **Object Properties**
12. *inclut*(*Lieu* → *Lieu*) dans **Object Properties**
13. *a_pour_capitale*(*Pays* → *Ville*) dans **Object Properties**

La différence entre les propriétés utilisées pour modéliser les questions 1.a et 1.b est le type de données impactées par ces propriétés.
D'une part, les **Object Properties** s'appliquent entre deux instances de classes afin de caractériser la relation qu'elles entretiennent (par exemple une ville est inclue dans un pays).
D'autre part, les **Data Properties** permettent de définir toute relation entre une instance de classe et une donnée pouvant lui être associée (par exemple un phénomène a une durée exprimée en nombre flottant de minutes).

2.1.2 Peuplement

Après avoir représenté la connaissance de la question 2.a par des **Individus** au sein de la table des **Individuals**, on constate que le raisonneur déduit les éléments suivants :

- A1 est un Phénomène
- Paris est une ville
- Paris est inclus dans France
- La France inclut Paris
- La France inclut Toulouse
- La France est un pays
- I1 est un instant

En utilisant les domain/range d'une data property, le raisonneur peut déterminer la classe d'un individu grâce aux données qu'il contient, alors que pour une object property il peut inférer les relations entre les différents individus et déterminer leur classe.
Nous avons implémenté le multilinguisme de la question 2.2.2-2 en ajoutant un label *temperature* à l'individu *température* pour lequel nous avons défini la langue sur *en* (anglais).

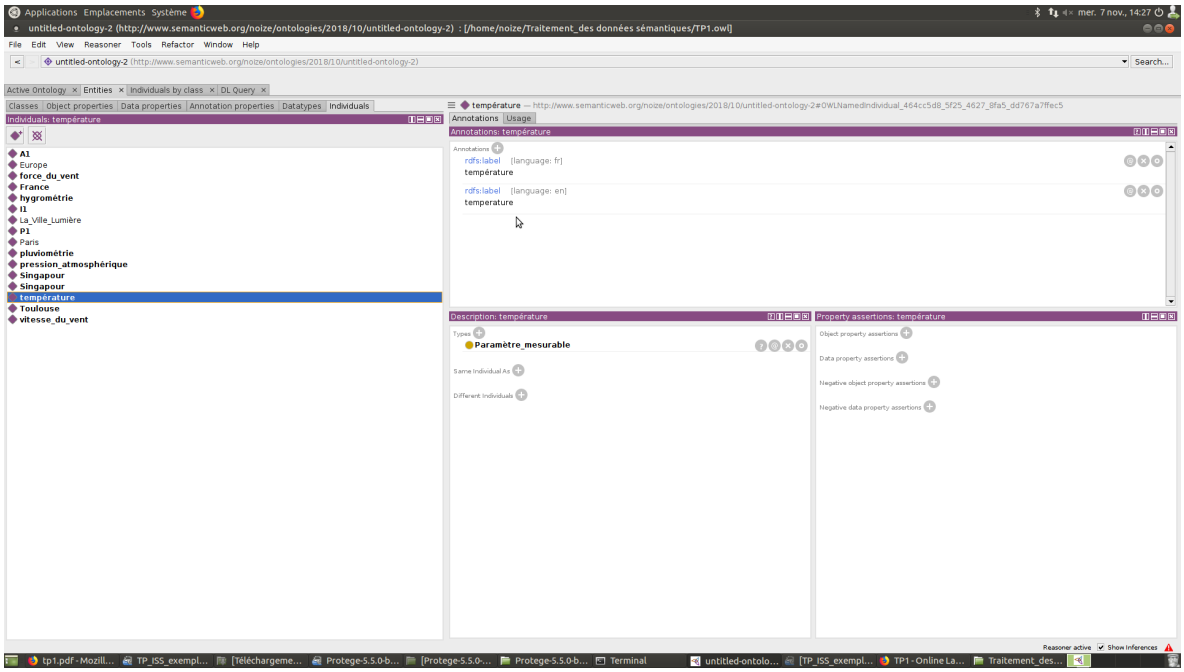


FIGURE 1 – Individu température dans la table des individus

2.2 L'ontologie lourde

2.2.1 Conception

Les axiomes logiques pouvant être représentés en owl incluent des disjonctions comme pour la question 3.1, qui assure que tout individu étant une ville ne peut pas être un pays, et inversement.

Dans la question 3.2 et 3.9, on modélise des classes définies, qui ont pour particularité d'être des restrictions de classes déjà existantes.
Par exemple, la classe *Phénomène court* est une restriction de la classe *Phénomène*, et s'implémente comme suit :

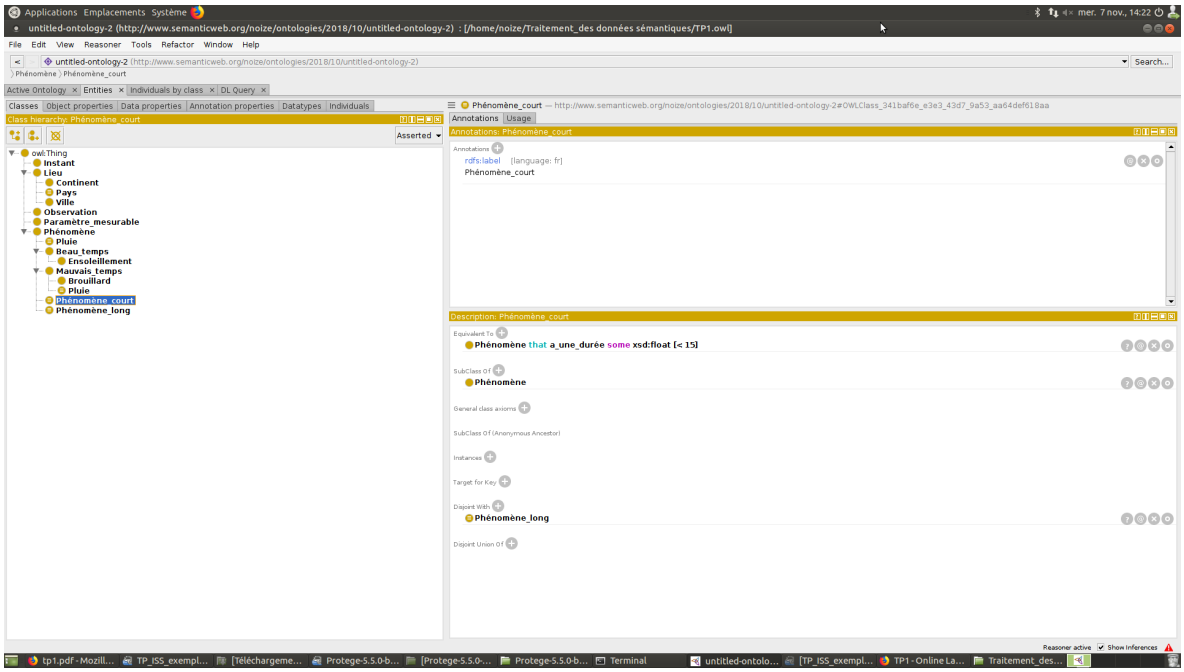


FIGURE 2 – Classe *Phénomène court*

Pour la question 3.8, on utilise la notion de sous-propriété : si P1 est sous-propriété de P2, alors tout individu ayant la propriété P1 a également la propriété P2. En effet, si un pays a pour capitale une ville alors cette dernière est forcément inclue dans le pays. Ainsi, *a pour capitale* est une sous propriété de *inclut*, nous l'avons implémenté comme suit :

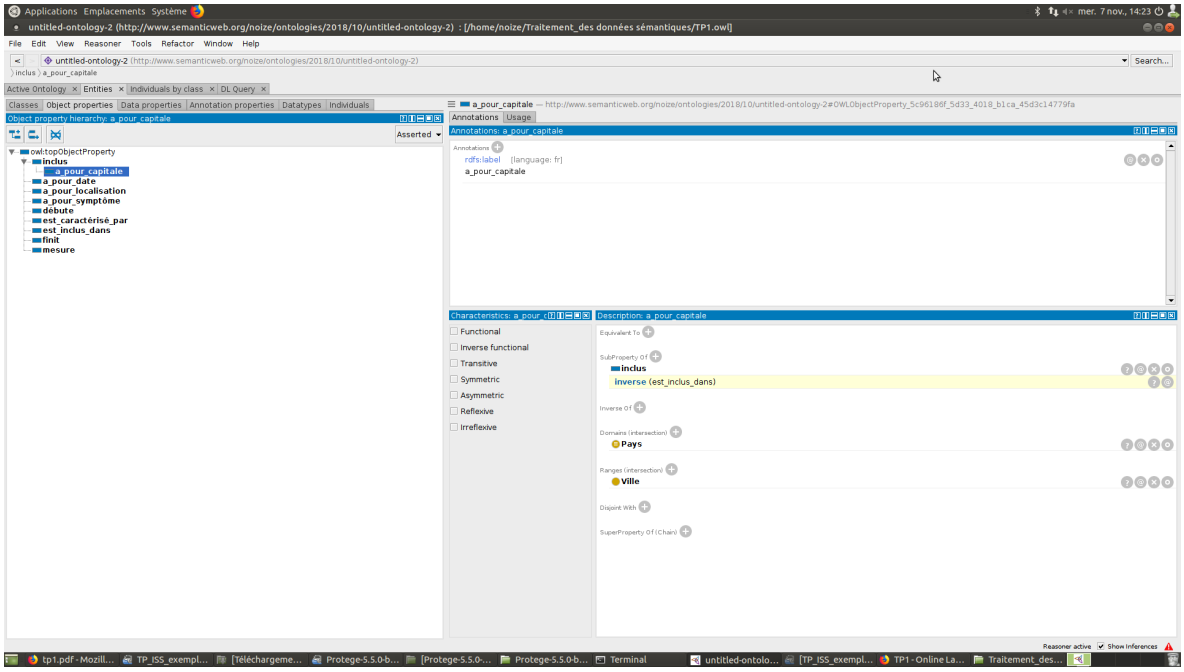


FIGURE 3 – *a pour capitale* sous propriété de *inclut*

2.2.2 Peuplement

Quand on définit à la fois *Paris* et *La ville lumière* comme capitales de la France, le raisonneur déduit de la propriété d'unicité de la capitale pour chaque pays que Paris est La ville lumière.

Le raisonneur déduit de A1 que c'est un phénomène car il sait que A1 a pour symptome P1, et que la relation d'Object Property *a pour symptome* est uniquement applicable à un phénomène pour une observation donnée.

3 Etat de l'ontologie

Dans cette section, nous allons vous présenter l'état dans lequel se trouve notre ontologie à la fin du TP n°1 à travers un ensemble de captures d'écran prises dans Protégé.

3.1 Classes

Voici les **Classes** de notre ontologie :

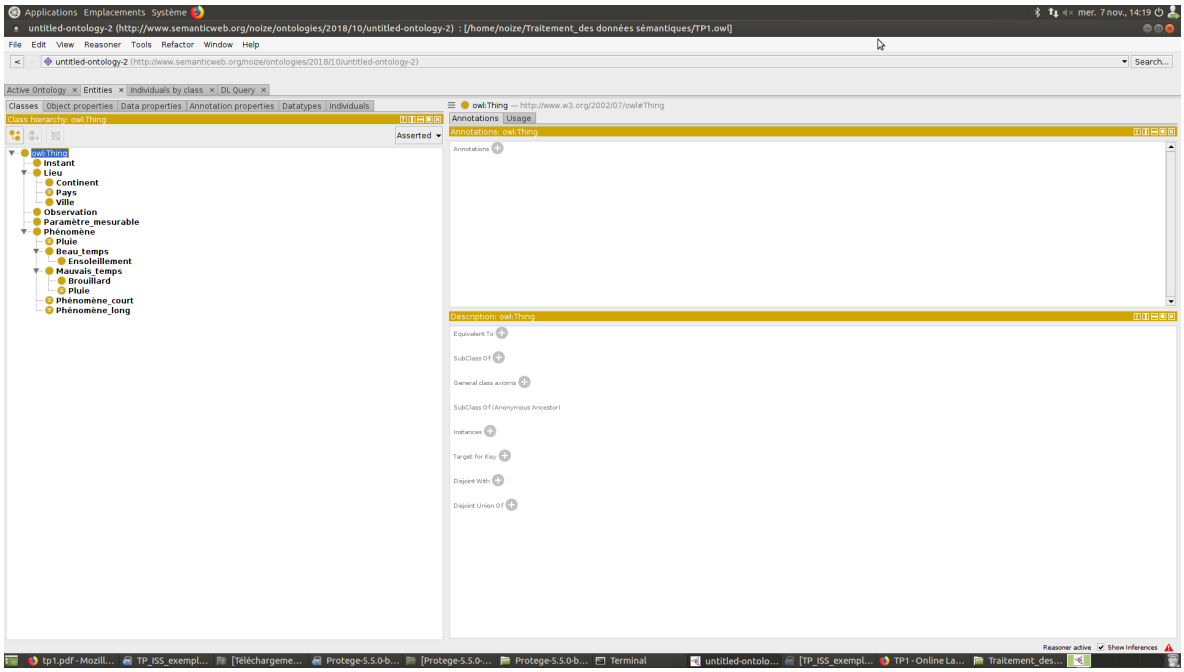


FIGURE 4 – Classes de notre ontologie

3.2 Object properties

Voici les **Object properties** de notre ontologie :

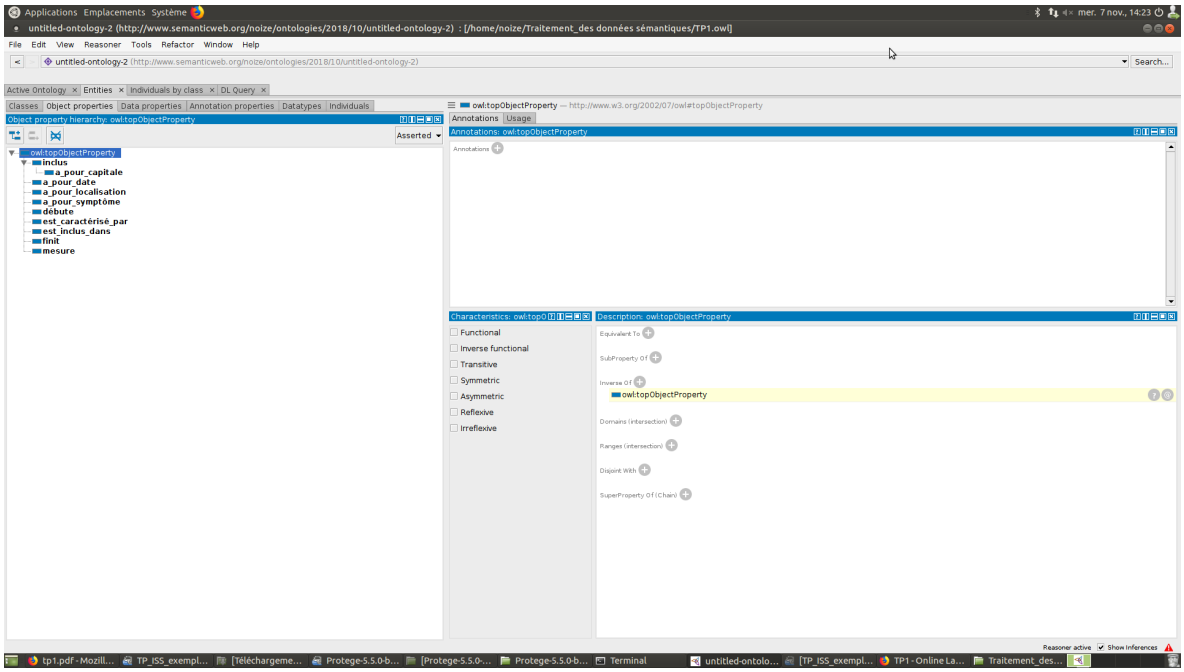


FIGURE 5 – Object properties de notre ontologie

3.3 Data properties

Voici les **Data properties** de notre ontologie :

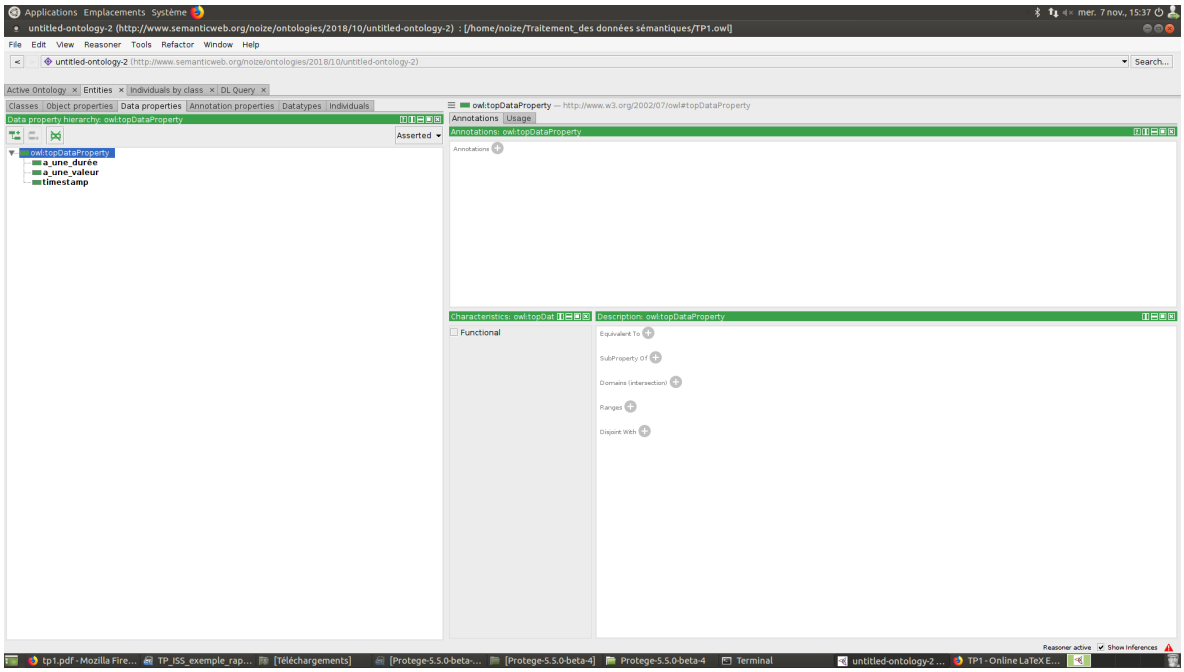


FIGURE 6 – Data properties de notre ontologie

3.4 Individuals

Voici les **Individus** de notre ontologie :

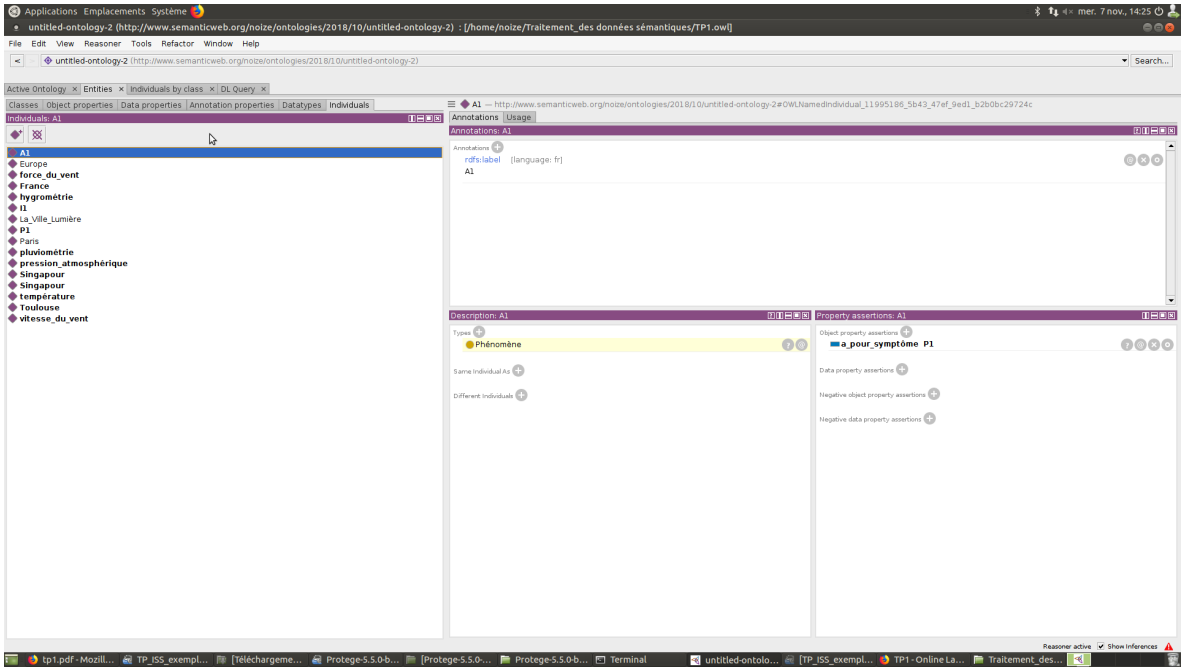


FIGURE 7 – Individuals de notre ontologie

4 Exploitation de l'ontologie

Le but de cette partie est de réutiliser l'ontologie précédemment créée afin d'annoter un open data set issue de la ville d'Aarhus au Danemark. L'idée est d'en suite se servir de la base de connaissances ainsi créée et de déterminer quels capteurs sont défectueux.

Par rapport aux données brutes issues du dataset, l'ontologie joue le rôle de terminologie. En effet, on retrouve au sein de l'ontologie la définition de tous les concepts et de leurs relations.

Comme pour la manipulation via Protégé, l'API fait une distinction entre le label et l'URI : le label est un juste un nom arbitraire donné par l'utilisateur et ne garantit à aucun moment l'unicité de l'identification. Pour cela, on utilise l'URI qui est l'unique identifiant de tout élément de l'ontologie.

L'API programmée fonctionne sous le modèle Model-View-Controler (MVC). Son principe est le suivant :

- Le modèle, la vue et le contrôleur sont trois services distincts
- Le modèle contient la majorité des algorithmes, compilant les données au plus bas niveau du modèle
- La vue s'occupe de l'affichage des données compilées, et offre à l'utilisateur la possibilité d'interagir avec le programme

— Le contrôleur fait l'interface entre la vue et le modèle, et permet une interaction entre les deux services

Ci-dessous une illustration du modèle MVC avec les interactions entre les différents services :

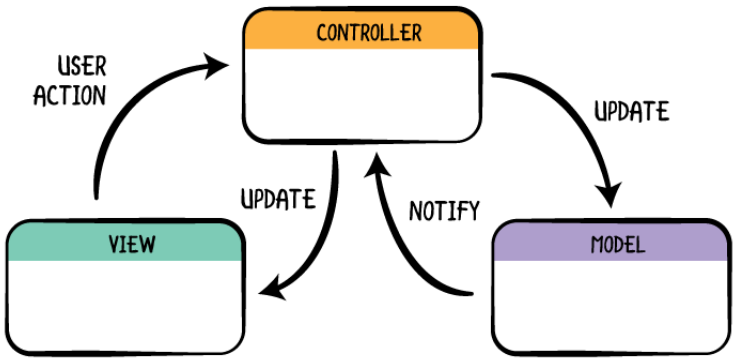


FIGURE 8 – Le modèle MVC

4.1 Modèle

4.1.1 *createPlace(String name)*

Cette fonction, qui prend en paramètre un String, crée une instance de la classe Lieu ayant pour label le nom entré en paramètre, et renvoie l'URI de l'instance de classe créée :

```
1 @Override
2 public String createPlace(String name) {
3     return model.createInstance(name,model.getEntityURI("Lieu").get(0));
4 }
```

4.1.2 *createInstant(TimestampEntity instant)*

La fonction *createInstant* crée une instance de la classe **Instant**, et lui associe une **data property** *a pour timestamp* ayant pour valeur le timestamp passé en paramètre. L'URI de l'instance de classe ainsi créée est alors retourné. Cette association est faite à l'unique condition qu'aucune autre instance n'existe déjà avec le même timestamp :

```
1 @Override
2 public String createInstant(TimestampEntity instant) {
3     for(int i=0;i<model.listLabels(model.getEntityURI("Instant").get(0)).size();i
          ++) {
4         if(model.listLabels(model.getEntityURI("Instant").get(0)).get(i)==instant.
              timestamp) {
5             return null;
6         }
7     }
8
```

```
9     String URI = model.createInstance(instant.timestamp, model.getEntityURI("
          Instant").get(0));
10    model.addDataPropertyToIndividual(URI, model.getEntityURI("a pour timestamp").
          get(0), instant.timestamp);
11    return URI;
12 }
```

4.1.3 *getInstantURI(TimestampEntity instant)*

Cette fonction retourne l'URI de l'instance de la classe **Instant** ayant pour timestamp le timestamp passé en paramètre :

```
1  @Override
2  public String getInstantURI(TimestampEntity instant) {
3      String URI = null;
4      String Instant_URI = model.getEntityURI("Instant").get(0);
5
6      for(int i=0;i<model.getInstancesURI(Instant_URI).size();i++) {
7          if(model.hasDataPropertyValue(model.getInstancesURI(model.getEntityURI("
               Instant").get(0)).get(i), model.getEntityURI("a pour timestamp").get(0),
                instant.timestamp)) {
8              URI = model.getInstancesURI(model.getEntityURI("Instant").get(0)).get(i)
                   ;
9          }
10     }
11
12     return URI;
13 }
```

4.1.4 *getInstantTimestamp(String instantURI)*

La fonction *getInstantTimestamp* permet de récupérer la valeur du timestamp associé à l'Instant dont l'URI est passé en paramètre de la fonction. Si cet URI ne correspond à aucun Instant, alors la fonction ne retourne rien :

```
1  @Override
2  public String getInstantTimestamp(String instantURI)
3  {
4      String timestamp = null;
5
6      for(int n=0;n<model.listProperties(instantURI).size();n++) {
7          if(model.listProperties(instantURI).get(n).get(0).equals(model.getEntityURI
               ("a pour timestamp").get(0))) {
8              timestamp = model.listProperties(instantURI).get(n).get(1);
9          }
```

```
    }

    return timestamp;
}
```

4.1.5 *createObs(String value, String paramURI, String instantURI)*

createObs crée une instance de la classe **Observation** avec les paramètres suivants :

- la valeur passée en paramètre est la mesure effectuée
- l'URI *paramURI* correspond à l'URI du capteur utilisé pour la mesure
- L'URI *instantURI* correspond à l'URI de l'Instant auquel à été prise la mesure

et retourne l'URI associé à l'observation créée :

```
@Override
    public String createObs(String value, String paramURI, String instantURI) {
        String obsURI = null;

        obsURI = model.createInstance(value, model.getEntityURI("Observation").get
            (0));
        model.addObjectPropertyToIndividual(obsURI,model.getEntityURI("a pour date"
            ).get(0), instantURI);
        model.addObservationToSensor(obsURI, model.whichSensorDidIt(
            getInstantTimestamp(instantURI), paramURI));
        model.addObjectPropertyToIndividual(obsURI,model.getEntityURI("mesure").get
            (0),paramURI);
        model.addDataPropertyToIndividual(obsURI,model.getEntityURI("a pour valeur"
            ).get(0),value);

        return obsURI;
}
```

4.2 Contrôleur

4.2.1 *instantiateObservations(List<ObservationEntity> obsList, String paramURI)*

La fonction *instantiateObservation* analyse la liste d'observations extraite du dataset et les instancie dans notre base de connaissance :

```
@Override
public void instantiateObservations(List<ObservationEntity> obsList, String
    paramURI) {
    Iterator<ObservationEntity> obs = obsList.iterator();
    while(obs.hasNext()) {
        ObservationEntity current = obs.next();
        String instantURI = customModel.createInstant(current.getTimestamp());
```

```
7            customModel.createObs(current.getValue().toString(), paramURI, instantURI);
8        }
9    }
```

4.3 Etat de la base de connaissances

Voici quelques captures d'écran montrant l'état de notre base de connaissances après export par l'API et import dans *Protégé*.

4.4 Classes

Voici les **Classes** de notre base de connaissances :

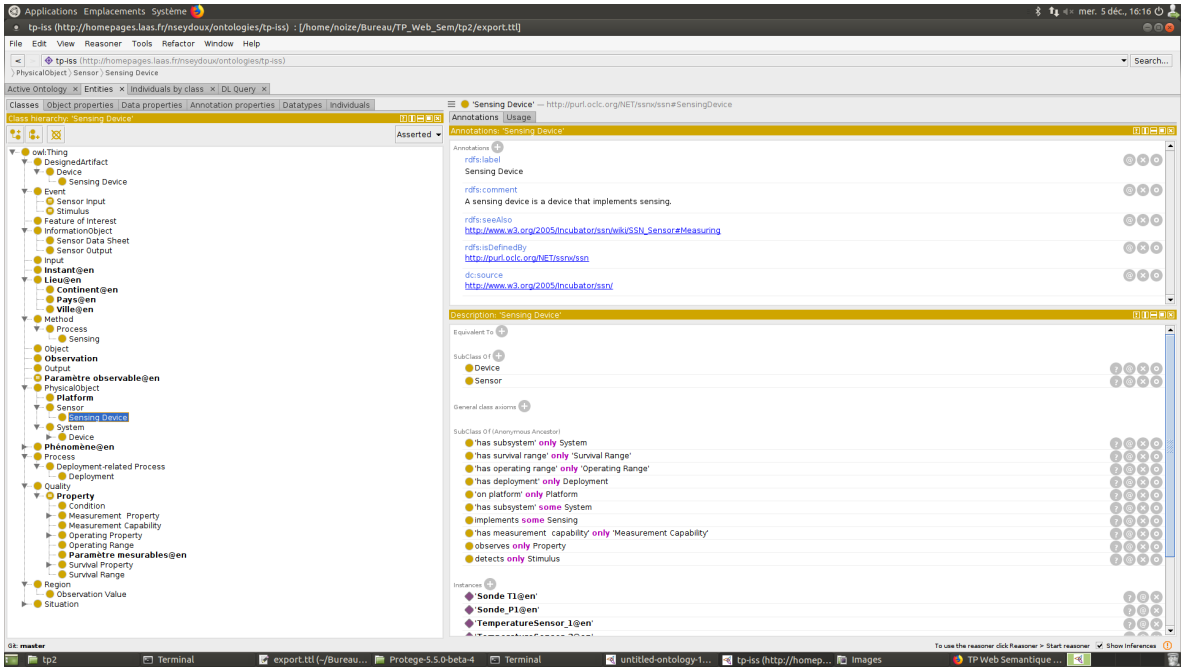


FIGURE 9 – Classes de notre base de connaissances

4.5 Object properties

Voici les **Object properties** de notre base de connaissances :

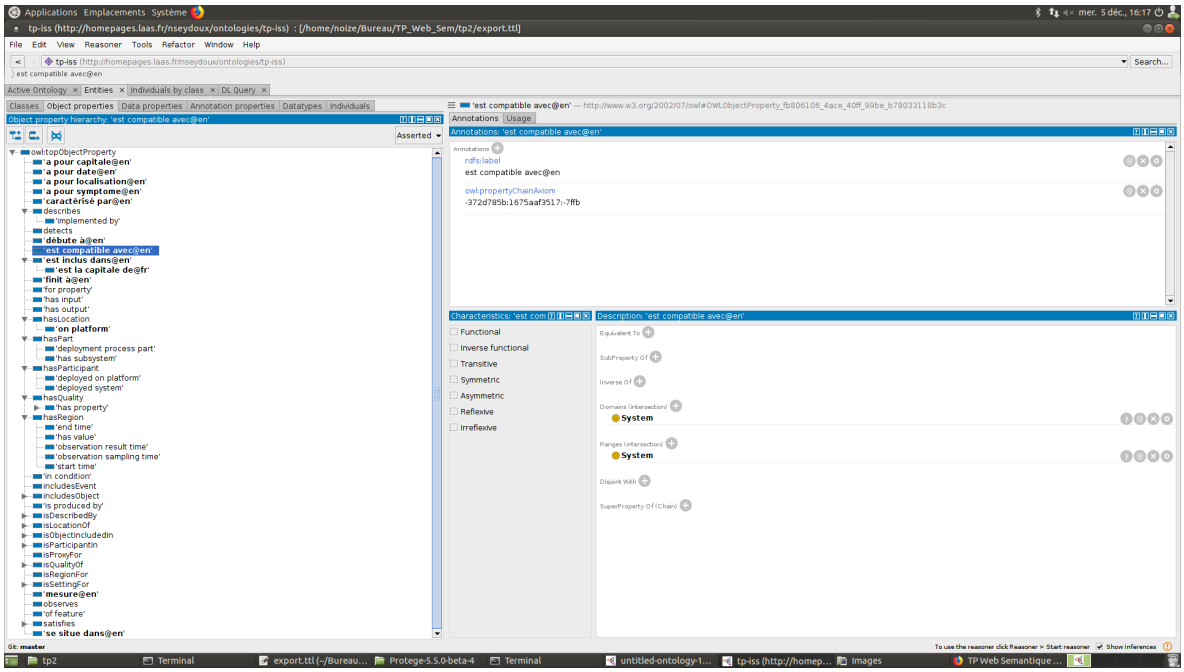


FIGURE 10 – Object properties de notre base de connaissances

4.6 Data properties

Voici les **Data properties** de notre base de connaissances :

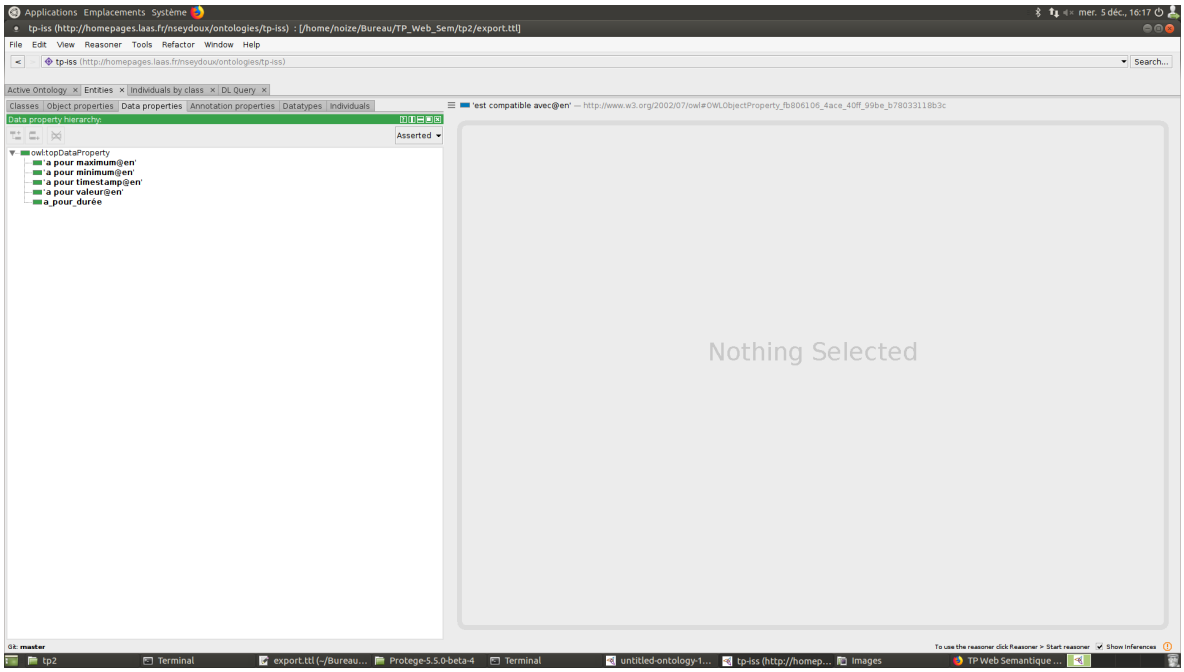


FIGURE 11 – Data properties de notre base de connaissances

4.7 Individuals

Voici les **Individus** de notre base de connaissances :

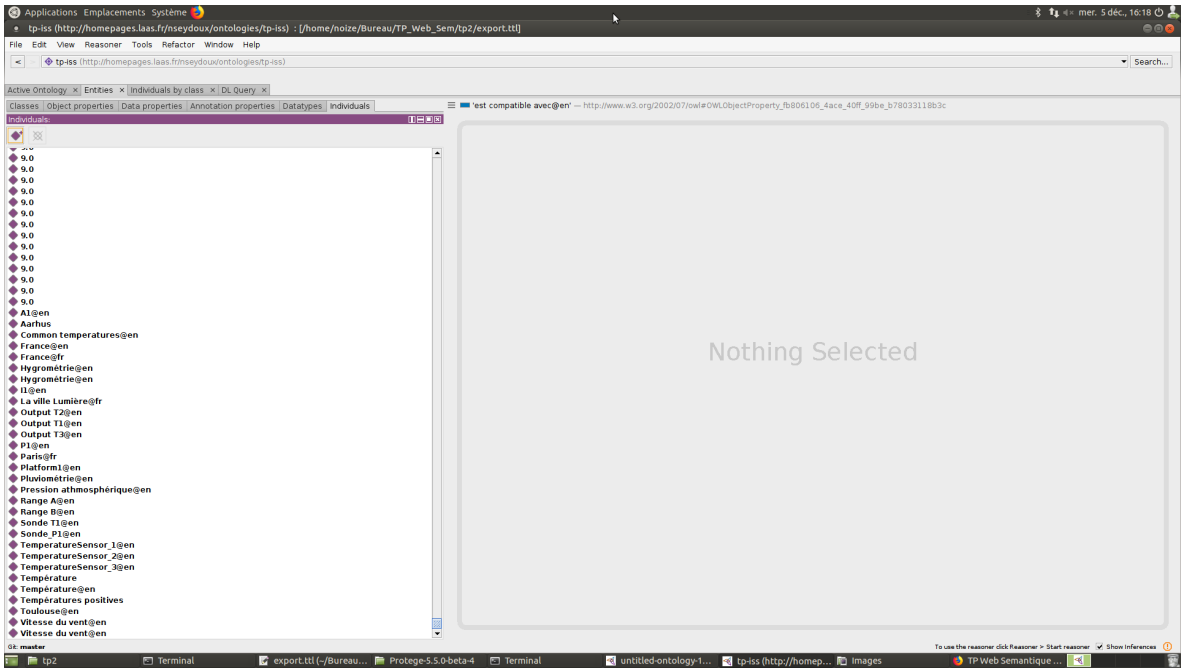


FIGURE 12 – Individuals de notre base de connaissances

5 Conclusion

Au cours de ces TPs, nous avons pu implémenter une base de connaissances.
En effet, nous avons préalablement défini une ontologie portant sur la météorologie que nous avons ensuite enrichie grâce à un open data set issu de la ville d'Aarhus au Danemark.
La définition de l'ontologie sous *Protégé* ainsi que la programmation de l'API ne nous ont pas posé de problème majeur. Cependant, nous n'avons pas réussi une fois la base de connaissances générée à déterminer les capteurs non fonctionnels, pour cause d'erreurs non résolues lors de l'exécution du raisonneur.