Création d'une ontologie en OWL avec Protégé 4/5

Jean Charlet, Audrey Baneyx, Xavier Aimé, Jacques Hilbey

▶ **To cite this version:**

Jean Charlet, Audrey Baneyx, Xavier Aimé, Jacques Hilbey. Création d'une ontologie en OWL avec Protégé 4/5. Master. France. 2023, pp.131. hal-04049141

HAL Id: hal-04049141
https://hal.science/hal-04049141

Submitted on 28 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Création d'une ontologie en OWL avec Protégé 4/5

Jean Charlet[1,2]
Audrey Baneyx[3], Xavier Aimé[2] , Jacques Hilbey[2]
[1]AP-HP
[2]Inserm U1142 – LIMICS
[3]Sciences Po

Révision : 28/03/2023

Jean.Charlet@upmc.fr

Réalisé d'après le document :

« A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools », disponible à l'adresse suivante : http://owl.cs.manchester.ac.uk/publications/talks-and-tutorials/protg-owl-tutorial/

Objectifs

A la fin de ce TP, vous serez en mesure de :

- vous servir d'un **éditeur** d'ontologies
- créer une ontologie de domaine
  - créer des **concepts**
  - créer des **relations**
  - créer des **restrictions**
- utiliser un **raisonneur**

Plan

- Quelques connaissances sur OWL
- Création d'un canevas d'ontologie
- La hiérarchie des classes et sous-classes
  - La disjonction
- Les relations
  - Les différentes caractéristiques
  - Le domaine et la portée
- Les restrictions
  - Les restrictions existentielles
  - Les restrictions universelles
- Les classifieurs

Les 3 versions d'OWL 1.0

- OWL-Lite (SHIF)
  - Simple hiérarchie de concepts et peu de contraintes
  - Organisation hiérarchique de thésauri
- **OWL-DL** (SHOIN)
  - Fondé sur les logiques de description
  - Implémente un fragment de la logique du premier ordre
  - Vérifie la hiérarchie des concepts et détecte les inconsistances
- OWL-Full
  - Langage le plus expressif
  - A utiliser dans des situations où l'expressivité est plus importante que la complétude et la décidabilité → pas de raisonnement automatique

Les 3 *profils* d'OWL 2.0 DL (1/2)

3 profils restreignant l'expressivité de OWL-DL et fondés sur des critères de décidabilité.

Fondé sur la LD SROIQ, and is geared towards enabling ontologies with a high degree of expressivity in the language:

- OWL2-EL – fondé sur EL++, which is geared towards scalable reasoning in the TBox (i.e., polymonial-time reasoning for most inference tasks such as classification)
- OWL2-QL – fondé sur DL-Lite, which is geared towards scalable query answering in the ABox (when dealing with lots of instance data and a relatively simple TBox)
- OWL2-RL – fondé sur Description Logic Programs (DLP), which has an expressivity that subsets that of OWL2 DL (the fragment that can be handled using a description logic)

Nota: OWL1 Lite et OWL1 DL n'ont plus de raison d'exister en tant que tels puisque remplacés par les profils. OWL / full existe toujours

Les 3 *profils* d'OWL 2.0 DL (2/2)

https://www.w3.org/TR/2009/REC-owl2-profiles-20091027/#OWL_2_EL
https://www.w3.org/TR/2009/REC-owl2-primer-20091027/#OWL_2_EL

https://www.w3.org/TR/2009/REC-owl2-profiles-20091027/#OWL_2_QL
https://www.w3.org/TR/2009/REC-owl2-primer-20091027/#OWL_2_QL

https://www.w3.org/TR/2009/REC-owl2-profiles-20091027/#OWL_2_RL
https://www.w3.org/TR/2009/REC-owl2-primer-20091027/#OWL_2_RL

Quels sont les composants d'OWL? (1/2)

- *Les **instances** de classes (individuals)*
  - Objets du domaine que nous souhaitons modéliser
    - Ex : Pierre, Thierry, Marie, Italie, USA…
- *Les **propriétés** ou relations (properties)*
  - Relations binaires entre des instances de classes
    - Ex : [Marie] (hasChild)[Pierre]
    - Caractéristiques :
      - Inverses : hasOwner/isOwnedBy
      - Fonctionnelle : une seule valeur
      - Transitive
      - Symétrique
    - « *ObjectProperties* » dans Protégé
  - Relations binaires entre des instances et des objets XML
    - Ex : [Marie] (hasAge)[integer:25]

Quels sont les composants d'OWL? (2/2)

- Les concepts ou classes
  - Ce sont des ensembles d'instances
    - Ex : le concept Pays est constitué de la France, l'Italie, les USA…
  - Les classes sont construites à partir de descriptions qui contraignent les conditions d'appartenance d'une instance à une classe ou d'une sous-classe à une classe
    -> Sujet du tutoriel

Avant de commencer...

- ***Protégé 5*** : éditeur d'ontologies gratuit et "open source" qui permet de structurer une base de connaissances

  http://protege.stanford.edu/

- Téléchargement du logiciel

  "Download now"

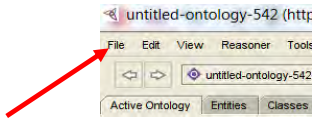
- Propose automatiquement la version pour l'ordinateur connecté

  "Download for …"

- Installation

Création d'une nouvelle ontologie avec Protégé 4 (1/3)

1. Démarrer Protégé 4
2. Dans le menu « File », choisissez « New… »



Création d'une nouvelle ontologie avec Protégé 4 (2/3)

- Dans le champ « *Ontology IRI* », changez l'adresse IRI par *ce que vous voulez* (ou ne changez rien)

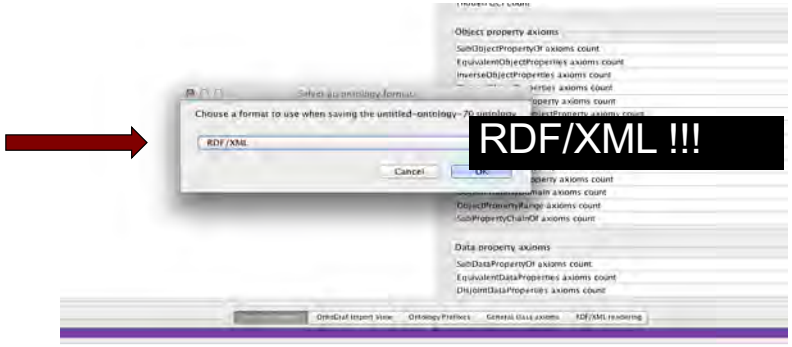
Création d'une nouvelle ontologie avec Protégé 4 (3/3)

- **Pensez à sauver votre ontologie**

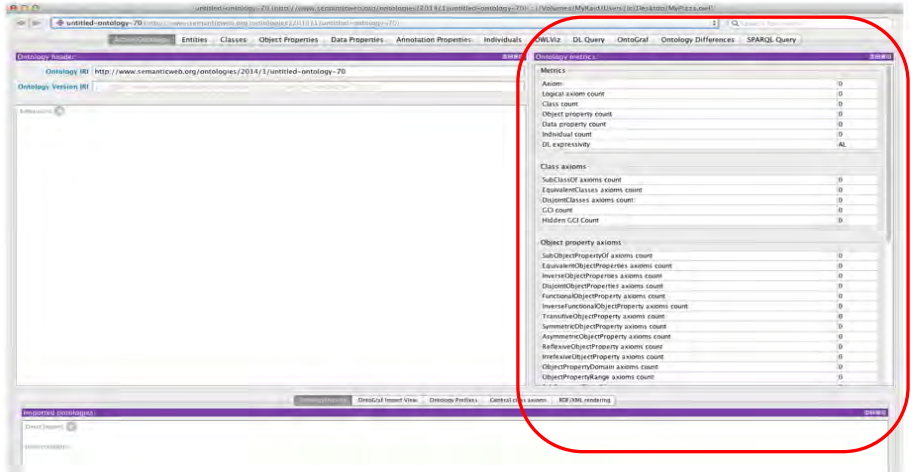
Dans le menu « File » :

- Choisissez « Save as… »
- Nommer votre fichier « MyPizza.owl »

Format de sauvegarde de l'ontologie



Mesures de l'ontologie

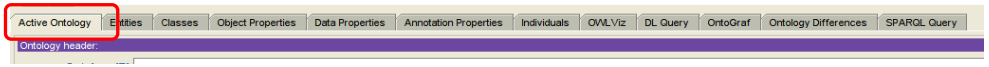


Création d'une ontologie avec Protégé 4

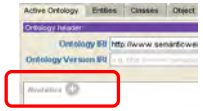
- Métadonnées de l'ontologie

Ajouter un commentaire à l'ontologie

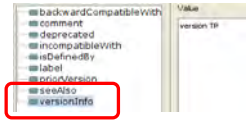
1. Assurez-vous que l' onglet « *Active Ontology* » est sélectionné



2. Dans la partie dédiée à « *Ontology Header* », cliquez sur le signe **+** à droite de « *Annotations* ». Une fenêtre éditable apparaît.

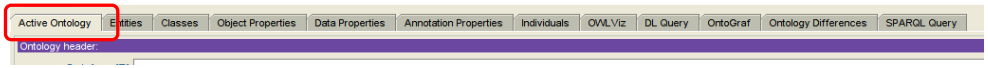


3. Entrez le commentaire « *Version du TP du xx/xx/xxx* » dans le champ « *versionInfo* ». Cliquez sur OK.

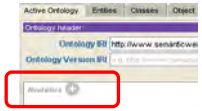


Créer un type de commentaire à l'ontologie

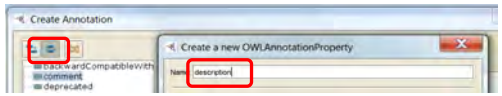
1. Assurez-vous que l' onglet « *Active Ontology* » est sélectionné



2. Dans la partie dédiée à « *Ontology Header* », cliquez sur le signe **+** à droite de « *Annotations* ». Une fenêtre éditable apparaît.



3. Sélectionnez l'icône du milieu pour ajouter une annotation. Appelez-là « *Description* ». Cliquez sur OK. Remplir ensuite votre description.



Créer un type de commentaire à l'ontologie

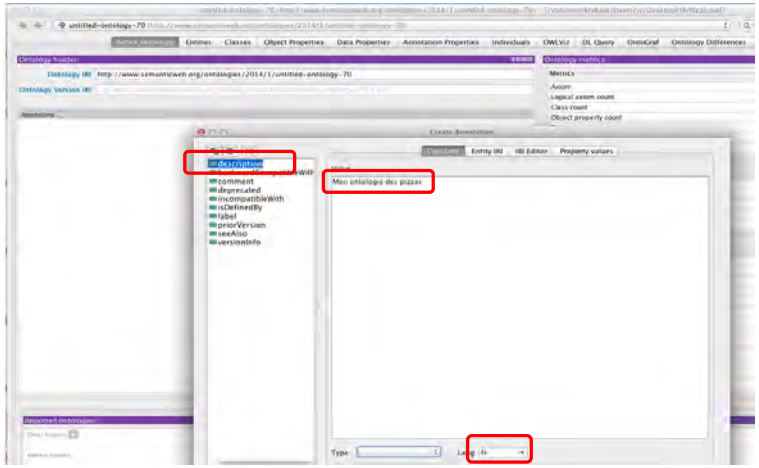


Fig 1: The Ontology Annotations View

Utiliser les métadonnées du DC (choix)

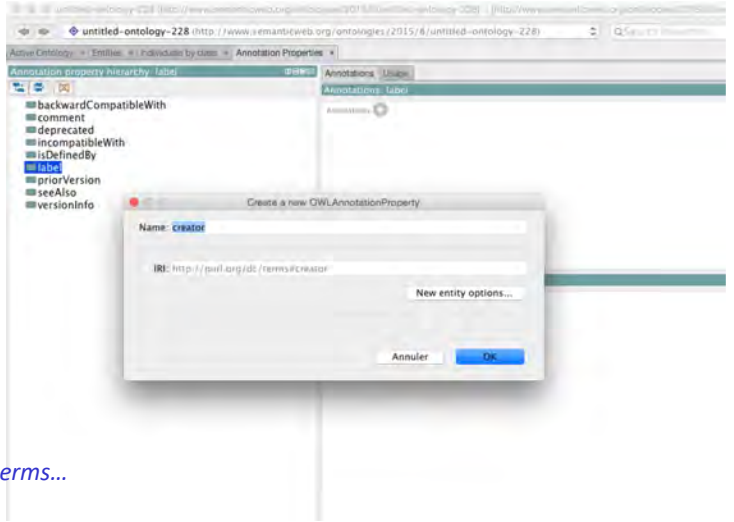
http://dublincore.org/documents/dcmi-terms/

55 éléments et contraintes de co-domaine vs 15 éléments originels sans contrainte

Properties in the */terms/* namespace	abstract , accessRights , accrualMethod , accrualPeriodicity , accrualPolicy , alternative , audience , available , bibliographicCitation , conformsTo , contributor , coverage , created , creator , date , dateAccepted , dateCopyrighted , dateSubmitted , description , educationLevel , extent , format , hasFormat , hasPart , hasVersion , identifier , instructionalMethod , isFormatOf , isPartOf , isReferencedBy , isReplacedBy , isRequiredBy , issued , isVersionOf , language , license , mediator , medium , modified , provenance , publisher , references , relation , replaces , requires , rights , rightsHolder , source , spatial , subject , tableOfContents , temporal , title , type , valid
Properties in the */elements/1.1/* namespace	contributor , coverage , creator , date , description , format , identifier , language , publisher , relation , rights , source , subject , title , type

http://purl.org/dc/terms/contributor *(dcterms:)* vs
http://purl.org/dc/elements/1.1/contributor *(dc:)*

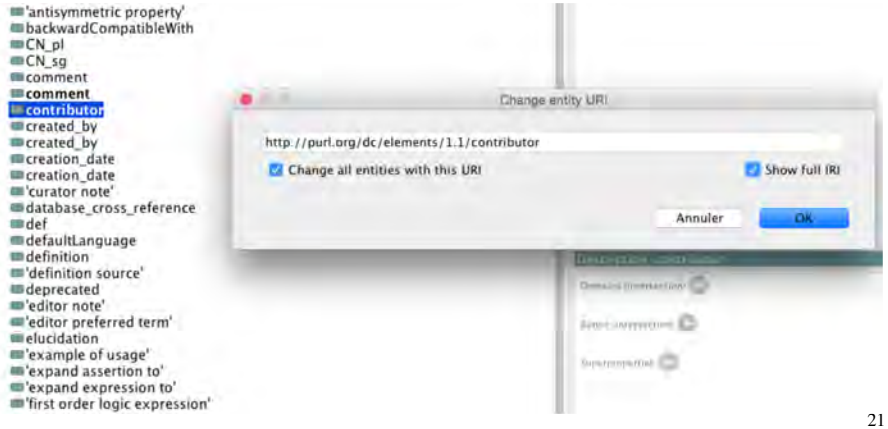
Utiliser les métadonnées du DC (new)



http:///purl.org/dc/terms...

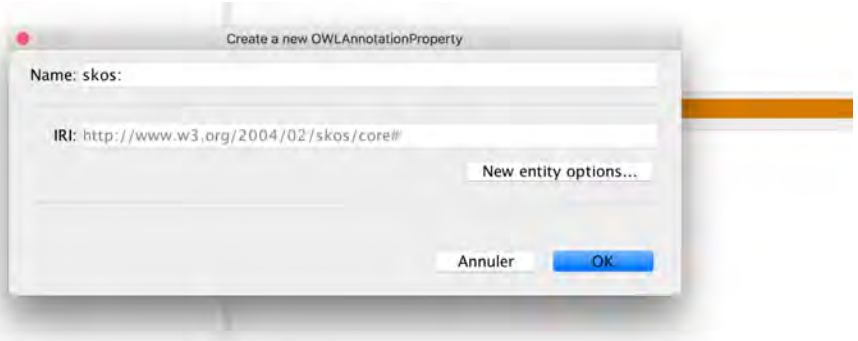
Utiliser les métadonnées du DC (old)

http://purl.org/dc/elements/1.1/...



Profiter de la reconnaissance des NS

- Protégé reconnaît les espaces de nom du fichier mais d'autres standards aussi



Création d'une ontologie avec Protégé 4

- Création de classes

L'onglet « *Classes* »

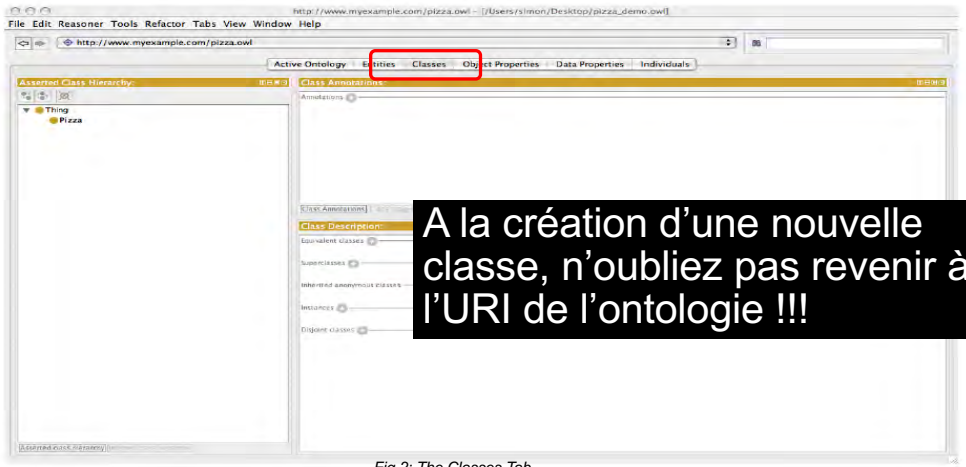
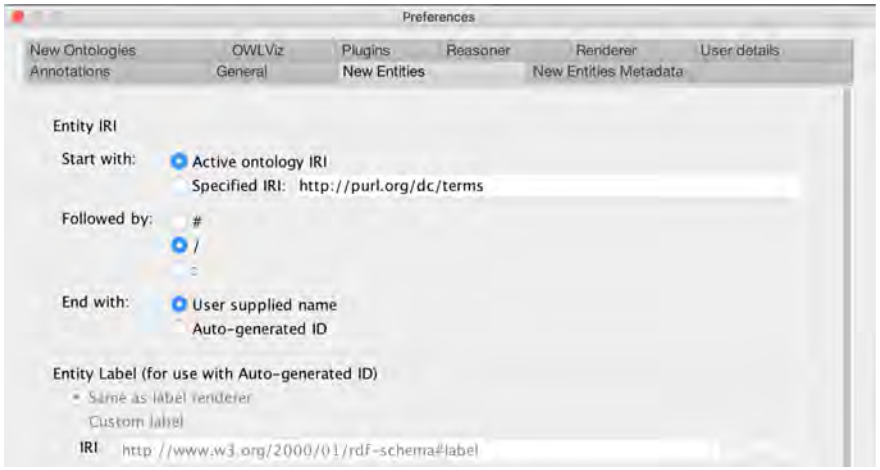


Fig 2: The Classes Tab

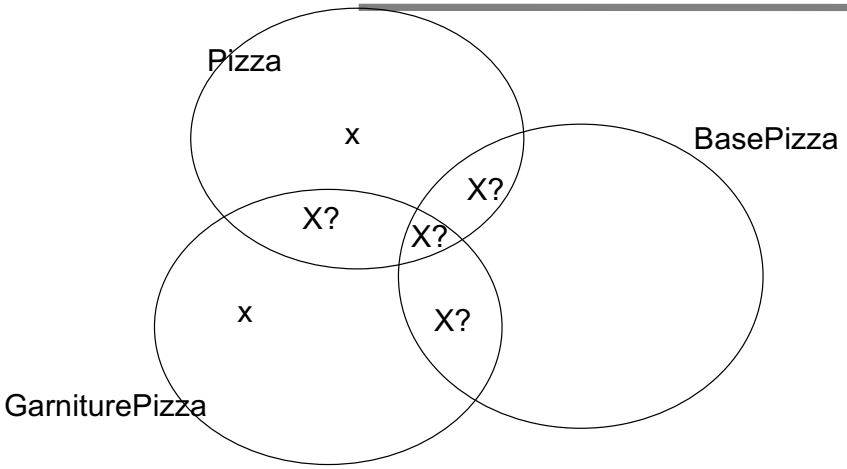
Revenir à la bonne URI (1/3)

- Par Preferences… /New Entity



Revenir à la bonne URI (2/3)

- En entrant le *namespace* directement dans le champ de création du concept



Icônes pour créer une hiérarchie de classes

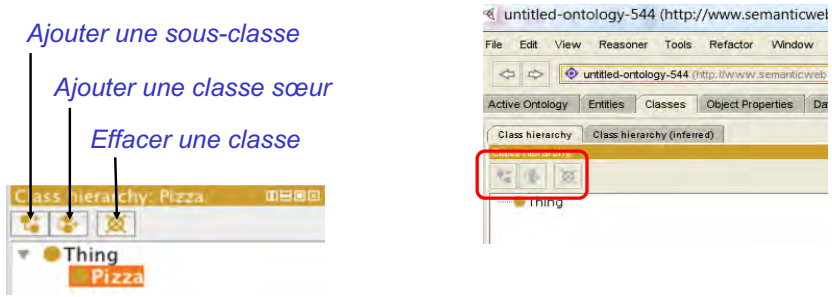
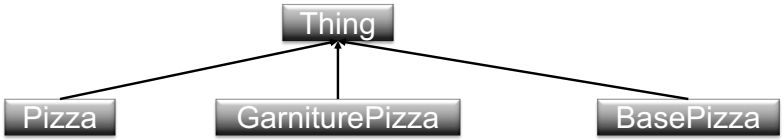


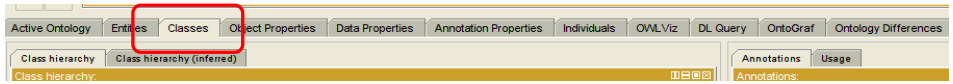
Fig 3 – Icônes des classes

Création de classes

- Objectif = créer la hiérarchie de concepts suivante :



1. Assurez-vous que l' onglet « *Classes* » est sélectionné



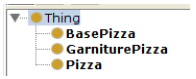
2. Sélectionnez le concept « *Thing* »

Création de classes

3. Cliquez sur le bouton d'ajout d'une classe. Une nouvelle classe apparaît comme sous-classe de celle que vous avez sélectionnée. Une boîte de dialogue apparaît et vous permet de saisir le nom de votre nouvelle classe. Saisir « *Pizza* »

4. Répétez l'opération pour les classes *GarniturePizza* et *BasePizza*.

Assurez-vous que ces classes apparaissent sous la classe « *Thing* » dans la hiérarchie.



Création de classes

→ *Vocabulaire* : Une hiérarchie de classes peut également être appelée une taxinomie.

→ Décidez de travailler :

. en *anglais (plus facile de retrouver des compléments sur le TP)*

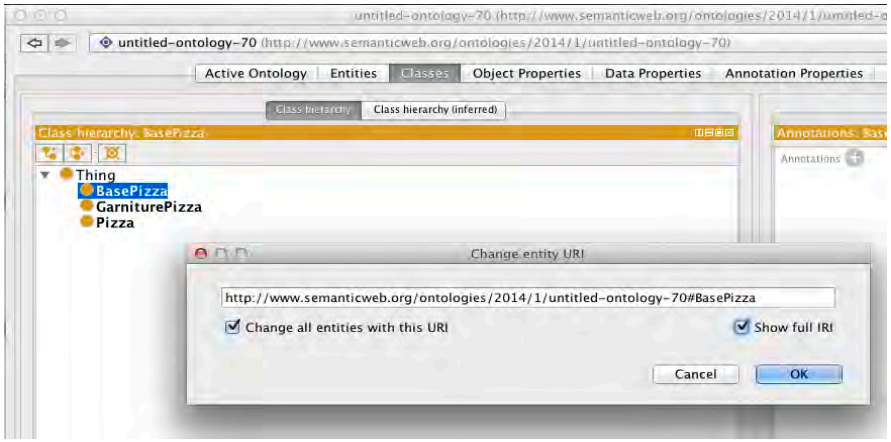
. en *français (plus logique)*

→ *Idée* : Après avoir créé « *Pizza* », plutôt que de sélectionner « *Thing* » et d'utiliser le bouton de création d'une sous-classe, vous pouvez utiliser le **bouton de création d'une classe sœur** pour créer « *GarniturePizza* » et « *BasePizza* ». Il faut dans ce cas sélectionner la classe « *Pizza* ».

- **Pensez à sauver votre ontologie**

Création de classes

- Sur les URI, les fragURI et la commande Refactor (ou Ctrl-U)



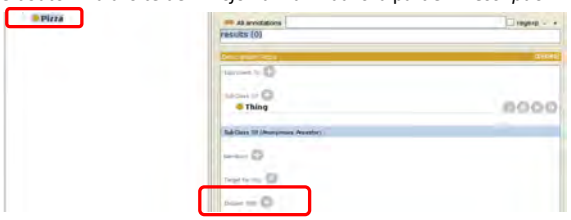
Création d'une ontologie avec Protégé 4

- Classes disjointes

Classes disjointes

- Faire de Pizza, GarniturePizza et BasePizza des classes disjointes les unes des autres

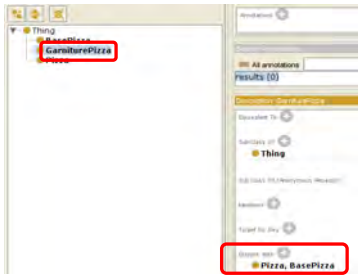
1. Sélectionnez la classe « *Pizza* » dans la hiérarchie.
2. Cliquez sur le bouton + à droite de « *Disjoint with* » dans la partie « *Description* ».



3. Une fenêtre s'ouvre et vous permet de choisir les classes qui doivent être disjointes. Si vous le faites pour une des classes (par exemple « *BasePizza* »), l'action sera automatiquement propagée sur les autres classes sélectionnées.

Classes disjointes

- Résultat :
  - Sélectionnez la classe « *GarniturePizza* ». Notez que cette classe est maintenant disjointe de « *Pizza* » et « *BasePizza* ».



Si on déplace ensuite les classes,
on crée facilement des incohérences

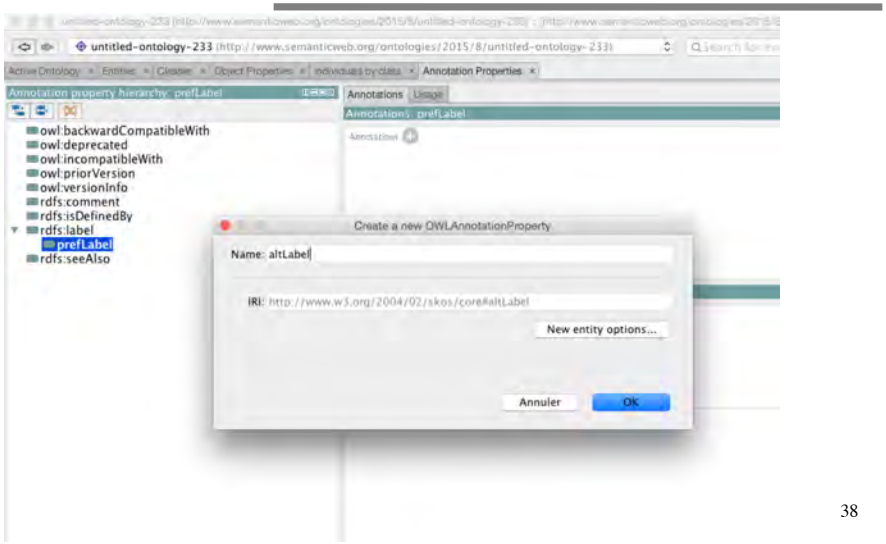
Qu'est ce que cela signifie?

- En OWL, les classes, par définition, se chevauchent. Nous ne pouvons pas supposer qu'un(e) individu/instance n'est pas membre d'une classe particulière simplement parce qu'il(elle) n'a pas été défini(e) comme étant un membre de cette classe.
- Pour séparer un groupe de classes, nous devons les déclarer comme disjointes les unes des autres. Cela assure qu'un individu créé comme membre d'une classe du groupe ne peut pas être membre d'une autre classe de ce même groupe.
- Dans notre exemple précédent, « *Pizza* », « *GarniturePizza* » et « *BasePizza* » sont disjointes entre elles. Cela signifie qu'aucun individu ne peut être à la fois une Pizza et une BasePizza.

Annotations multilingues

- Afficher l'ontologie (render) par ses identifiants
- Créer des label français et anglais pour chacun des 3 concepts
- Afficher l'ontologie en français
- Afficher l'ontologie en anglais

Création des skos:prefLabel et skos:altLabel

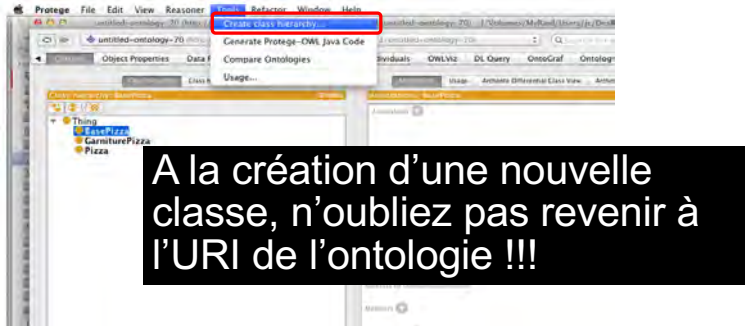


Création d'une ontologie avec Protégé 4

- Créer des classes avec les outils Protégé

Créer des classes avec les outils Protégé

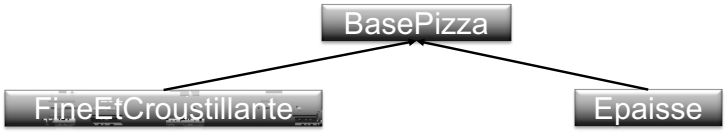
- Les outils OWL mis à disposition dans cette version de Protégé sont des plugins. Ils servent à simplifier des traitements répétitifs et longs.
- Nous allons utiliser l'outil « ***Create Class Hierarchy*** » pour ajouter des sous-classes à la classe « *BasePizza* ».



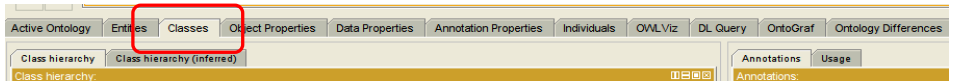
A la création d'une nouvelle classe, n'oubliez pas revenir à l'URI de l'ontologie !!!

Créer des classes avec les outils Protégé

- Objectif = créer les sous-classes de « *BasePizza* »



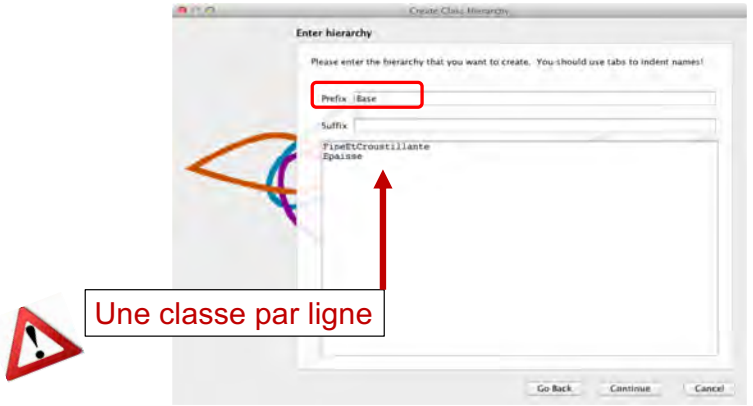
1. Assurez-vous que l' onglet « *Classes* » est sélectionné



2. Menu *Tools* > *Create Class Hierarchy.* Sélectionnez le concept « *BasePizza* »

Créer des classes avec les outils Protégé

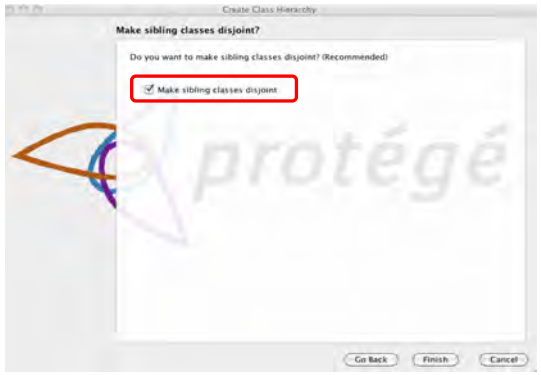
3. Cliquez sur « *Continue* ». Indiquer les sous-classes de « *BasePizza* » que nous souhaitons créer. Tapez « FineEtCroustillante » (pour une pizza à pâte fine) et « Epaisse » (pâte épaisse). **Mettez « *Base* » comme préfixe**.



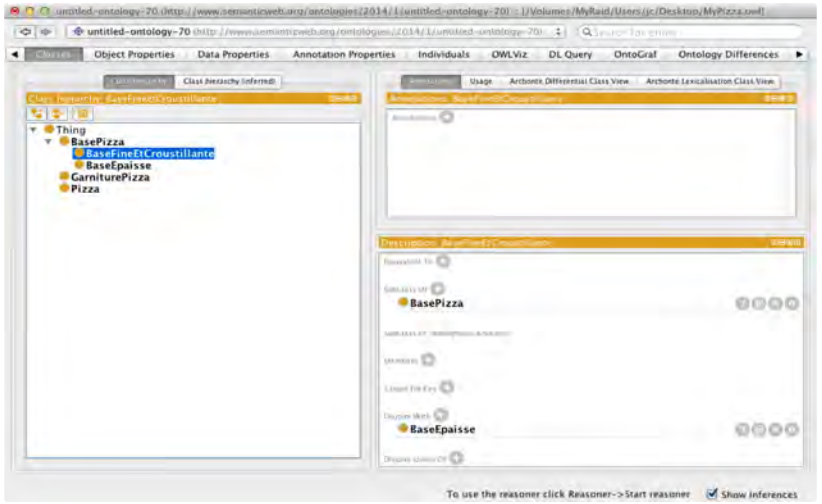
Une classe par ligne

Créer des classes avec les outils Protégé

4. Cliquez sur « *Continue* ». L'outil vérifie que les noms saisis correspondent au style prédéfini (pas d'espace, pas d'accent...). Il vérifie également que ces noms n'existent pas déjà.
5. Cochez le bouton radio « *Make sibbling classes disjoint* » (sous-classes disjointes).

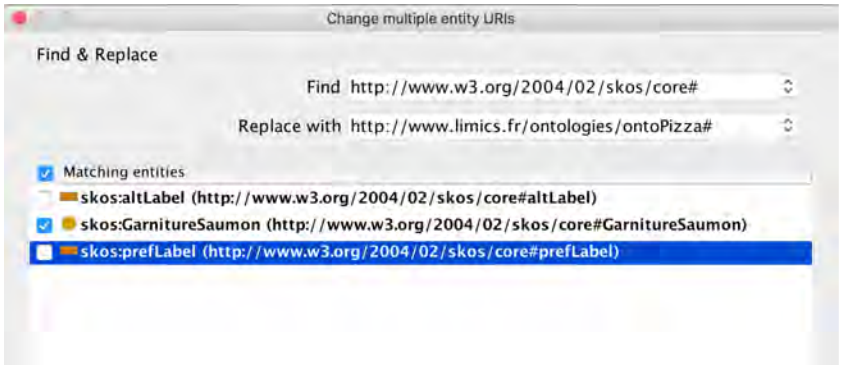


Créer des classes avec les outils Protégé



Revenir à la bonne URI (3/3)

- Si l'erreur est déjà commise, correction d'erreur multiples par Refactor/Rename multiple entities et sélection par le popup menu et sélection spécifique par coches



Création d'une ontologie avec Protégé 4

- Créer des sous-classes de « *GarniturePizza* »

Créer des sous-classes de *GarniturePizza*

- Objectif = créer les sous-classes de « *GarniturePizza* »

- **Fromage** :
  - Mozarella
  - Parmesan

- **Fruits de la mer** :
  - Anchois
  - Crevette
  - Thon
  - Saumon

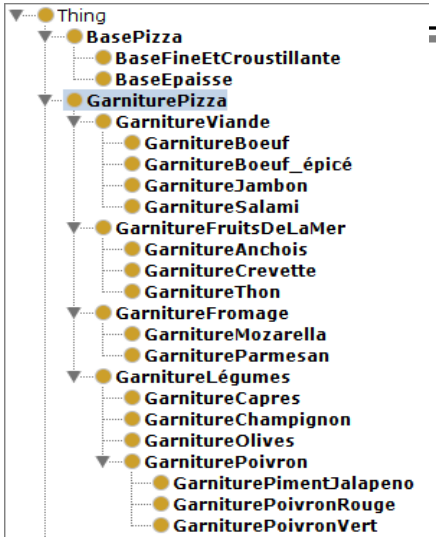
Légumes :
- Tomate
- Câpres
- Champignon
- Olives
- Poivron :
  - Piment Jalapeno
  - Poivron vert
  - Poivron rouge

- **Viande**
  - Salami
  - Bœuf
  - Jambon
  - Pepperoni

Créer des sous-classes de *GarniturePizza*

1. Sélectionnez la classe « *GarniturePizza* » dans la hiérarchie.
2. Lancez l'outil « *Create class hierarchy »*.
3. Assurez-vous que « *GarniturePizza* » est sélectionné.
4. Cliquez sur « *Continue* ».
5. Nous voulons que les noms des nouvelles classes commencent tous par '*Garniture*' → dans la zone de texte intitulée « *Prefix »*, tapez « Garniture ». L'outil l'ajoutera automatiquement à l'ensemble des noms de classes que vous allez créer.
6. L'outil permet de créer une hiérarchie de classes en utilisant les **indentations** (tabulation).

Résultat



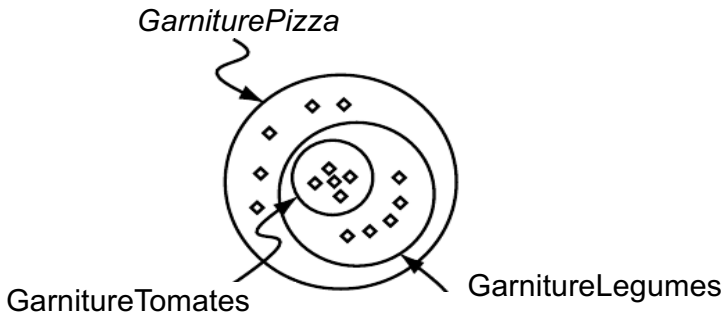
De l'intension à l'extension

- Ce à quoi vous pensez en fabriquant une ontologie est **l'intension** des concepts.
- Ce que vous fabriquez dans Protégé respecte une *logique des prédicats*. Sans le savoir vous traitez vos concepts d'une manière logique donc ensembliste et ils existent par les objets, *les instances* ou *les sous-classes,* qui appartiennent à la classe.
- Vous passez de **l'intension** à **l'extension**.
- Vous devez vous en souvenir pour comprendre certains comportements de l'outil et des *plugins* de raisonnement.

https://fr.wikipedia.org/wiki/Intension_et_extension

L'intension est la définition d'un concept par ses propriétés alors que l'intention correspond à un dessein délibéré, une action de la volonté par laquelle on se fixe un but

La signification des sous-classes



Tous les individus (instances) qui sont membres de la classe « GarnitureTomates » sont également membres de la classe « GarnitureLegumes » et membres de la classe « GarniturePizza » car « GarnitureTomates » est une sous-classe de « GarnitureLegumes » qui est une sous-classe de « GarniturePizza ».

Le point

Nous avons créé quelques classes simples, certaines sont des sous-classes dans la hiérarchie. Jusqu'ici la construction de la hiérarchie des classes peut sembler facile et intuitive.

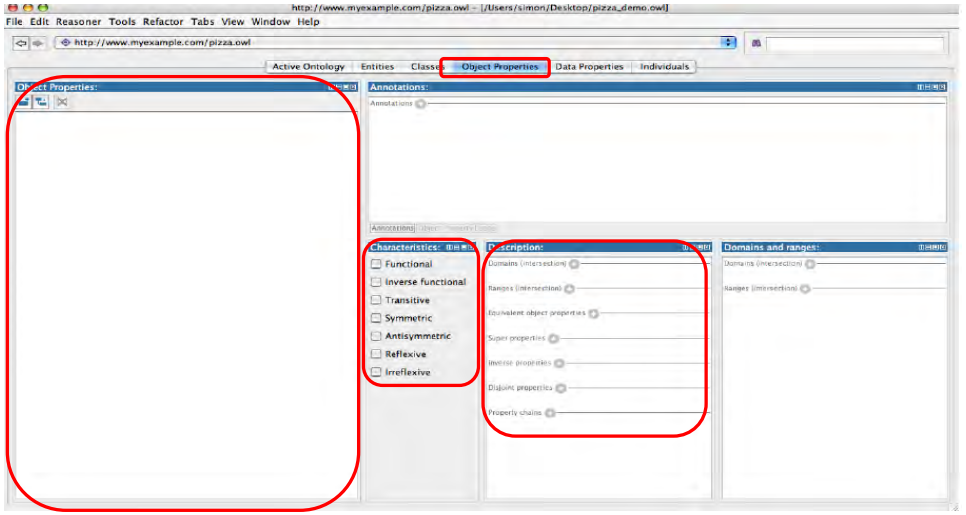
Cependant, qu'est-ce que cela signifie d'être une sous-classe d'une autre en OWL? Par exemple, qu'est ce que cela implique pour « *GarnitureLegumes* » d'être une sous-classe de « *GarniturePizza* », ou pour « *GarnitureTomates* » d'être une sous-classe de « *GarnitureLegumes* »?

En OWL être une sous-classe a nécessairement des implications. En d'autres termes, si « *GarnitureLegumes* » est une sous-classe de « *GarniturePizza* » alors toutes les instances de « *GarnitureLegumes* » sont des instances de « *GarniturePizza* », sans exception — si quelque chose est une « *GarnitureLegumes* » alors c'est aussi une « *GarniturePizza* ».

Création d'une ontologie avec Protégé 4

- Les relations

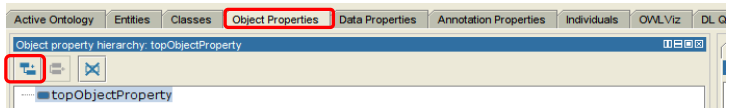
Les relations en OWL / OWL Properties



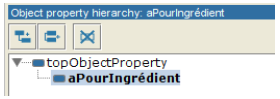
Créer une relation

- Objectif = créer une relation « *aPourIngrédient* »

1. Allez sur l'onglet « *Object Properties* ». Cliquez sur « *Add Sub-Property* » pour créer une nouvelle relation.



2. Nommez la nouvelle propriété « *aPourIngredient* » en remplissant la boîte de dialogue qui s'affiche.

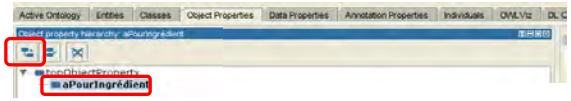


Créer des sous-relations

- Objectif = créer les relations « *aPourGarniture* » et « *aPourBase* » comme sous-relations de la relation « *aPourIngredient* »

Le langage OWL permet de définir des sous relations et donc de créer des hiérarchies de relations comme on a des hiérarchies de classes. Les sous-relations (propriétés) sont des spécialisations de leur relation mère.

1. Allez sur l'onglet « *Object Properties* ». Sélectionnez « *aPourIngredient* » *puis* cliquez sur « *Add Sub-Property* » pour créer une nouvelle relation.



2. Nommez la nouvelle propriété « *aPourGarniture* » en remplissant la boîte de dialogue qui s'affiche.

3. Faire de même pour la relation « *aPourBase* »

Création d'une ontologie avec Protégé 4

- Les caractéristiques des relations

Les différentes caractéristiques des relations en OWL

- **Fonctionnelle**
  - Une relation est déclarée fonctionnelle lorsqu'un seul individu au plus peut y être relié.
    - Jean hasBirthMother Catherine. HasBirthMother est fonctionnelle.
- **Inverse**
  - Chaque relation peut avoir une relation inverse
    - La relation hasParent a pour inverse la relation hasChild, et réciproquement.
- **Fonctionnelle inverse**
  - Cela veut dire que la relation inverse est déclarée fonctionnelle
    - Exemple : isBirthMotherOf
- **Transitive**
  - Peter hasAncestor Mat. Mat hasAncestor Jim => Peter hasAncestor Jim.
- **Symétrique**
  - Mat hasSibling Jim => Jim hasSibling Mat.
- **Asymétrique**
  - Robert isChildOf Peter => Peter is not a child of Robert.
- **Réflexive**
  - Georges knows George
- **Irréflexive**
  - Il est impossible que A soit en relation avec A. hasMother

Créer des relations inverses

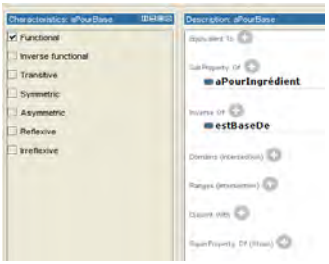
- Pour créer une ontologie complète, nous allons spécifier les relations inverses correspondantes à celles existantes.

1. Créez la relation « *estIngredientDe* » qui deviendra l'inverse de « *aPourIngredient* ».
2. Cliquez sur le **+** à droite de « *Inverse Of* ». Cette action ouvre une boîte de dialogue dans laquelle vous allez devoir sélectionner la relation inverse (ici « *aPourIngredient* »).

3. Créez la relation « *estBase* [illegible] *aPourBase* ». Déclarez la inverse.
4. Répétez l'opération en créant « *estGarnitureDe* », l'inverse de la relation « *aPourGarniture* ».

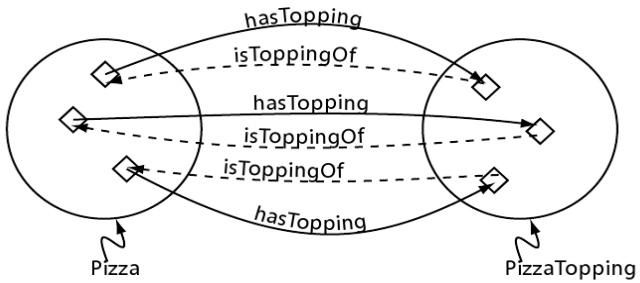
Caractériser les relations

- la relation « *aPourIngredient* » est **transitive**
- la relation « *aPourBase* » est **fonctionnelle**



Domaine et Portée

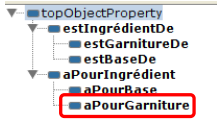
- En OWL, les relations sont **binaires** et interviennent pour mettre en relation deux individus :
  - [Domaine] (Relation) [Portée]
  - *[Pizza] (aPourGarniture) [GarniturePizza]*



Domaine et Portée

- Objectif = définir le domaine et la portée (Domains/Ranges) de « *APourGarniture* »

1. Sélectionnez la relation « *aPourGarniture* » dans la hiérarchie.
2. Fixez la portée (co-domaine) : cliquez sur **+** à droite de « Ranges » et choisir la classe « *Pizza* »
3. Fixez le domaine : cliquez sur **+** à droite de « Domains » et choisir la classe « *GarniturePizza* »



Domaine et Portée

1. Sélectionnez la relation « *aPourBase* »
2. Spécifiez « *Pizza* » comme domaine et « *BasePizza* » comme portée
3. Sélectionnez la relation **inverse** « *estBaseDe* »
4. Spécifiez « *BasePizza* » comme domaine et « *Pizza* » comme portée

Le point

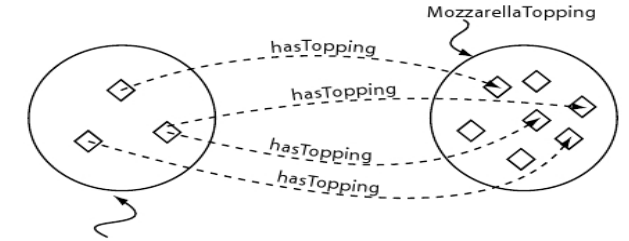
- Si l'on reprend l'exemple de la relation « *aPourGarniture* » :
  - Les individus qui appartiennent à la partie gauche de la relation sont interprétés comme étant membres de la classe « *Pizza* ».
  - Les individus qui apparaissent dans le contexte droit de la relation sont compris comme appartenant à la classe « *GarniturePizza* ».
- Nous venons de créer un certain nombre de relations.
- Nous allons maintenant les utiliser pour décrire et spécifier le fonctionnement des classes de notre ontologie.

Création d'une ontologie avec Protégé 4

- Les restrictions

Les deux types de restrictions (1)

- **Restriction existentielle** (*some*)
  - Les restrictions existentielles sont les plus courantes dans les ontologies OWL
  - Cette restriction décrit une classe d'individus qui entretient **au moins une** (some) relation avec un individu membre d'une classe spécifique



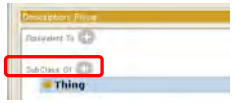
Les deux types de restrictions (2)

- La restriction **universelle** (*only*)
  - Cette restriction décrit des classes d'individus qui pour une relation donnée n'ont de lien **qu**'avec les individus d'une classe spécifique
  - Par exemple, la classe des individus qui n'ont comme garniture (*aPourGarniture*) que les individus appartenant à la classe des légumes (*GarnitureLegumes*).

Ajouter d'une restriction

- Objectif = ajouter une restriction spécifiant que « Pizza » doit avoir une « BasePizza »

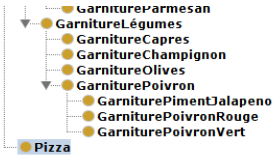
1. Sélectionnez la classe « *Pizza* » dans l'onglet « *Classes* »
2. Cliquez sur **+** à droite de « SubClasOf ».



3. Cela ouvre une boîte de dialogue contenant le « *Class expression editor* » dans lequel vous allez saisir votre restriction.
   a. Vous pouvez soit écrire « *aPourBase* » soit faire un « drag and drop ». Attention, la manière d'écrire le mot doit être la même que celle existante.
   b. Écrivez ensuite le type de restriction, ici « some ».
   c. Enfin, écrivez « *BasePizza* ».

Ajouter d'une restriction

Pizza et BasePizza sont disjointes mais cela n'interdit pas leur présence dans une même formule logique, pour peu qu'elle ne soit pas falsifiée (à l'inverse, p. ex., de « Pizza and BasePizza »)



Classes anonymes

- La restriction que l'on rajoute sur la pizza, à savoir qu'elle a une base de pizza, peut aussi être décrite comme *le fait qu'un individu de la classe Pizza doit être en relation avec un individu qui est membre de la classe PizzaBase par la propriété aPourBase*
- Cette restriction est en rdf, une *classe anonyme* de type owl:restriction

```
<owl:Class rdf:about="&untitled-ontology-135;Pizza">
<rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="&untitled-ontology-135;aPourBase"/>
        <owl:someValuesFrom rdf:resource="&untitled-ontology-135;BasePizza"/>
      </owl:Restriction>
  </rdfs:subClassOf>
 </owl:Class>
```

- Cette classe n'est pas référencée, pas posée dans la hiérarchie
- On parle de *blank node* ou *anonymous class (http://fr.wikipedia.org/wiki/Ressource_anonyme)*

A vous de jouer !

1. Sélectionnez la classe « *Pizza* » dans la hiérarchie
2. Ajoutez une sous-classe nommée « *PizzaNommee* »
3. Créez une sous-classe de « *PizzaNommee* » que vous appellerez « *MargheritaPizza* ».
4. Ajoutez un commentaire à cette classe en précisant qu'il s'agit d'une pizza composée de mozarella et d'une garniture tomate.

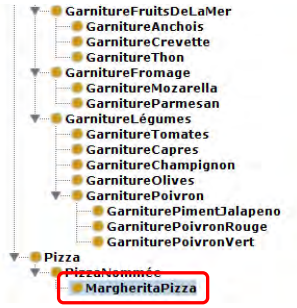
5. Sélectionnez « *MargheritaPizza* » dans la hiérarchie
6. Décrire cette classe comme ayant, comme garniture, de la mozarella.

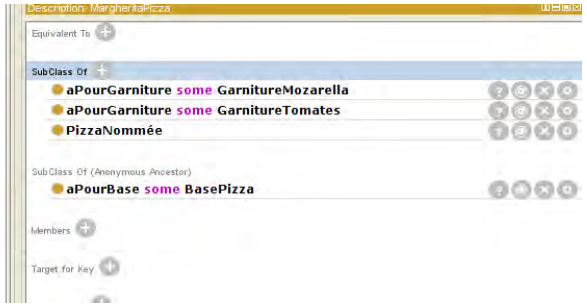
   *aPourGarniture **some** GarnitureMozarella*
7. Répétez l'opération pour dire qu'elle a aussi de la tomate

   *aPourGarniture **some** GarnitureTomates*

Cela veut dire que la Margherita contient au moins (au minimum) de la mozarella et de la tomate mais … elle peut contenir également d'autres garnitures. Créer les concepts qui peuvent manquer.

Résultat





En termes de classes anonymes

- En termes formels et en lisant les restrictions ligne par ligne, si qq chose est membre de la classe PizzaMargherita, il est nécessaire:
  - qu'il soit membre de la classe PizzaNommee

    Et
  - qu'il soit un membre de la classe anonyme des choses qui sont liés à au moins un membre de la classe GarnitureMozzarella via la propriété aPourGarniture

    Et
  - qu'il soit un membre de la classe anonyme des choses qui sont liés à au moins un membre de la classe GarnitureTomates via la propriété aPourGarniture

Monde ouvert *vs* monde fermé

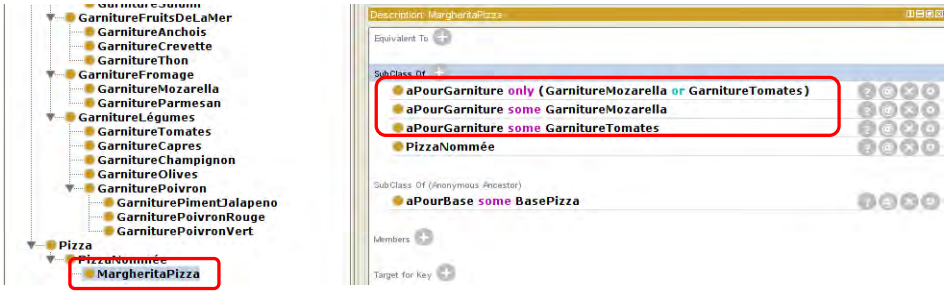
- Une Margherita contient au moins (au minimum) de la mozarella et de la tomate mais … Tant qu'on n'a rien écrit d'autre, elle peut contenir également d'autres garnitures.
- Ce n'est pas parce que quelque chose n'a pas été dit, qu'il est faux. Tout peut être vrai tant qu'il n'a pas été déclaré faux. Nous sommes dans l'*hypothèse du monde ouvert* quand nous créons des ontologies. Si l'on veut que quelque chose soit faux nous devons explicitement le dire.
- C'est le contraire de ce qui se passe dans une base de données.
- Dans une base de données, si une réponse à une requête ne ramène rien, alors la requête est dite falsifiée : ce qui n'est pas trouvé est faux. Dans une base de données, nous sommes dans l'*hypothèse du monde fermé*.
- Pour reproduire ce comportement dans une ontologie, il faut créer un *axiome de clôture*, une restriction universelle (only)

https://en.wikipedia.org/wiki/Closed-world_assumption

Créer une restriction universelle

1. Sélectionnez la « *MargheritaPizza* » dans la hiérarchie
2. Décrire cette classe comme ayant absolument une garniture à la mozarella OU une garniture à la tomate ET RIEN D'AUTRE.

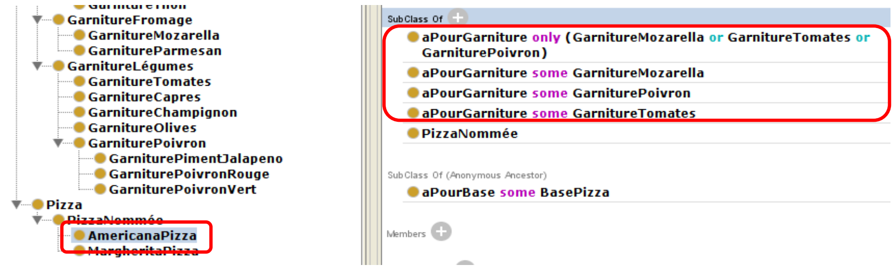
Résultat



Créer une AmericanaPizza

1. Sélectionnez la *MargheritaPizza* dans la hiérarchie
2. Enlevez l'axiome de clôture (avec le *only*)
3. Menu *Edit > Duplicate selected class*.
4. Renommez cette seconde classe en « *AmericanaPizza* »
5. Ajoutez la ou les restrictions pour dire que cette classe :
   – est composée uniquement de mozarella, de tomates et de poivrons

Résultat



Réponse à des questions (FAQ)

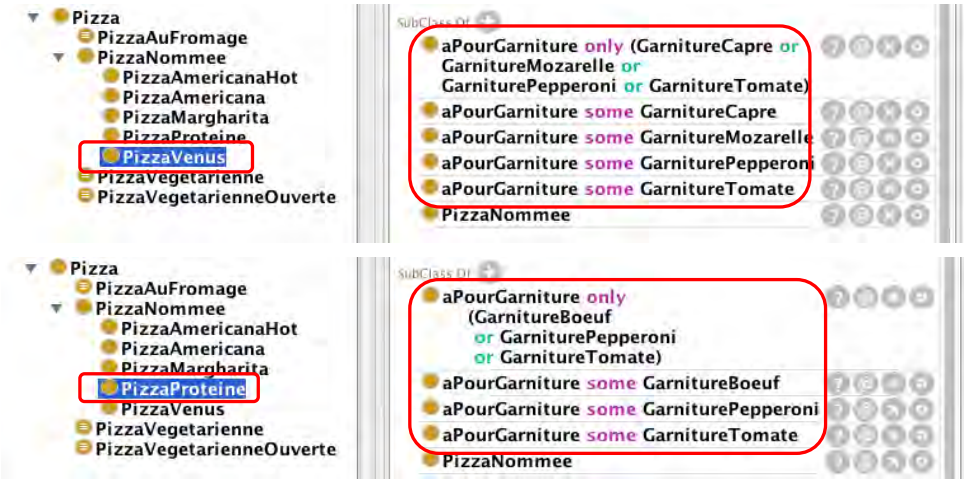
- *Pourquoi je ne <u>peux</u> pas écrire*
  aPourGarniture only (GarnitureFromage and GarnitureTomates) *?*
  Car il n'existe pas de garniture qui soit **à la fois** garniture au fromage et garniture au légume. En plus, c'est inconsistant avec aPourGarniture some …

- *Pourquoi je ne <u>veux</u> pas écrire*
  aPourGarniture only (GarnitureFromage or GarnitureTomates) *tout seul ?*
  Car dans ce cas-là, il suffit d'une seule des 2 garnitures (au fromage ou au légume) pour que la formule soit valide.

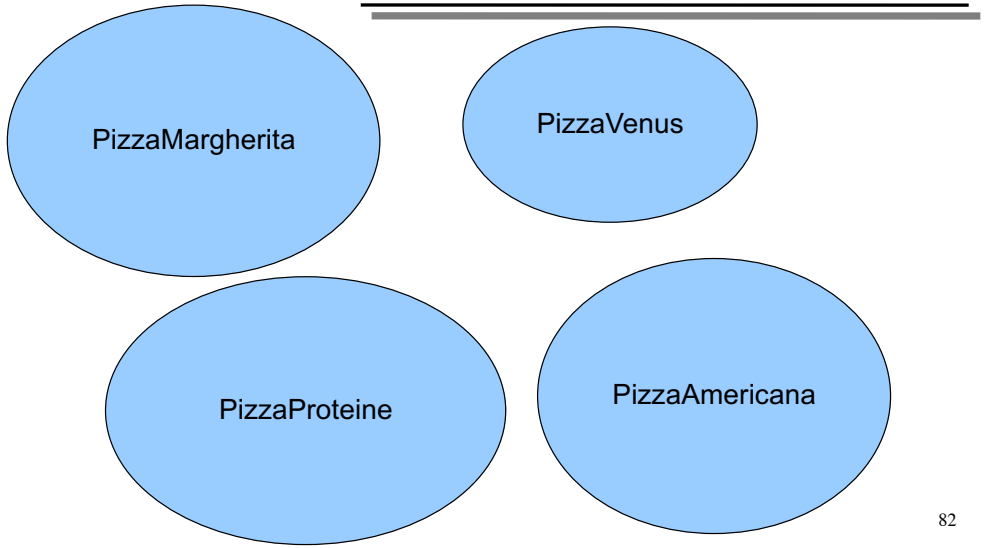
Créer une PizzaAmericanaHot, une PizzaVenus et une PizzaProteines

- La *PizzaAmericanaHot* est presque la même que la *PizzaAmericana*.
  - Dupliquez la *PizzaAmericana*, renommez-la et ajouter une restriction existentielle comme suit à la place du simple poivron :
    aPourGarniture some GarniturePimentJalapeno
- La *PizzaVenus* est une *PizzaMargherita* mais a, comme garniture supplémentaire des câpres et du pepperoni:
  aPourGarniture some GarnitureCapre
  aPourGarniture some GarniturePepperoni
- La *PizzaProteines* est une *pizza avec des tomates, du bœuf et du pepperoni*
  aPourGarniture some GarnitureTomate
  aPourGarniture some GarnitureBoeuf
  aPourGarniture some GarniturePepperoni

Résultat



Déclarer les sous-classes de PizzaNommeedisjointes des unes des autres



Création d'une ontologie avec Protégé 4

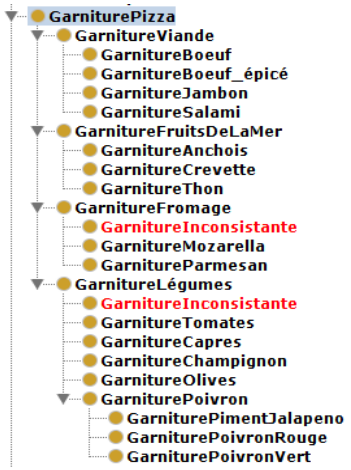
- Inconsistance des classes

Classes inconsistantes

- Objectif : créer une classe qui est une sous-classe de « *GarnitureFromage* » et aussi de « *GarnitureLegumes* ».

1. Sélectionnez la classe « *GarnitureFromage* »
2. Créez une sous-classe nommée « *GarnitureInconsistante* »
3. Sélectionnez « GarnitureInconsistante » dans la hiérarchie
4. Cliquez sur le **+** à droite de « SubClassOf » et sélectionnez la classe « *GarnitureLegumes* ». Validez.
5. La classe « *GarnitureLegumes* » a normalement été ajoutée à la liste des superclasses de « *GarnitureInconsistante* »
6. Lancer le raisonneur (Menu *Reasoner > Start Reasoner*)

Résultat



la classe GarnitureInconsitante
est inconsistante

Signification

- La classe *GarnitureInconsistante* est définie comme étant à la fois du fromage et des légumes. Or, aucune instance du monde ne peut être les deux à la fois. Cela dit, pour le raisonneur le nom des classes n'a pas de signification.
- L'inconsistance est détectée car nous avons déclaré les deux classes mères, *GarnitureFromage* et *GarnitureLegumes* comme étant disjointe l'une de l'autre.
- Autrement dit, les individus membres de la classe *GarnitureFromage* ne peuvent pas être membres de la classe *GarnitureLegumes* et vice-versa.

Enlever l'indication de disjonction

1. Enlever l'indication de disjonction entre *GarnitureLegumes* et *GarnitureFromage*
2. Relancer le classifieur

   → Normalement, la classe *GarnitureInconsistante* n'est plus inconsistante et n'est plus en rouge,

   → Désormais un individu du monde décrit peut être à la fois un membre de la classe *GarnitureFromage* et un membre de la classe *GarnitureLegumes*

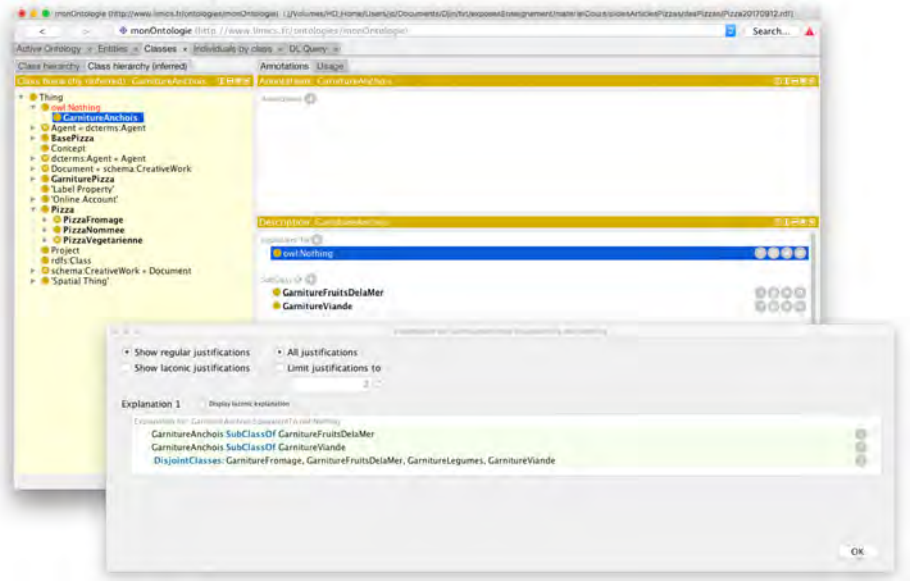
3. Remettre l'indication de disjonction.

Il convient donc d'être très prudent quand on spécifie (ou non) une disjonction.

Quand c'est "rouge", que faire ?

- Une incohérence se propage. Remonter à la source.
- Une classe peut avoir une condition non satisfaite dans une condition existentielle (*some)*
- Chaque sous-classe d'une classe insatisfiable est elle-même insatisfiable.
- Quelques possibilités :
  - Violations des axiomes d'exclusion mutuelle
  - Expressions insatisfiables
  - En particulier, confusion entre "and" et "or"
  - Violation d'une contrainte universelle (*only*), y compris domaine et co-domaine
  - *aPourGarniture fonctionnel et utilisé 2 fois*
  - *Des erreurs sur les (co-)domaines des relations*

Déboguer



Disjonction, exhaustivité et arbre

- Mettre des relations de disjonctions entre les concepts primitifs doit être fait quand on commence à penser que l'ontologie a une structure un peu stable
- Faire cela suppose qu'on a un arbre de concepts primitifs
- C'est pragmatique mais surtout épistémologiquement fondé : les objets n'ont qu'une seule essence, une seule manière d'être.
- Ces disjonctions n'empêchent pas d'utiliser 2 concepts exclusifs les uns des autres pour construire une représentation
  - *Une pizza et une garniture de pizza n'ont rien à voir l'une avec l'autre. Il n'y a pas d'instance qui soit à la fois Pizza et garniture mais une MargheritaPizza est bien une pizza avec une garniture (de fromage ou tomate).*

Création d'une PizzaAuFromage

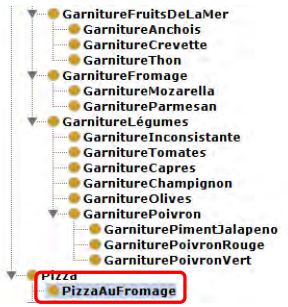
1. Créer une « *PizzaAuFromage* » comme sous-classe de « Pizza » (elle a donc pour classe sœur « *PizzaNommee* »)
2. Définissez-la comme étant une pizza avec du fromage :
   (aPourGarniture some GarnitureFromage)
3. Et ensuite, que fait-on ?

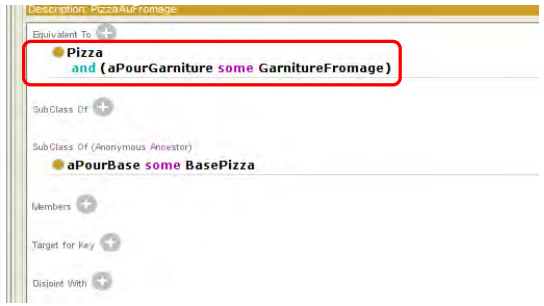
On veut qu'une pizza contenant du fromage puisse être une *PizzaAuFromage.*
Nota : *On ne veut pas modéliser les pizzas qui n'ont que du fromage, on veut modéliser les pizzas qui ne sont définies que par le fait d'avoir du fromage*

4. Nous souhaitons donc modifier la définition pour dire qu'une pizza au fromage **a comme condition nécessaire et suffisante** de contenir du fromage

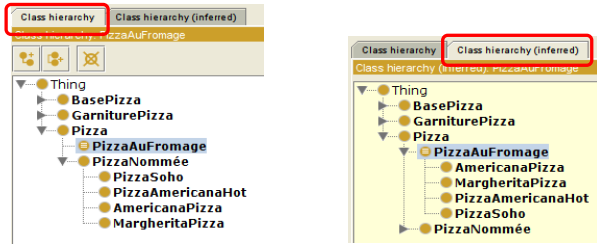
Critère nécessaire et suffisant → classe définie

1. Sélectionnez « PizzaAuFromage » dans la hiérarchie
2. Menu *Edit > Convert to defined class*
3. Relancer le raisonneur





Résultat pour la PizzaAuFromage



Conditions nécessaires *vs* nécessaires et suffisantes I

- Jusqu'à la *PizzaAufromage*, les conditions exprimées dans les descriptions de classes ont été des ***conditions nécessaires***. Pour ces classes, ces conditions sont celles qu'une instance doit remplir pour être un membre de la classe.
- Pour certaines classes, *e.g. PizzaAufromage*, ces conditions représentent la superclasse de la classe. Autrement dit, dans quelle classe mettre une instance dont on connaît les conditions exactes qu'elle remplit ? On la met dans une classe décrite par ***conditions nécessaires et suffisantes***, une classe définie.
- Autrement dit, par rapport au contexte de la pizza au fromage, on a besoin d'une classe d'équivalence pour contraindre les contraintes qui s'appliquent sur cette pizza, pour se mettre dans une situation où il n'y a QUE cette contrainte.
- OWL supporte ces conditions nécessaires et suffisantes qui permettent de déterminer que toute instance qui satisfait ces conditions peut être rangée, par inférence, dans la classe définie par ces mêmes conditions

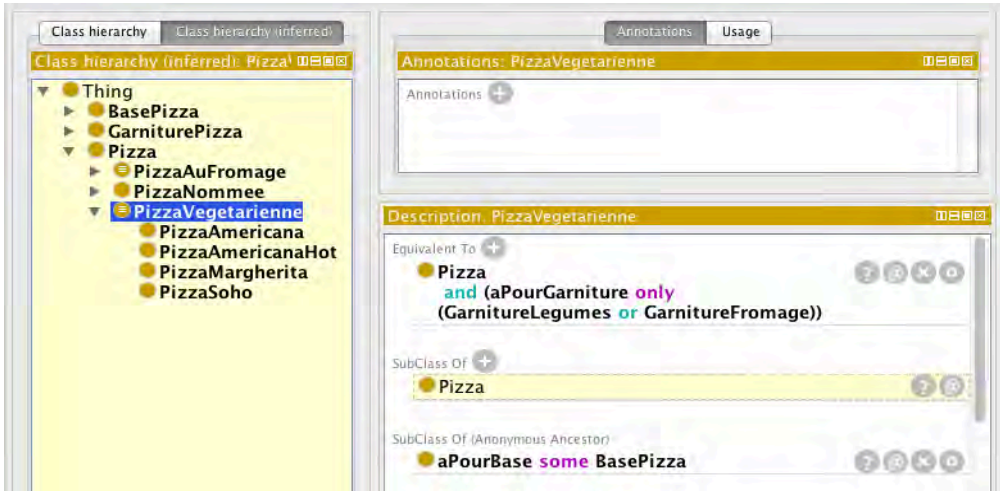
Conditions nécessaires *vs* nécessaires et suffisantes II

- On oppose ainsi les classes primitives, définies par conditions nécessaires, aux classe définies, définies par conditions nécessaires et suffisantes.
- Les classes qui ont seulement des conditions nécessaires supportent une description, les classes qui ont des conditions nécessaires et suffisantes ont une définition. Cette définition est plus précisément une définition logique précise.
- Une distinction peut être faite entre les informations axiomatiques (*asserted*) et les informations inférées (*inferred*).

A vous de jouer !!!

1. Créer une « *PizzaVegetarienne* » comme sous-classe de « *Pizza* » (elle a donc pour classe sœur « PizzaNommee »)
2. Définissez-la comme étant une pizza avec du fromage ou des légumes exclusivement (en réfléchissant comme pour la pizza au fromage)
3. Lancer le raisonneur.

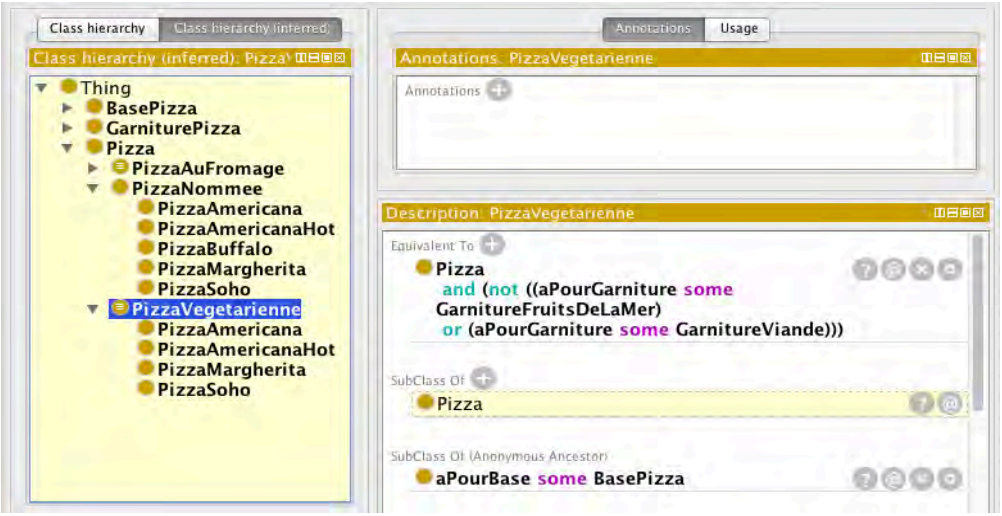
Résultat



Créez une pizza végétarienne (bis)

1. Créer une « *PizzaVegetarienneOuverte* » comme sous-classe de « Pizza » (elle a donc pour classe sœur « *PizzaNommee* »)
2. Définissez-la comme étant une pizza sans viande ni fruit de mer
3. Lancer le raisonneur
4. Qu'en concluez-vous ?

Résultat



… qui peut aussi s'écrire

Pizza and
(not (aPourGarniture some GarnitureViande))
and
(not (aPourGarniture some GarnitureFruitsDelaMer)))

Ou (écriture la plus compacte)

Pizza and (not (aPourGarniture some (GarnitureViande or GarnitureFruitsDelaMer)))

Attention !!

- Enlevez l'axiome de disjonction entre les garnitures
- Resynchronisez le raisonneur
- Expliquez le résultat

Attention !!

- Enlevez l'axiome de disjonction entre les garnitures
- Resynchronisez le raisonneur
- Expliquez le résultat

  - La modélisation par *not* ne « fonctionne » que si la disjonction des garnitures est axiomatisée

Comparaison des 2 modélisations

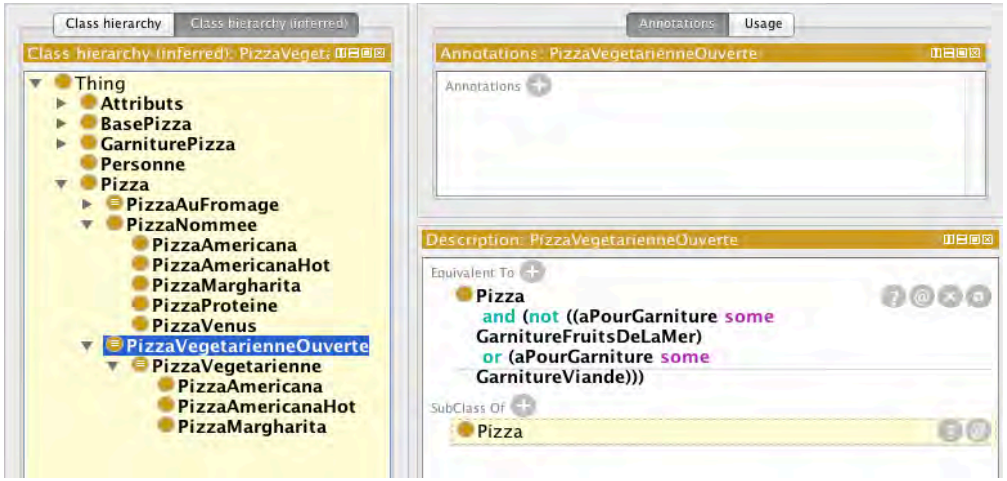
- Faites les 2 modélisations dans le même fichier (« PizzaVegetarienne » et « PizzaVegetarienneOuverte » et lancez le raisonneur. Qu'en concluez-vous et pourquoi ?

Comparaison des 2 modélisations

- Faites les 2 modélisations dans le même fichier (« PizzaVegetarienne » et « PizzaVegetarienneOuverte » et lancez le raisonneur. Qu'en concluez-vous et pourquoi ?

  - La modélisation par *not* est plus générale puisqu'elle laisse le monde ouvert… à l'arrivée de nouveaux ingrédients (*e.g. œufs)*

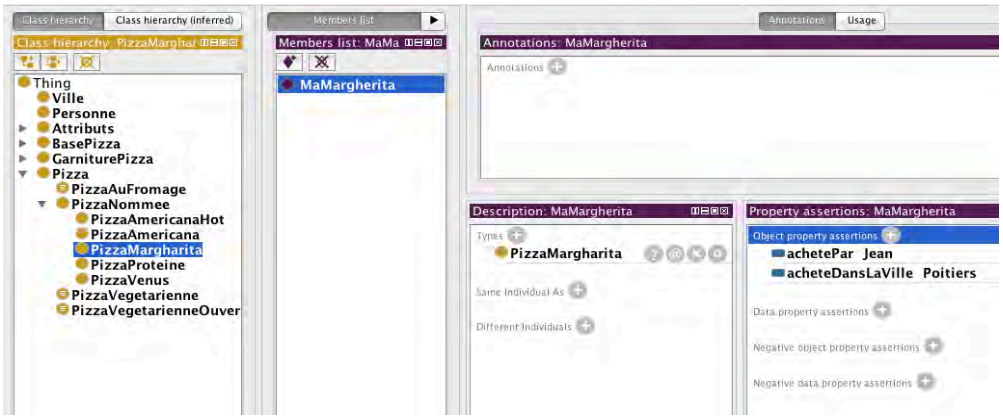
Résultat



23. Instances

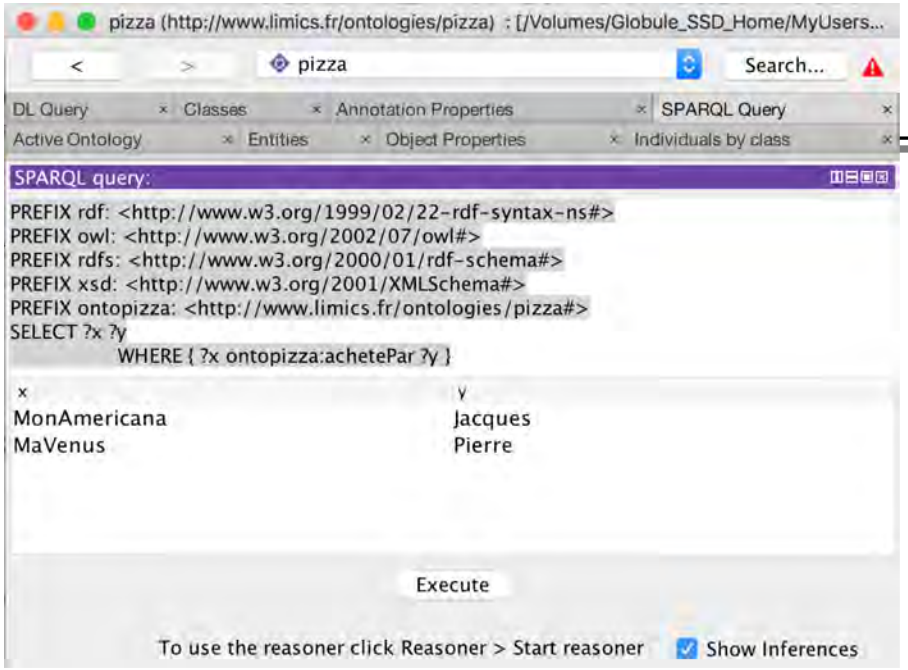
- Créer de nouvelles classes
  - Personne
  - Ville
- Créer de nouvelles relations (domaine : PizzaNommee)
  - achetePar
  - acheteDansLaVille
- Créer de quoi modéliser l'âge d'une personne (entrer le type integer)
- Créez des instances de PizzaNommee (précises), de Personne, de Ville
- Donnez pour chaque instance de pizza, un acheteur et une ville d'achat (ctrl-blanc pour compléter les fraguris)

Résultat



24. requêtes SPARQL sur les instances (1/3)

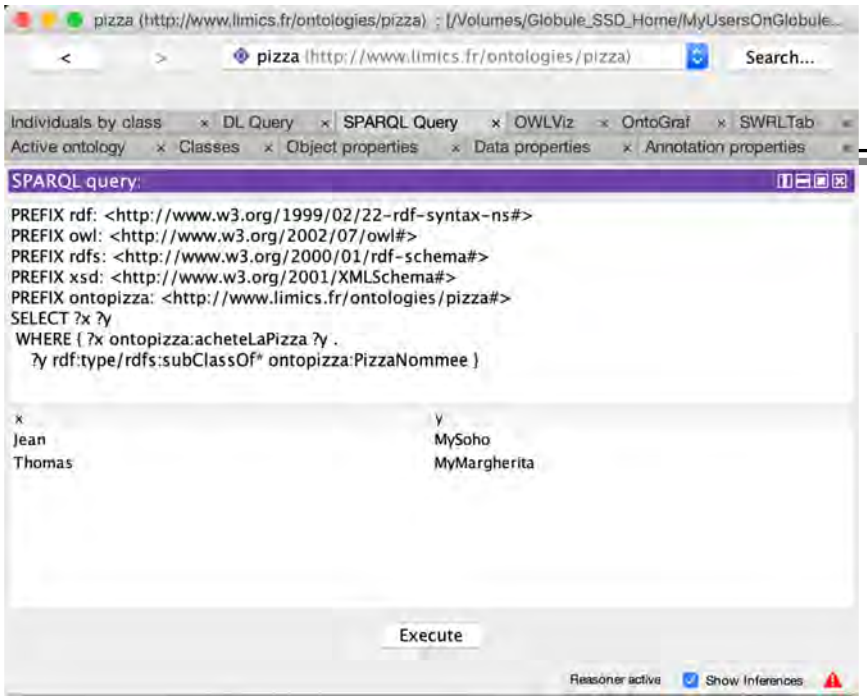
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ontopizza: <http://www.limics.fr/ontologies/pizza#>
SELECT ?x ?y
    WHERE { ?x ontopizza:achetePar ?y }
```



24. requêtes SPARQL sur les instances (2/3)

- Si les personnes peuvent acheter plusieurs choses dans la base, je peux vouloir ne m'intéresser qu'aux instances de certaines classes (ici des pizzas)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ontopizza: <http://www.limics.fr/ontologies/pizza#>
SELECT ?x ?y
    WHERE  { ?x ontopizza:acheteLaPizza ?y .
      ?y rdf:type/rdfs:subClassOf* ontopizza:PizzaNommee }
```



24. requêtes SPARQL sur les instances (3/3)

```
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX owl: http://www.w3.org/2002/07/owl#
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#
PREFIX xsd: http://www.w3.org/2001/XMLSchema#
SELECT ?subject ?object
        WHERE { ?subject rdfs:subClassOf ?object }
```

24.2. requêtes SPARQL sur Dbpedia (1/8)

- SPARQL endpoint : https://dbpedia.org/sparql
- Affichage HTML
- Export dans différents formats, dont CSV

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?film WHERE {
?film dbo:director dbp:Alfred_Hitchcock.
}
```

24.2. requêtes SPARQL sur Dbpedia (2/8)

La requête précédente renvoie des URL…

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?titre WHERE {
?film dbo:director dbp:Alfred_Hitchcock;
rdfs:label ?titre.
}
```

24.2. requêtes SPARQL sur Dbpedia (3/8)

La requête précédente renvoie toutes les langues…

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?titre WHERE {
?film dbo:director dbp:Alfred_Hitchcock;
rdfs:label ?titre.
FILTER (langMatches(lang(?titre), 'en'))
}
```

24.2. requêtes SPARQL sur Dbpedia (4/8)

On ajoute les têtes d'affiche…

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?titre ?acteur WHERE {
?film dbo:director dbp:Alfred_Hitchcock;
rdfs:label ?titre;
dbo:starring ?a.
?a rdfs:label ?acteur.
FILTER (langMatches(lang(?titre), 'en'))
}
```

24.2. requêtes SPARQL sur Dbpedia (5/8)

La requête précédente renvoie les noms des acteurs dans toutes les langues…

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?titre ?acteur WHERE {
?film dbo:director dbp:Alfred_Hitchcock;
rdfs:label ?titre;
dbo:starring ?a.
?a rdfs:label ?acteur.
FILTER (langMatches(lang(?titre), 'en') &&
langMatches(lang(?acteur), 'en'))
}
```

24.2. requêtes SPARQL sur Dbpedia (6/8)

On souhaite une seule ligne par titre…

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?titre (sql:group_concat(?acteur, ',') as ?acteurs) WHERE {
?film dbo:director dbp:Alfred_Hitchcock;
rdfs:label ?titre;
dbo:starring ?a.
?a rdfs:label ?acteur.
FILTER (langMatches(lang(?titre), 'en') &&
langMatches(lang(?acteur), 'en'))
}
```

24.2. requêtes SPARQL sur Dbpedia (7/8)

On souhaite les résultats en ordre alphabétique…

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?titre (sql:group_concat(?acteur, ',') as ?acteurs) WHERE {
?film dbo:director dbp:Alfred_Hitchcock;
rdfs:label ?titre;
dbo:starring ?a.
?a rdfs:label ?acteur.
FILTER (langMatches(lang(?titre), 'en') &&
langMatches(lang(?acteur), 'en'))
}
ORDER BY ?titre
```

24.2. requêtes SPARQL sur Dbpedia (8/8)

On souhaite avoir le résumé quand il est disponible…

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/resource/>
SELECT ?titre ?abstract (sql:group_concat(?acteur, ',') as ?acteurs)
WHERE {
?film dbo:director dbp:Alfred_Hitchcock;
rdfs:label ?titre;
dbo:starring ?a.
?a rdfs:label ?acteur.
OPTIONAL{?film dbo:abstract ?abstract}
FILTER (langMatches(lang(?titre), 'en') &&
langMatches(lang(?acteur), 'en') && langMatches(lang(?abstract),
'en'))
}
```

24.3. requêtes SPARQL sur wikidata (1/6)

- SPARQL endpoint : https://query.wikidata.org/
- Affichage HTML
- Export dans différents formats, dont CSV
- Possibilités de visualisation supplémentaires (#defaultView)
- Mais identifiants moins *user friendly*

```
SELECT ?eyeColor ?human
WHERE{
  ?human wdt:P31 wd:Q5.
  ?human wdt:P1340 ?eyeColor.
}
```

24.3. requêtes SPARQL sur wikidata (2/6)

On veut plutôt la fréquence des couleurs d'yeux…

```
SELECT ?eyeColor (COUNT(?human) AS ?count)
WHERE{
  ?human wdt:P31 wd:Q5.
  ?human wdt:P1340 ?eyeColor.
}
GROUP BY ?eyeColor
```

24.3. requêtes SPARQL sur wikidata (3/6)

On veut les fréquences par ordre croissant…

```
SELECT ?eyeColor (COUNT(?human) AS ?count)
WHERE{
  ?human wdt:P31 wd:Q5.
  ?human wdt:P1340 ?eyeColor.
}
GROUP BY ?eyeColor
ORDER BY DESC(?count)
```

24.3. requêtes SPARQL sur wikidata (4/6)

Et on préférerait avoir le label en français quand il existe…
(il suffit d'ajouter « Label » au nom de la variable et d'utiliser le SERVICE wikibase:label)

```
SELECT ?eyeColor ?eyeColorLabel (COUNT(?human) AS ?count)
WHERE{
  ?human wdt:P31 wd:Q5.
  ?human wdt:P1340 ?eyeColor.
   SERVICE wikibase:label { bd:serviceParam wikibase:language
"[AUTO_LANGUAGE],en". }
}
GROUP BY ?eyeColor ?eyeColorLabel
ORDER BY DESC(?count)
```

24.3. requêtes SPARQL sur wikidata (5/6)

Et une visualisation en bubblechart ?

```
#defaultView:BubbleChart
SELECT ?eyeColor ?eyeColorLabel (COUNT(?human) AS ?count)
WHERE{
  ?human wdt:P31 wd:Q5.
  ?human wdt:P1340 ?eyeColor.
   SERVICE wikibase:label { bd:serviceParam wikibase:language
"[AUTO_LANGUAGE],en". }
}
GROUP BY ?eyeColor ?eyeColorLabel
ORDER BY DESC(?count)
```

24.3. requêtes SPARQL sur wikidata (6/6)

Beaucoup plus d'exemples sur:

https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples

(« wikidata queries examples » dans un moteur de recherche)

25. Visualisation

- Chargez l'onglet « OntoGraf » si nécessaire

25. Arbres *versus* treillis (1/2)

- Créer les attributs d'ingrédients piquants, Epice, Moyen, Doux, ExtraDoux et la relation aPourQualiteDEtreEpice. Faites-en une value partition
- Créer les attributs d'ingrédients maigres, TresCalorique, MoyennementCalorique, PeuCalorique et la relation aPourQualiteDEtreCalorique. Faites-en une value partition
- Déclarez une Pizza *définie*, comme épicée si sa qualité est Epice
- Déclarez une Pizza *définie* comme calorique si sa qualité est TresCalorique ou Moyennement calorique

25. Arbres *versus* treillis (2/2)

- Précisez que la GarnitureBoeufEpice est Epice. Idem pour GarniturePoivreVert
- Précisez que GarnitureAnchoix et GarnitureFruits sont PeuCalorique
- Affichez l'arbre des GarniturePizza avant classification puis après classification
- Discutez

Monde ouvert *vs* monde fermé (suite)

- Une *value partition* est la façon de dire que l'ensemble des attributs listés est exhaustive … aucun autre attribut caractérisant le caractère épicé n'existe.
- C'est une façon de fermer le monde. C'est pour cela qu'on l'appelle *closure* axiom. Et ce n'est pas parce que quelque chose n'a pas été dit, qu'il est faux. Tout peut être vrai tant qu'il n'a pas été déclaré faux. Nous sommes dans l'*hypothèse du monde ouvert* quand nous créons des ontologies. Si l'on veut que quelque chose soit faux nous devons explicitement le dire.
- C'est le contraire de ce qui se passe dans une base de données.
- Dans une base de données, si une réponse à une requête ne ramène rien, alors la requête est falsifiée : ce qui n'est pas trouvé est faux. Dans une base de données, nous sommes dans l'*hypothèse du monde fermé*.
- Pour reproduire ce comportement dans une ontologie, il faut créer un *axiome de clôture*, une restriction universelle (only)

Annexes