

A Practical Guide To Building OWL Ontologies and Knowledge Graphs Using Protégé 5 Extended by Pace University

A Revision/Extension of Matthew Horridge's "A Practical Guide To Building
OWL Ontologies Using Protégé 4 and CO-ODE Tools Edition 1.3" for Protégé 5

Seidenberg School of Computer Science and Information Systems

Pace University

2015

Chapter 1

What are OWL Ontologies?

Ontologies are used to capture knowledge about some domain of interest. An ontology describes the concepts in the domain and also the relationships that hold between those concepts. Different ontology languages provide different facilities. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C). OWL makes it possible to describe concepts with a richer set of operators - e.g. intersection, union and negation. It makes it possible for concepts to be defined as well as described. Complex concepts can therefore be built up in definitions out of simpler concepts. Furthermore, the logical model allows the use of a reasoner which can check whether or not all of the statements and definitions in the ontology are mutually consistent and can also recognize which concepts fit under which definitions. The reasoner can therefore help to maintain the hierarchy correctly. This is particularly useful when dealing with cases where classes can have more than one parent.

1.1 Components of OWL Ontologies

An OWL ontology consists of Individuals, Properties, and Classes.

1.1.1 Individuals

Individuals, represent objects in the domain in which we are interested. OWL does not use the Unique Name Assumption (UNA). This means that two different names could actually refer to the same individual. For example, “Queen Elizabeth”, “The Queen” and “Elizabeth Windsor” might all refer to the same individual. In OWL, it must be explicitly stated that individuals are the same as each other, or different to each other — otherwise they might be the same as each other, or they might be different to each other. Figure 1.1 shows a representation of some individuals in some domain—in this tutorial we represent individuals as diamonds in diagrams. Individuals are also known as instances. Individuals can be referred to as being ‘instances of classes’.



Figure 1.1: Representation of Individuals

1.1.2 Properties

Properties are binary relations on individuals - i.e. properties link two individuals together. For example, the property hasSibling might link the individual Matthew to the individual Gemma, or the property

hasChild might link the individual Peter to the individual Matthew. Properties can have inverses. For example, the inverse of hasOwner is isOwnedBy. Properties can be limited to having a single value – i.e. to being functional. They can also be either transitive or symmetric. These “property characteristics” are explained in detail in Section 2.8. Figure 1.2 shows a representation of some properties linking some individuals together.

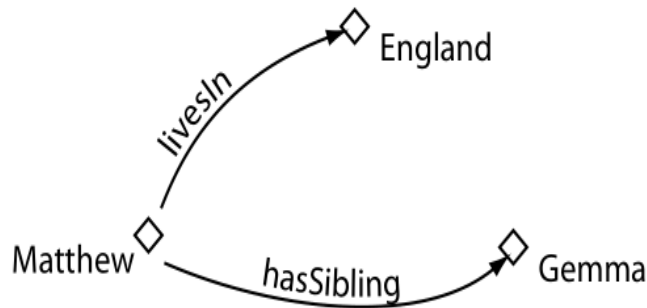


Figure 1.2: Representation of Properties

1.1.3 Classes

OWL classes are interpreted as sets that contain individuals. They are described using formal (mathematical) descriptions that state precisely the requirements for membership of the class. For example, the class Cat would contain all the individuals that are cats in our domain of interest. Classes may be organized into a superclass-subclass hierarchy, which is also known as a taxonomy. Subclasses specialize (“are subsumed by”) their superclasses. For example consider the classes Animal and Cat – Cat might be a subclass of Animal (so Animal is the superclass of Cat). This says that, “All cats are animals”, “All members of the class Cat are members of the class Animal”, “Being a Cat implies that you’re an Animal”, and “Cat is subsumed by Animal”. One of the key features of OWL-DL is that these superclass-subclass relationships (subsumption relationships) can be computed automatically by a reasoner – more on this later. Figure 1.3 shows a representation of some classes containing individuals – classes are represented as circles or ovals, rather like sets in Venn diagrams.

In OWL classes are built up of descriptions that specify the conditions that must be satisfied by an individual for it to be a member of the class. How to formulate these descriptions will be explained as the tutorial progresses.

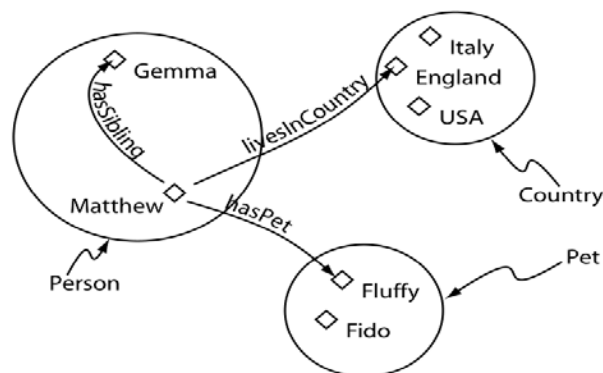


Figure 1.3: Representation of Classes (Containing Individuals)

Chapter 2

Building an OWL Ontology

This chapter describes how to create an ontology of Pizzas. We use Pizzas because we have found them to provide many useful examples.

Exercise 1: Create a new OWL Ontology

1. Start Protégé
2. A new OWL ontology is created by protégé automatically
3. Every ontology is named using an International Resource Identifier (IRI). Replace the default IRI with <http://www.pizza.com/ontologies/pizza.owl> in **Ontology IRI** of Active Ontology tab.
4. You will also want to save your Ontology to a file on your PC. You can click File|Save as ..., choose format 'RDF/XML' in the pop-up window, browse your hard disk for a folder, specify name for the new file (you might want to name your file 'pizza'), and click "Save" to save your ontology to a new file.

As can be seen from Figure 2.1, the “Active Ontology Tab” allows information about the ontology to be specified. For example, the ontology IRI can be changed, annotations on the ontology such as comments may be added and edited, and namespaces and imports can be set up via this tab.

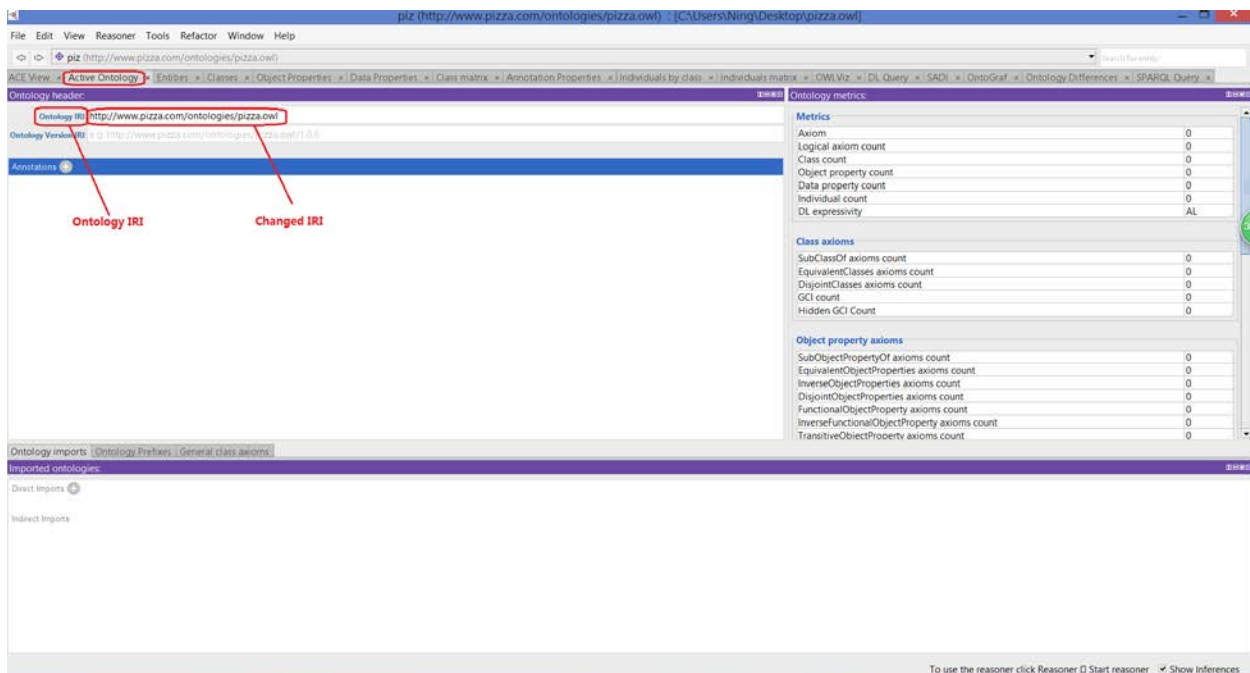


Figure 2.1: The Active Ontology Tab

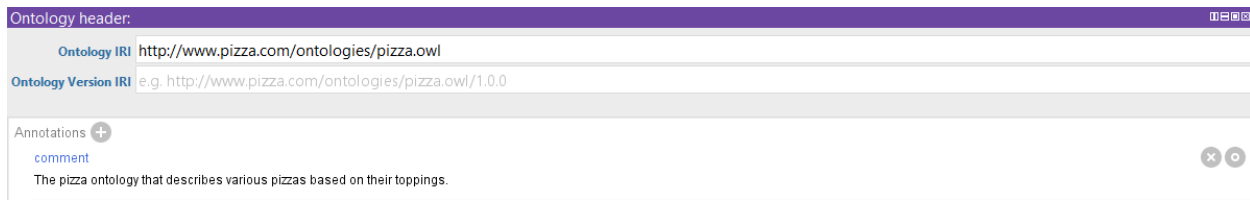



Figure 2.2: The Ontology Annotations

Exercise 2: Add a comment to the ontology

1. Ensure that the “Active Ontology Tab” is selected.
2. Click the “Add” icon () next to Annotations. An “Create Annotation” editing window will appear. Select ‘comment’ from the list of built in annotation URIs and type your comment in the text box in the right hand pane.
3. Enter a comment such as *The pizza ontology that describes various pizzas based on their toppings*. And press OK to assign the comment. The annotations view on the ‘Active Ontology Tab’ should look like the picture shown in Figure 2.2

2.1 Named Classes

As mentioned previously, an ontology contains classes – indeed, the main building blocks of an OWL ontology are classes. In Protégé 5.0, editing of classes is carried out using the “Classes” Tab shown in Figure 2.3. If you cannot see it, you can enable it by checking “Windows|Tabs|Classes”. The initial class hierarchy tree view should resemble the picture shown in Figure 2.4. The empty ontology contains one class called Thing. As mentioned previously, OWL classes are interpreted as sets of individuals (or sets of objects). The class Thing is the class that represents the set containing all individuals. Because of this all classes are subclasses of Thing. Let’s add some classes to the ontology in order to define what we believe a pizza to be.

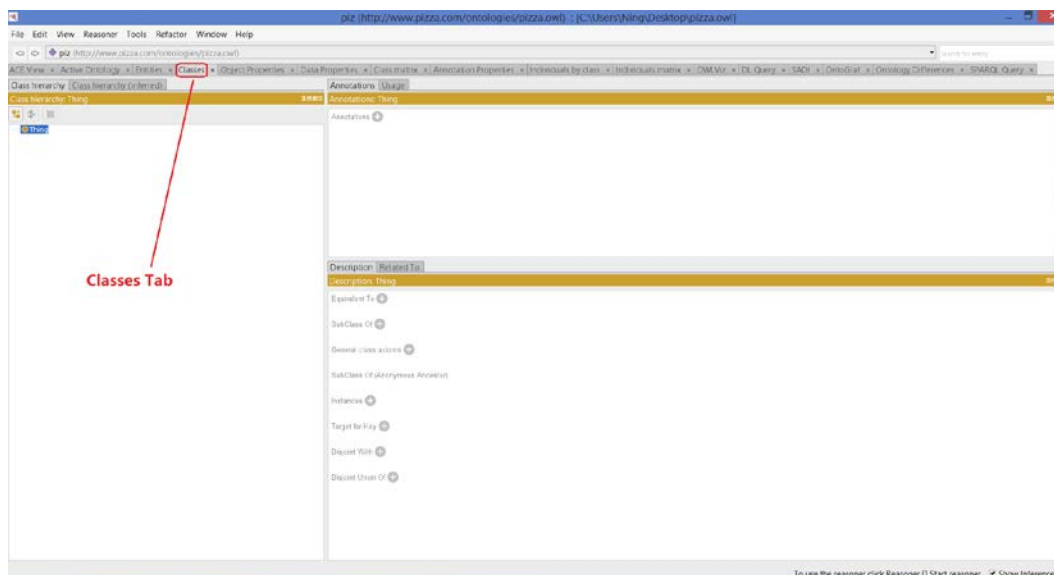


Figure 2.3: The Classes Tab

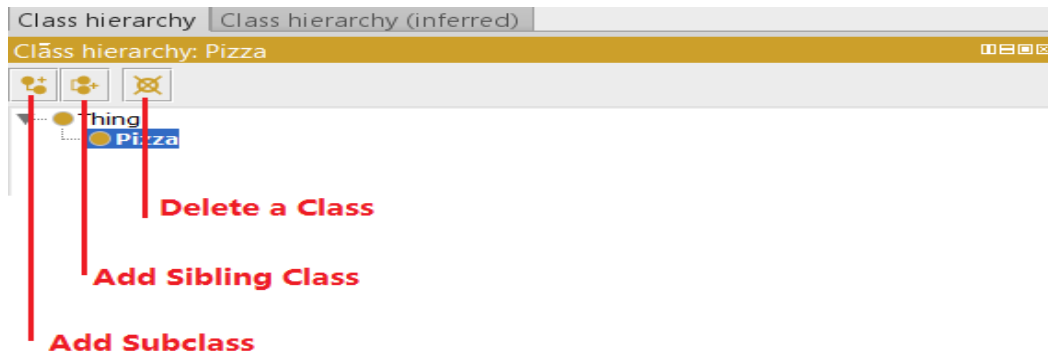




Figure 2.4: The Class Hierarchy Pane

Exercise 4: Create classes **Pizza**, **PizzaTopping** and **PizzaBase**

1. Ensure that the “Classes” tab is selected
2. Click to select “Thing”. Press the “Add subclass” icon () shown in Figure 2.4. This button creates a new class as a subclass of the selected class (in this case we want to create a subclass of Thing).
3. A dialog will appear for you to name your class, enter **Pizza** (as shown in Figure 2.5) and hit return.
4. Repeat the previous steps to add the classes **PizzaTopping** and also **PizzaBase**, ensuring that Thing is selected before the ‘Add’ icon () is pressed so that the classes are created as subclasses of **Thing**.

The class hierarchy should now resemble the hierarchy shown in Figure 2.6.

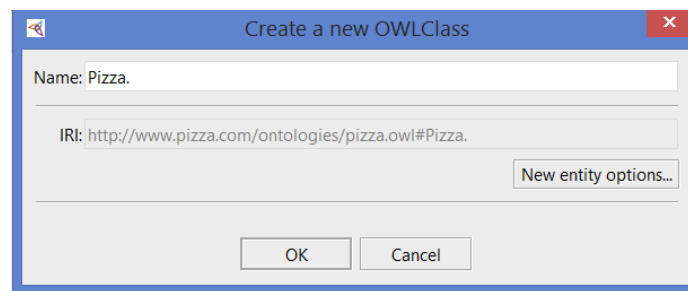


Figure 2.5: Class Name Dialog

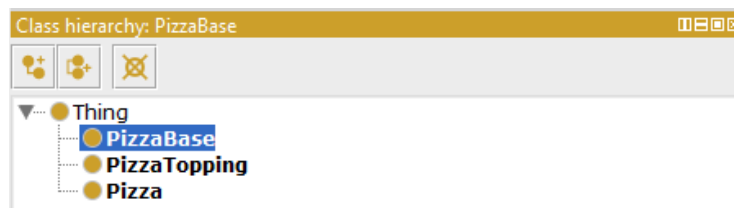




Figure 2.6: The Initial Class Hierarchy

2.2 Disjoint Classes

Having added the classes `Pizza`, `PizzaTopping` and `PizzaBase` to the ontology, we now need to say these classes are disjoint, so that an individual (or object) cannot be an instance of more than one of these three classes. To specify that a class is disjoint from other classes, select the class, then click the icon  next to ‘Disjoint With’ at the bottom of the “Class Description” view.

Exercise 4: Make `Pizza`, `PizzaTopping` and `PizzaBase` disjoint from each other

1. Select the class `Pizza` in the class hierarchy.
2. Press the “Disjoint With”  button in the “Class Description” view. This will bring up a dialog where you can select multiple classes to be disjoint. Expand Thing, and select `PizzaBase` and `PizzaTopping`, then click “OK” to close the dialogue. This will make `PizzaBase` and `PizzaTopping` (the sibling classes of `Pizza`) disjoint from `Pizza`.

Notice that the disjoint classes view now displays `PizzaBase` and `PizzaTopping`. Select the class `PizzaBase`. Notice that the disjoint classes view displays the classes that are now disjoint to `PizzaBase`, namely `Pizza` and `PizzaTopping`.

OWL Classes are assumed to ‘overlap’. We therefore cannot assume that an individual is not a member of a particular class simply because it has not been asserted to be a member of that class. In order to ‘separate’ a group of classes we must make them disjoint from one another. This ensures that an individual which has been asserted to be a member of one of the classes in the group cannot be a member of any other classes in that group. In our above example `Pizza`, `PizzaTopping` and `PizzaBase` have been made disjoint from one another. This means that it is not possible for an individual to be a member of a combination of these classes – it would not make sense for an individual to be a `Pizza` and a `PizzaBase`!

2.3 Using Create Class Hierarchy To Create Classes

In this section we will use the “Create Class Hierarchy” tool to add some subclasses of the class `PizzaBase`.

Exercise 5: Use the “Create Class Hierarchy” tool to create `ThinAndCrispy` and `DeepPan` as subclasses of `PizzaBase`

1. Select the class `PizzaBase` in the class hierarchy.
2. From the Tools menu on the Protégé menu bar select “Create class hierarchy...”
3. The “Create Class Hierarchy” window shown in Figure 2.7 will appear. Since we preselected the `PizzaBase` class, `PizzaBase` is now selected as the base class. You can now change the base class.
4. Press the ‘Continue’ button of the window — The page shown in Figure 2.8 will be displayed. We now need to tell the tool the subclasses of `PizzaBase` that we want to create. In the large text area, type in the class name `ThinAndCrispyBase` (for a thin based pizza) and hit return. Also enter the class name `DeepPanBase` so that the page resembles that shown in Figure 2.8.
5. Hit the “Continue” button on the tool. The tool checks that the names entered adhere to the naming styles (no spaces etc.). It also checks for uniqueness – no two class names may be the

same. If there are any errors in the class names, they will be presented on this page, along with suggestions for corrections.

6. Hit the “Continue” button on the tool. Ensure the tick box ‘Make all new classes disjoint’ is ticked — instead of having to use the disjoint classes view, the tool will automatically make the new classes disjoint for us.

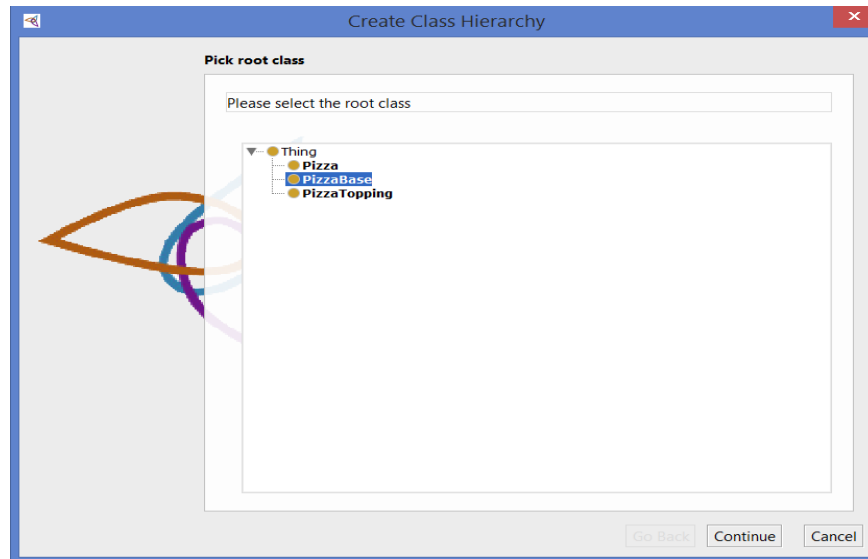


Figure 2.7: Create Class Hierarchy: Select class

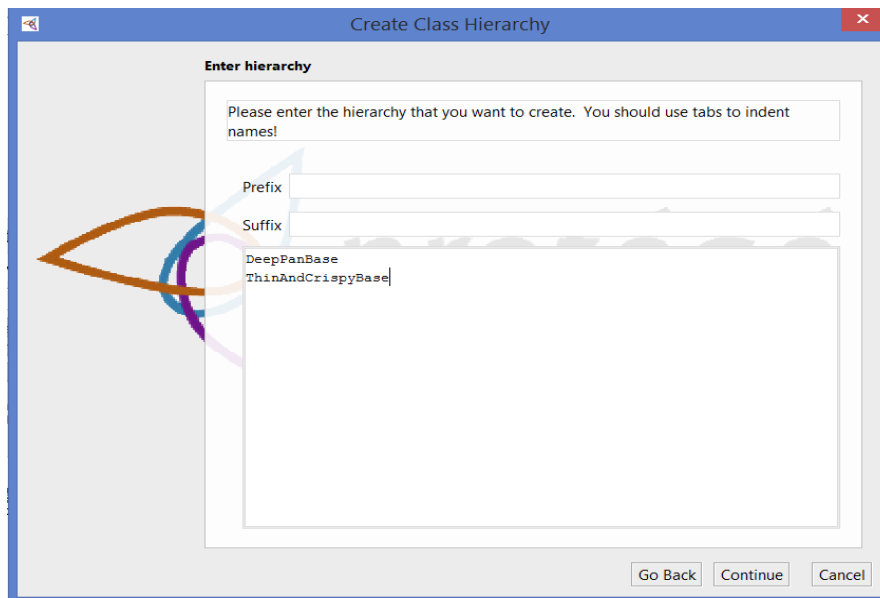


Figure 2.8: Create Class Hierarchy: Enter classes

Click ‘Finish’ to dismiss the tool. The ontology should now have ThinAndCrispyBase and also DeepPanBase as subclasses of PizzaBase. These new classes should be disjoint to each other. Hence, a

pizza base cannot be both thin and crispy and deep pan. It isn't difficult to see that if we had a lot of classes to add to the ontology, the tool would dramatically speed up the process of adding them.

On page one of the “Create class hierarchy” wizard, the classes to be created are entered. If we had a lot of classes to create that had the same prefix or suffix, we could use the options to auto prepend and auto append text to the class names that we entered.

Creating Some Pizza Toppings

Now that we have some basic classes, let's create some pizza toppings. In order to be useful later on, the toppings will be grouped into various categories — meat toppings, vegetable toppings, cheese toppings and seafood toppings.

Exercise 6: Create some subclasses of PizzaTopping

1. Select the class PizzaTopping in the class hierarchy.
2. Invoke the ‘Create class hierarchy...’ tool in the same way as the tool was started in the previous exercise.
3. Ensure PizzaTopping is selected and press the ‘Continue’ button
4. We want all our topping class names to end in “topping”, so in the “Suffix” field, enter Topping. The tool will save us some typing by automatically appending Topping to all of our class names.
5. The tool allows a hierarchy of classes to be entered using a tab indented tree. Using the text area in the tool, enter the class names as shown in Figure 2.9. Note that class names must be indented using tabs, so for example SpicyBeef, which we want to be a subclass of Meat, is entered under Meat and indented with a tab. Likewise, Pepperoni is also entered under Meat below SpicyBeef and also indented with a tab.
6. Having entered a tab indented list of classes, press the “Continue” button and then make sure that ‘**Make all primitive siblings disjoint**’ check box is ticked so that new sibling classes are made disjoint with each other.
7. Press the ‘Finish’ button to create the classes and close the tool.

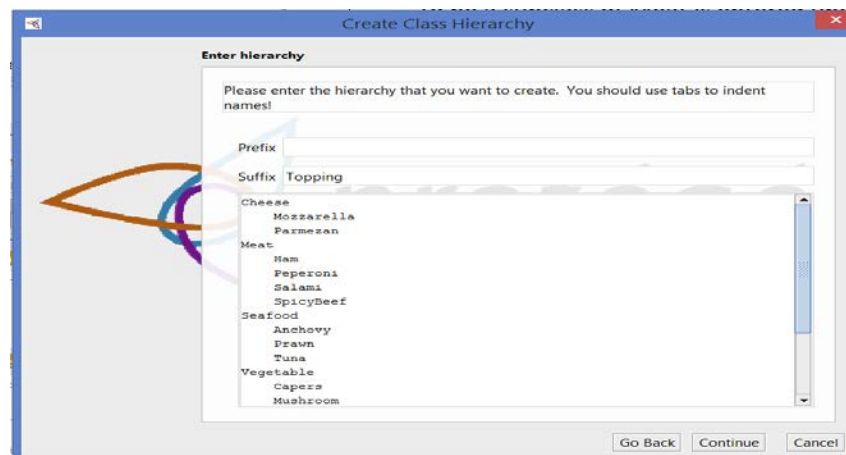
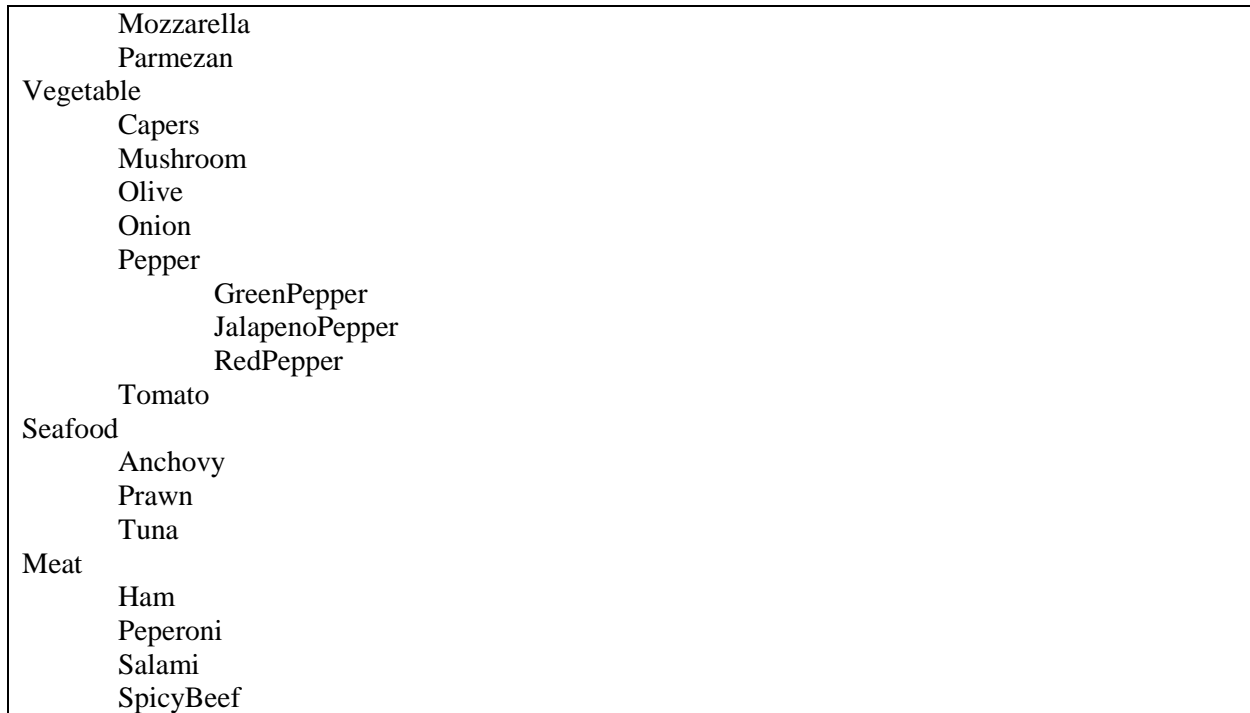


Figure 2.9: Topping Hierarchy

The input list is here. You can copy and paste to save your time.

Cheese



The class hierarchy should now look similar to that shown in Figure 2.10 (the ordering of classes may be slightly different).

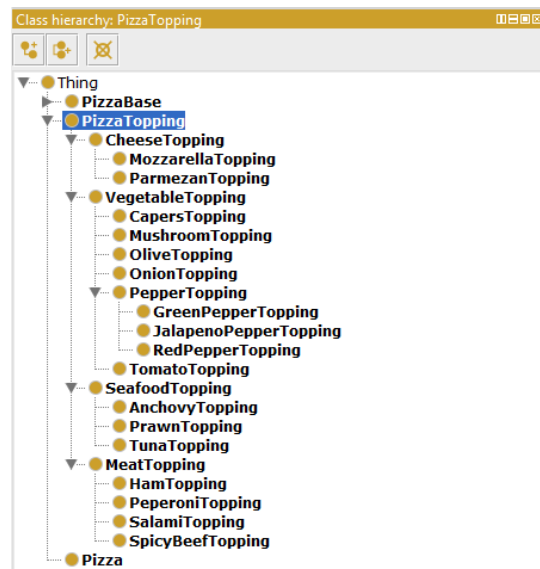


Figure 2.10: Class Hierarchy

2.4 OWL Properties

OWL Properties represent relationships. There are two main types of properties, Object properties and Datatype properties. Object properties are relationships between two individuals. Data type properties link the individuals to data literals. In this chapter we will focus on Object properties. Object properties link an individual to an individual. OWL also has a third type of property – Annotation properties. Annotation properties can be used to add information (metadata — data about data) to classes, individuals and object and datatype properties. Figure 2.11 depicts an example of each type of property.

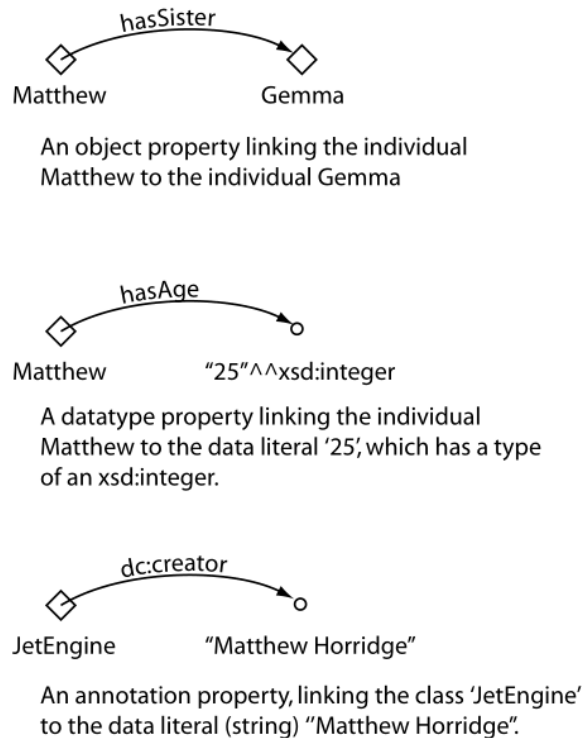


Figure 2.11: The Different types of OWL Properties

Properties may be created using the 'Object Properties' tab shown in Figure 2.12. If you don't see this tab, you can enable it by checking "Window/Tabs/Object Properties". Figure 2.13 shows the buttons located in the top left hand corner of the 'Object Properties' tab that are used for creating OWL properties. As can be seen from Figure 2.13, there are buttons for creating Datatype properties, Object properties and Annotation properties. Most properties created in this tutorial will be Object properties.

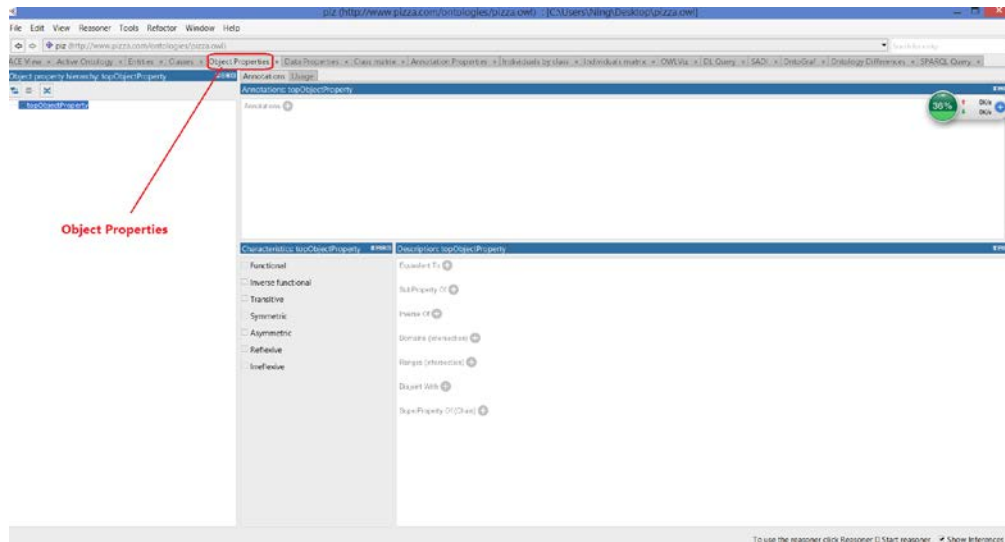


Figure 2.12: The Properties Tab

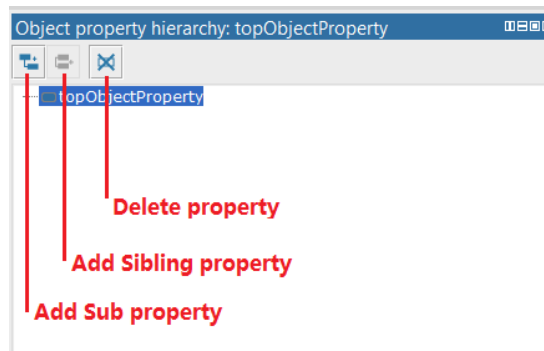



Figure 2.13: Property Creation Buttons

Exercise 8: Create an object property called hasIngredient

1. Switch to the “Object Properties” tab. Select “topObjectProperty” as the base object property. Use the ‘Add sub property’ button () to create a new object property.
2. Name the property to hasIngredient using the “Create a new OWLObjectProperty” dialogue that pops up, as shown in Figure 2.14 (The ‘Property Name Dialog’).

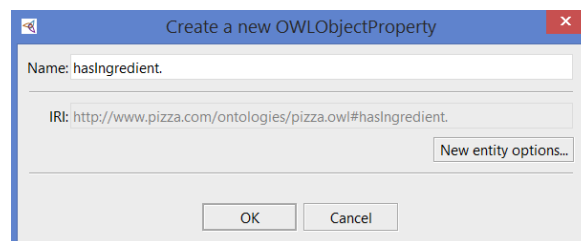



Figure 2.14: Property Name Dialog

Although there is no strict naming convention for properties, we recommend that property names start with a lower case letter, have no spaces and have the remaining words capitalized. We also recommend that properties are prefixed with the word ‘has’, or the word ‘is’, for example hasPart, isPartOf, hasManufacturer, isProducerOf. Not only does this convention help make the intent of the property clearer to humans, it is also taken advantage of by the “English Prose Tooltip Generator”, which uses this naming convention where possible to generate more human readable expressions for class descriptions.

Having added the hasIngredient property, we will now add two more properties — hasTopping, and hasBase. In OWL, properties may have sub properties, so that it is possible to form hierarchies of properties. Sub properties specialize their super properties (in the same way that subclasses specialize their superclasses). For example, the property hasMother might specialize the more general property of hasParent. In the case of our pizza ontology the properties hasTopping and hasBase should be created as sub properties of hasIngredient. If the hasTopping property (or the hasBase property) links two individuals, this implies that the two individuals are related by the hasIngredient property.

Exercise 8: Create hasTopping and hasBase as sub-properties of hasIngredient

1. To create the hasTopping property as a sub property of the hasIngredient property, select the hasIngredient property in the property hierarchy on the “Object Properties” tab.
2. Press the “Add sub property” button (). A new object property will be created as a sub property of the hasIngredient property.
3. Name the new property to hasTopping.
4. Repeat the above steps but name the property hasBase.(see Figure 2.15)

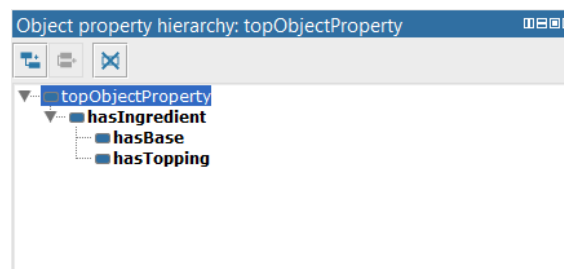


Figure 2.15: Create hasTopping and hasBase Properties

Note that it is also possible to create sub properties of datatype properties. However, it is not possible to mix and match object properties and datatype properties with regards to sub properties. For example, it is not possible to create an object property that is the sub property of a datatype property and vice-versa.

2.5 Inverse Properties

Each object property may have a corresponding inverse property. If some property links individual “a” to individual “b”, then its inverse property will link individual “b” to individual “a”. For example, Figure 2.16 shows the property hasParent and its inverse property hasChild — if Matthew hasParent Jean, then because of the inverse property we can infer that Jean hasChild Matthew.

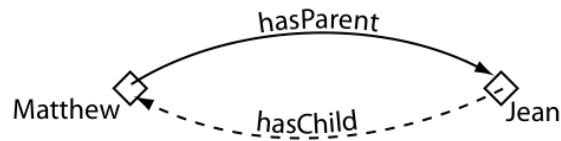


Figure 2.16: An Example of an Inverse Property

Inverse properties can be created/specified using the inverse property view shown in Figure 2.17. For completeness we will specify inverse properties for our existing properties in the Pizza Ontology.

Exercise 9: Create some inverse properties

1. Use the “Object Properties” tab to create a new Object property called `isIngredientOf` (this will become the inverse property of `hasIngredient`) as a sibling of `hasIngredient`.
2. While `isIngredientOf` is selected, press the ‘Add’ icon (+) next to “Inverse Of” on the “Property Description” view. This will display a dialog from which properties may be selected. Select the `hasIngredient` property and press “OK”. The property `hasIngredient` should now be displayed in the “Inverse Of” view.
3. Select the `hasBase` property.
4. Press the ‘Add’ icon (+) next to “Inverse Of” on the “Property Description” view. Create a new property in this dialog called `isBaseOf` as sub object property of `isIngredientOf`. Select this property and click ‘OK’. Notice that `hasBase` now has an inverse property assigned called `isBaseOf`.
5. Select the `hasTopping` property.
6. Press the ‘Add’ icon (+) next to “Inverse of” on the ‘Property Description’ view. Use the property dialog that pops up to create a new property `isToppingOf` as sub object property of `isIngredientOf`, and press ‘OK’.

Figure 2.17 shows the resulting object properties and sample inverse relationship.

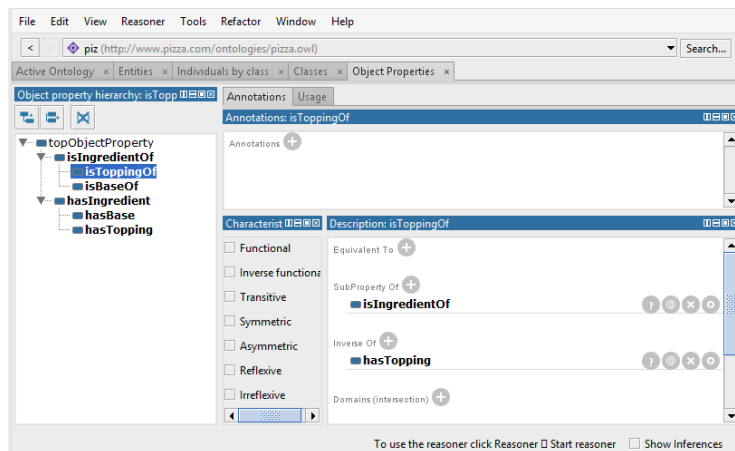


Figure 2.17: The Inverse Property View

2.6 OWL Object Property Characteristics

OWL allows the meaning of properties to be enriched through the use of property characteristics. The following sections discuss the various characteristics that properties may have:

2.6.1 Functional Properties

If a property is functional, for a given individual, there can be at most one individual that is related to the individual via the property. Figure 2.18 shows an example of a functional property `hasBirthMother`—anyone can only have one birth mother. If we say that the individual Jean `hasBirthMother` Peggy and we also say that the individual Jean `hasBirthMother` Margaret, then because `hasBirthMother` is a functional property, we can infer that Peggy and Margaret must be the same individual. It should be noted however, that if Peggy and Margaret were explicitly stated to be two different individuals then the above statements would lead to an inconsistency. Functional properties are also known as single valued properties and also features.

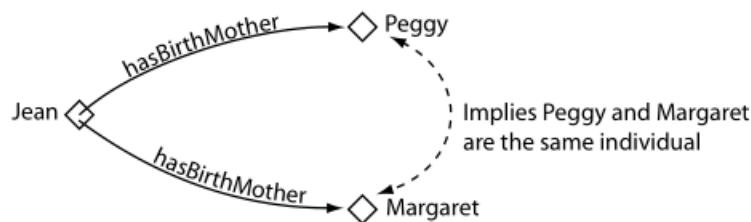


Figure 2.18: An Example Of A Functional Property: `hasBirthMother`

2.6.2 Inverse Functional Properties

If a property is inverse functional then it means that the inverse property is functional. For a given individual, there can be at most one individual related to that individual via the property. Figure 2.19 shows an example of an inverse functional property `isBirthMotherOf`. This is the inverse property of `hasBirthMother` — since `hasBirthMother` is functional, `isBirthMotherOf` is inverse functional. If we state that Peggy is the birth mother of Jean, and we also state that Margaret is the birth mother of Jean, then we can infer that Peggy and Margaret are the same individual.

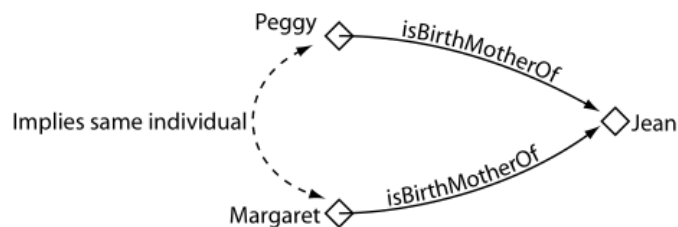


Figure 2.19: An Example of an Inverse Functional Property

2.6.3 Transitive Properties

If a property P is transitive, and the property relates individual a to individual b , and also individual b to individual c , then we can infer that individual a is related to individual c via property P . For example, Figure 2.20 shows an example of the transitive property `hasAncestor`. If the individual Matthew has an ancestor that is Peter, and Peter has an ancestor that is William, then we can infer that Matthew has an ancestor that is William – this is indicated by the dashed line in Figure 2.20.

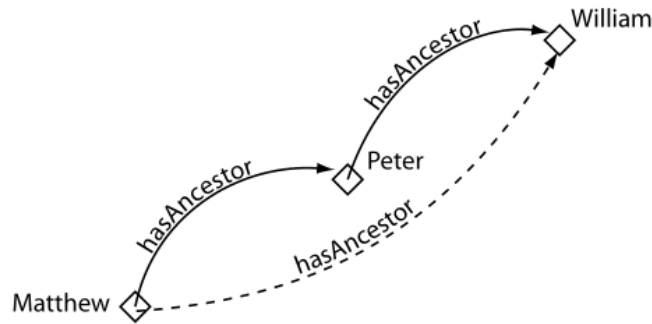


Figure 2.20: An Example of A Transitive Property: `hasAncestor`

2.6.4 Symmetric Properties

If a property P is symmetric, and the property relates individual a to individual b then individual b is also related to individual a via property P . Figure 2.21 shows an example of a symmetric property. If the individual Matthew is related to the individual Gemma via the `hasSibling` property, then we can infer that Gemma must also be related to Matthew via the `hasSibling` property. In other words, if Matthew has a sibling that is Gemma, then Gemma must have a sibling that is Matthew. Put another way, the property is its own inverse property.

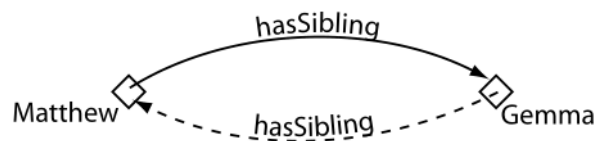


Figure 2.21: An Example of a Symmetric Property

2.6.5 Asymmetric Properties

If a property P is asymmetric, and the property relates individual a to individual b , then individual b cannot be related to individual a via property P . Figure 2.22 shows an example of an asymmetric property. If the individual Robert is related to the individual David via the `isChildOf` property, then it can be inferred that David is not related to Robert via the `isChildOf` property. It is, however, reasonable to state that David could be related to another individual Bill via the `isChildOf` property. In other words, if Robert is a child of David, then David cannot be a child of Robert, but David can be a child of Bill.

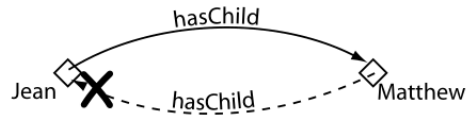


Figure 2.22: An example of the Asymmetric Property

2.6.6 Reflexive Properties

A property P is said to be reflexive when the property must relate any individual a to itself. In Figure 2.23 we can see an example of this: using the property `knows`, an individual George must have a relationship to itself using the property `knows`. In other words, George must know himself. However, in addition, it is possible for George to know other people; therefore the individual George can have a relationship with individual Simon along the property `knows`.

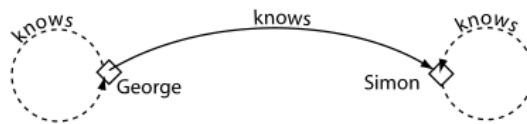


Figure 2.23: An example of a Reflexive Property

2.6.7 Irreflexive Properties

If a property P is irreflexive, it can be described as, if P relates an individual a to individual b , then individual a and individual b cannot be the same. An example of this would be the property `motherOf`: an individual Alice can be related to individual Bob along the property `motherOf`, but Alice cannot be `motherOf` herself (Figure 2.24).

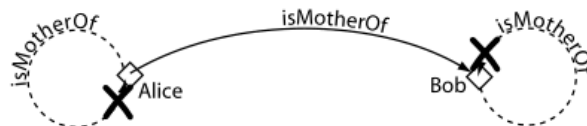


Figure 2.24: An example of an Irreflexive Property

We want to make the `hasIngredient` property transitive, so that for example if a pizza topping has an ingredient, then the pizza itself also has that ingredient. To set the property characteristics of a property, the property characteristics view shown in Figure 2.25, which is located in the lower right hand corner of the object properties tab, is used.

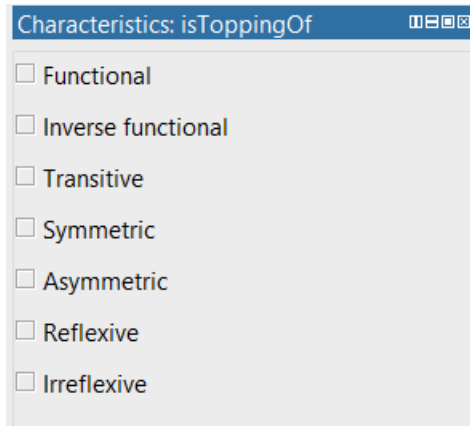


Figure 2.25: Property Characteristics Views

Exercise 10: Make the hasIngredient property transitive

1. Select the hasIngredient property in the property hierarchy on the “Object Properties” tab
2. Tick the “Transitive” tick box on the “Property Characteristics View”
3. Select the isIngredientOf property, which is the inverse of hasIngredient. Ensure that the transitive tick box is ticked

We now want to say that our pizza can only have one base. There are numerous ways that this could be accomplished. However, to do this we will make the hasBase property functional, so that it may have only one value for a given individual.

Exercise 11: Make the hasBase property functional

1. Select the hasBase property.
2. Click the “Functional” tick box on the “Property Characteristics View” so that it is ticked.

2.7 Property Domains and Ranges

Properties may have a domain and a range specified. Properties link individuals from the domain to individuals from the range. For example, in our pizza ontology, the property hasTopping would probably link individuals belonging to the class Pizza to individuals belonging to the class of PizzaTopping. In this case the domain of the hasTopping property is Pizza and the range is PizzaTopping— this is depicted in Figure 2.26.

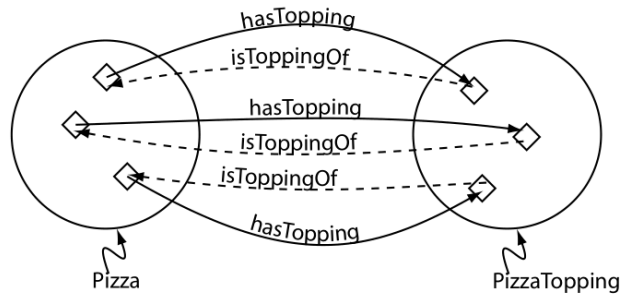



Figure 2.26: The Domain and Range for the hasTopping Property and its Inverse Property isToppingOf

In the previous steps we have ensured that the domains and ranges for properties are also set up for inverse properties in a correct manner. In general, domain for a property is the range for its inverse, and the range for a property is the domain for its inverse — Figure 2.26 illustrates this for the hasTopping and isToppingOf.

We now want to specify that the hasTopping property has a range of PizzaTopping. To do this, the range view shown in Figure 2.27 is used.

Exercise 12: Specify the range of hasTopping

1. Make sure that the hasTopping property is selected in the property hierarchy on the “Object Properties” tab.
2. Press the “Add” icon () next to “Ranges” on the “Property Description” view (Figure 2.27). A dialog will appear. Select its “Class hierarchy” tab. It will allow a class to be selected from the ontology class hierarchy.
3. Select PizzaTopping and press the “OK” button. PizzaTopping should now be displayed in the range list.

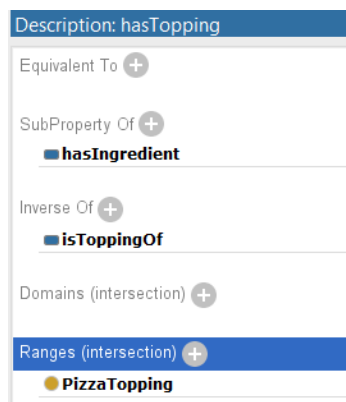



Figure 2.27: Property Range View (For Object Properties)

To specify the domain of a property, the domain view shown in Figure 2.28 is used.

Exercise 13: Specify Pizza as the domain of the hasTopping property

1. Make sure that the hasTopping property is selected in the property hierarchy on the “Object Properties” tab.
2. Press the “Add” icon () next to “Domains” on the “Property Description” view. A dialog will appear that allows a class to be selected from the ontology class hierarchy. Choose the “Class hierarchy” tab if needed.
3. Select Pizza and press the OK button. Pizza should now be displayed in the domain list.

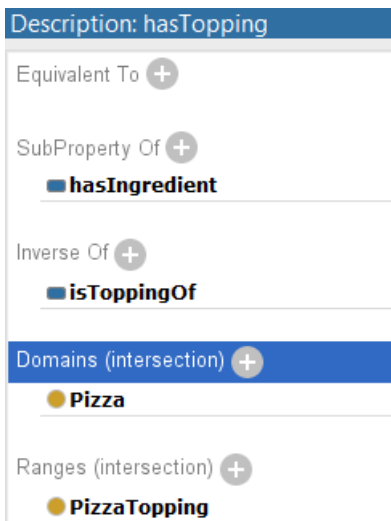


Figure 2.28: Property Domain View

Exercise 14: Specify the domain and range for the hasBase property and its inverse property isBaseOf

1. Select the hasBase property.
2. Specify the domain of the hasBase property as Pizza
3. Specify the range of the hasBase property as PizzaBase
4. Select the isBaseOf property. Notice that the domain of isBaseOf should be the range of the inverse property hasBase and that the range of isBaseOf should be the domain of the inverse property hasBase.
5. Make the domain of the isBaseOf property PizzaBase.
6. Make the range of the isBaseOf property Pizza.

2.8 Describing and Defining Classes

Having created some properties we can now use these properties to describe and define our Pizza Ontology classes.

2.8.1 Property Restrictions

Recall that in OWL, properties describe binary relationships. Datatype properties describe relationships between individuals and data values. Object properties describe relationships between two individuals. The key idea is that a class of individuals is described or defined by the relationships that these individuals participate in. In OWL we can define such classes by using restrictions.

Restriction Examples

Let's take a look at some examples to help clarify the kinds of classes of individuals that we might want to describe based on their properties.

- The class of individuals that have at least one hasSibling relationship.
- The class of individuals that have at least one hasSibling relationship to members of Man – i.e. things that have at least one sibling that is a man.
- The class of individuals that only have hasSibling relationships to individuals that are Women – i.e. things that only have siblings that are women (sisters).
- The class of individuals that have more than three hasSibling relationships.
- The class of individuals that have at least one hasTopping relationship to individuals that are members of MozzarellaTopping – i.e. the class of things that have at least one kind of mozzarella topping.
- The class of individuals that only have hasTopping relationships to members of VegetableTopping – i.e. the class of individuals that only have toppings that are vegetable toppings.

In OWL we can describe all of the above classes of individuals using restrictions. Restrictions in OWL fall into three main categories:

- Quantifier Restrictions
- Cardinality Restrictions
- hasValue Restrictions.

The restrictions for a class are displayed and edited using the “Class Description View”. The “Class Description View” is the ‘heart of’ the “Classes” tab in protégé, and holds virtually all of the information used to describe a class. At first glance, the “Class Description View” may seem complicated, however, it will become apparent that it is an incredibly powerful way of describing and defining classes. Restrictions are used in OWL class descriptions to specify anonymous superclasses of the class being described.

We will initially use quantifier restrictions, which can be further categorized into existential restrictions and universal restrictions. Both types of restrictions will be illustrated with examples throughout the tutorial.

Existential and Universal Restrictions

- Existential restrictions describe classes of individuals that participate in at least one relationship along a specified property to individuals that are members of a specified class. For example, “the class of individuals that have at least one (some) hasTopping relationship to members of MozzarellaTopping”. In Protégé 5 the keyword “some” is used to denote existential restrictions.
- Universal restrictions describe classes of individuals that for a given property only have relationships along this property to individuals that are members of a specified class. For example, “the class of individuals that only have hasTopping relationships to members of VegetableTopping”. In Protégé 5 the keyword “only” is used.

Let's take a closer look at the example of an existential restriction. The restriction hasTopping some MozzarellaTopping is an existential restriction (as indicated by the some keyword), which acts along the hasTopping property, and has a filler MozzarellaTopping. This restriction describes the class of individuals that have at least one hasTopping relationship to an individual that is a member of the class MozzarellaTopping. This restriction is depicted in Figure 2.29 — the diamonds in the Figure represent individuals. As can be seen from Figure 2.29, the restriction is a class which contains the individuals that satisfy the restriction.

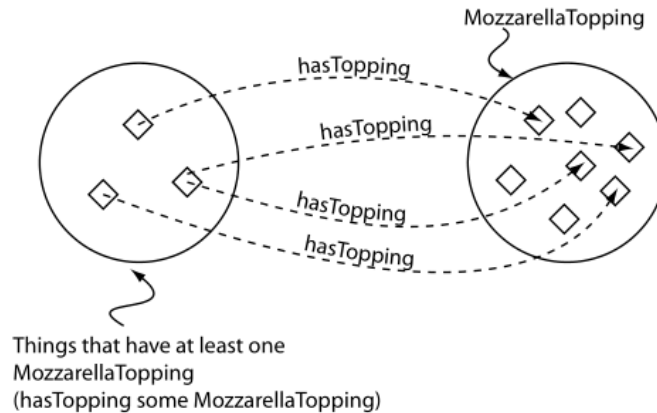


Figure 2.29: The Restriction `hasTopping some Mozzarella`

2.8.2 Existential Restrictions

Existential restrictions are by far the most common type of restrictions in OWL ontologies. An existential restriction describes a class of individuals that have at least one (some) relationship along a specified property to an individual that is a member of a specified class. For example, `hasBase some PizzaBase` describes all of the individuals that have at least one relationship along the `hasBase` property to an individual that is a member of the class `PizzaBase` — in more natural English, all of the individuals that have at least one pizza base.

Exercise 15: Add a restriction to `Pizza` that specifies a `Pizza` must have a `PizzaBase`

1. Select `Pizza` from the class hierarchy on the “Classes” tab.
2. Select the “Add” icon (⊕) next to “Subclass Of” header in the ‘Class Description View’ shown in Figure 2.30 in order to create a necessary condition. A “Pizza” window will pop up. Click its “Class expression editor” tab. The bottom text box is for creating restrictions.

The text box allows you to construct restrictions using class, property and individual names. You can drag and drop classes, properties and individuals into the text box, or type them in, the text box with check all the values you enter and alert you to any errors. To create a restriction we have to do three things:

- Enter the property to be restricted from the property list.
- Enter a type of restriction from the restriction types e.g. “some” for an existential restriction.
- Specify a filler for the restriction

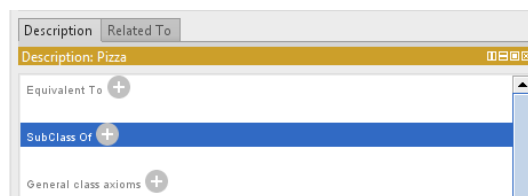


Figure 2.30: Creating a Necessary Restriction

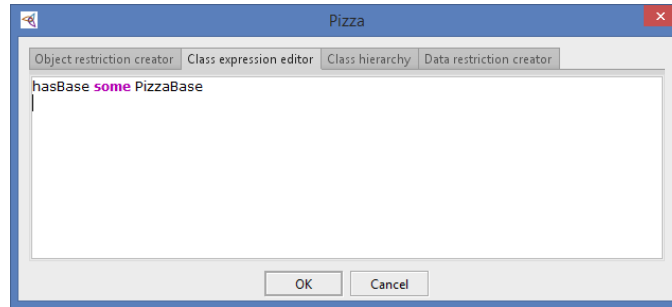


Figure 2.31: Creating a Restriction in the Text Box, with Auto-Complete

Exercise 16: Add a restriction to Pizza that specifies a Pizza must have a PizzaBase (Continued...)

1. You can either drag and drop hasBase from the property list into the create restriction text box, or type it in.
2. Now add the type or restriction, we will use an existential restriction so type “some”.
3. Specify that the filler is PizzaBase — to do this either: type PizzaBase into the filler edit box, or drag and drop PizzaBase into the text box as show in Figure 2.31
4. Press “OK” to create the restriction and close the create restriction text box. If all information was entered correctly the dialog will close and the restriction will be displayed in the “Class Description View”. If there were errors they will be underlined in red in the text box, or popup will give some hints to the cause of the error — if this is the case, recheck that the type of restriction, the property and filler have been specified correctly.

The class description view should now look similar to the picture shown in Figure 2.32

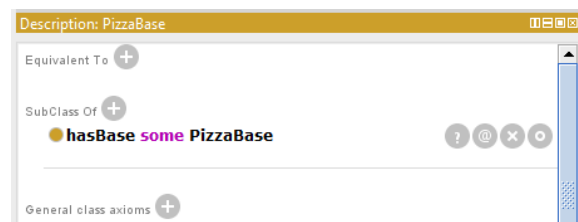



Figure 2.32: Class Description View: Description of a Pizza

Creating Some Different Kinds of Pizzas


It’s now time to add some different kinds of pizzas to our ontology. We will start off by adding a “MargheritaPizza”, which is a pizza that has toppings of mozzarella and tomato. In order to keep our ontology tidy, we will group our different pizzas under the class “NamedPizza”:

Exercise 17: Create a subclass of Pizza called NamedPizza, and a subclass of NamedPizza called MargheritaPizza

1. Select the class Pizza from the class hierarchy on the “Classes” tab.
2. Press the “Add subclass” icon () to create a new subclass of Pizza, and name it NamedPizza.
3. Create a new subclass of NamedPizza, and name it MargheritaPizza.
4. Add a comment to the class MargheritaPizza using the “Annotations” view that is located next to the class hierarchy view: A pizza that only has Mozzarella and Tomato toppings – it’s always a good idea to document classes, properties etc. during ontology editing sessions in order to communicate intentions to other ontology builders.


Having created the class MargheritaPizza we now need to specify the toppings that it has. To do this we will add two restrictions to say that a MargheritaPizza has the toppings MozzarellaTopping and TomatoTopping.

Exercise 18: Create an existential (some) restriction on MargheritaPizza that acts along the property hasTopping with a filler of MozzarellaTopping to specify that a MargheritaPizza has at least one MozzarellaTopping

1. Make sure that MargheritaPizza is selected in the class hierarchy.
2. Use the “Add” icon () on the “Subclass Of” section of the “Class Description view” to open a text box.
3. Type hasTopping as the property to be restricted in the text box.
4. Type “some” to create the existential restriction.
5. Type the class MozzarellaTopping as the filler for the restriction — remember that this can be achieved by typing the class name MozzarellaTopping into the filler edit box, or by using drag and drop from the class hierarchy.
6. Press “OK” to create the restriction — if there are any errors, the restriction will not be created, and the error will be highlighted in red.

Now specify that MargheritaPizzas also have TomatoTopping.

Exercise 20: Create an existential restriction (some) on MargheritaPizza that acts along the property hasTopping with a filler of TomatoTopping to specify that a MargheritaPizza has at least one TomatoTopping

1. Ensure that MargheritaPizza is selected in the class hierarchy.
2. Use the “Add” icon () on the “Subclass Of” section of the ‘Class Description View’ to open the text box.
3. Type hasTopping as the property to be restricted.
4. Type “some” to create the existential restriction.
5. Type the class TomatoTopping as the filler for the restriction.
6. Click “Enter” to create restriction dialog to create the restriction.

The “Class Description View” should now look similar to the picture shown in Figure 2.33

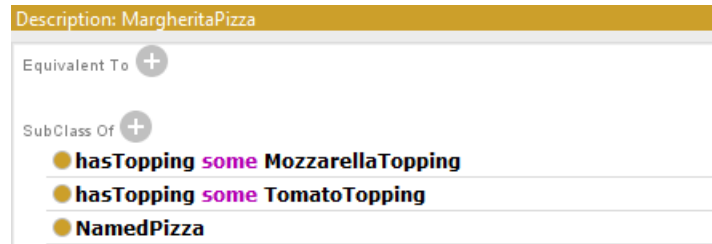



Figure 2.33: The Class Description View Showing a Description of a MargheritaPizza

Now create the class to represent an Americana Pizza, which has toppings of pepperoni, mozzarella and tomato. Because the class AmericanaPizza is very similar to the class MargheritaPizza (i.e. an Americana pizza is almost the same as a Margherita pizza but with an extra topping of pepperoni) we will make a clone of the MargheritaPizza class and then add an extra restriction to say that it has a topping of pepperoni.

Exercise 20: Create AmericanaPizza by cloning and modifying the description of MargheritaPizza

1. Select the class MargheritaPizza in the class hierarchy on the Classes tab.
2. Select Duplicate selected class from the 'Edit' menu. A dialog will appear for you to name the new class. Enter AmericanaPizza as the new class name. Uncheck "Duplicate annotations". Click "OK". The AmericanaPizza class will be created with exactly the same conditions (restrictions etc.) as MargheritaPizza.
3. Ensuring that AmericanaPizza is still selected, select the 'Add' icon () next to the "Subclass Of" header in the class description view, as we want to add a new restriction to the necessary conditions for AmericanaPizza.
4. Type the property hasTopping as the property to be restricted.
5. Type "some" to create the existential restriction.
6. Specify the restriction filler as the class PepperoniTopping by either typing PepperoniTopping into the text box, or by using drag and drop from the class hierarchy.
7. Press OK to create the restriction.

The "Class Description View" should now look like the picture shown in Figure 2.34

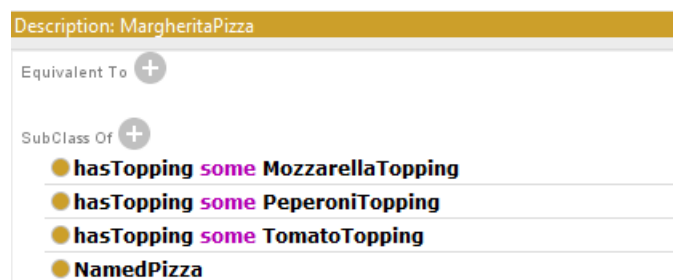


Figure 2.34: The Class Description View Displaying the Description for AmericanaPizza

Exercise 21: Create an AmericanHotPizza and a SohoPizza

1. An AmericanHotPizza is almost the same as an AmericanaPizza, but has Jalapeno peppers on it — create this by cloning the class AmericanaPizza and adding an existential restriction along the hasTopping property with a filler of JalapenoPepperTopping.
2. A SohoPizza is almost the same as a MargheritaPizza but has additional toppings of olives, parmezan cheese — create this by cloning MargheritaPizza and adding two existential restrictions along the property hasTopping, one with a filler of OliveTopping, another one is ParmezanTopping.

For AmericanHot pizza the class description view should now look like the picture shown in Figure 2.35.

For SohoPizza the class description view should now look like the picture shown in 2.36.

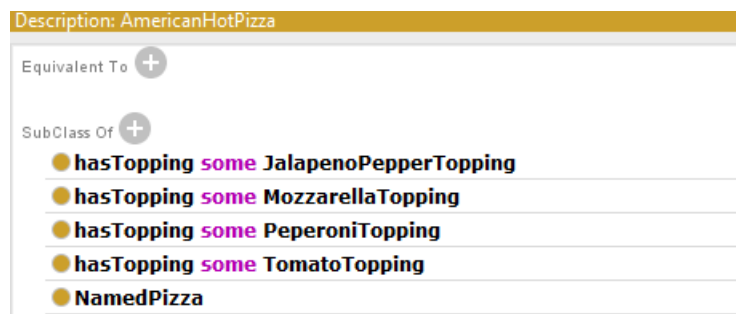


Figure 2.35: The Class Description View Displaying the Description for AmericanHotPizza

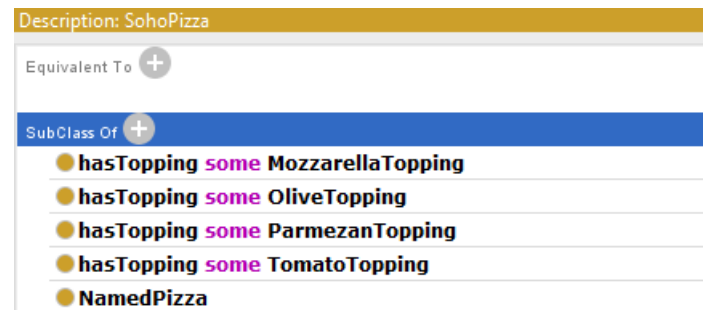


Figure 2.36: The Class Description View Displaying the Description for SohoPizza

Having created these pizzas we now need to make them disjoint from one another:

Exercise 22: Make subclasses of NamedPizza disjoint from each other

1. Select the class MargheritaPizza in the class hierarchy on the “Classes” tab.
2. Select the “Make primitive siblings disjoint” option in the “Edit” menu to make the pizzas disjoint from each other.

2.9 Using a Reasoner

One of the key features of ontologies that are described using OWL-DL is that they can be processed by a reasoner. One of the main services offered by a reasoner is to test whether or not one class is a subclass of another class. By performing such tests on the classes in an ontology it is possible for a reasoner to compute the inferred ontology class hierarchy. Another standard service that is offered by reasoners is consistency checking. Based on the description (conditions) of a class the reasoner can check whether or not it is possible for the class to have any instances. A class is deemed to be inconsistent if it cannot possibly have any instances.

2.9.1 Invoking the Reasoner

Protégé 5 allows different OWL reasoners to be plugged in. The ontology can be “sent to the reasoner” to automatically compute the classification hierarchy, and also to check the logical consistency of the ontology. In Protégé 5 the “manually constructed” class hierarchy is called the *asserted hierarchy*. The class hierarchy that is automatically computed by the reasoner is called the *inferred hierarchy*. You can click “Reasoner” tab and choose “start reasoner” as shown in Figure 2.37

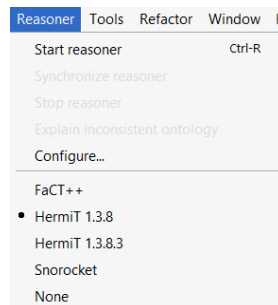


Figure 2.37: Reasoner Menu

2.9.2 Inconsistent Classes

In order to demonstrate the use of the reasoner in detecting inconsistencies in the ontology we will create a class that is a subclass of both CheeseTopping and also VegetableTopping. This strategy is often used as a check so that we can see that we have built our ontology correctly. Classes that are added in order to test the integrity of the ontology are sometimes known as Probe Classes.

Exercise 23: Add a Probe Class called ProbeInconsistentTopping which is a subclass of both CheeseTopping and VegetableTopping

1. Select the class CheeseTopping from the class hierarchy on the Classes tab.
2. Create a subclass of CheeseTopping named ProbeInconsistentTopping.
3. Add a comment to the ProbeInconsistentTopping class that is something along the lines of, “This class should be inconsistent when the ontology is classified.” This will enable anyone who looks at our pizza ontology to see that we deliberately meant the class to be inconsistent.

4. Ensure that the ProbeInconsistentTopping class is selected in the class hierarchy, and then select the “Subclass Of” header in the ‘Class Description View’.
5. Click on the “Subclass Of” button on the “Class Description View”. This will display a dialog containing the class hierarchy from which a class may be selected. Select the class VegetableTopping and then press the “OK” button. The class VegetableTopping will be added as a superclass, so that the class description view should look like the picture in Figure 2.38.

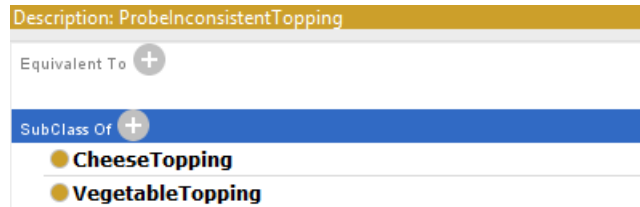


Figure 2.38: The Class Description View Displaying ProbeInconsistentTopping

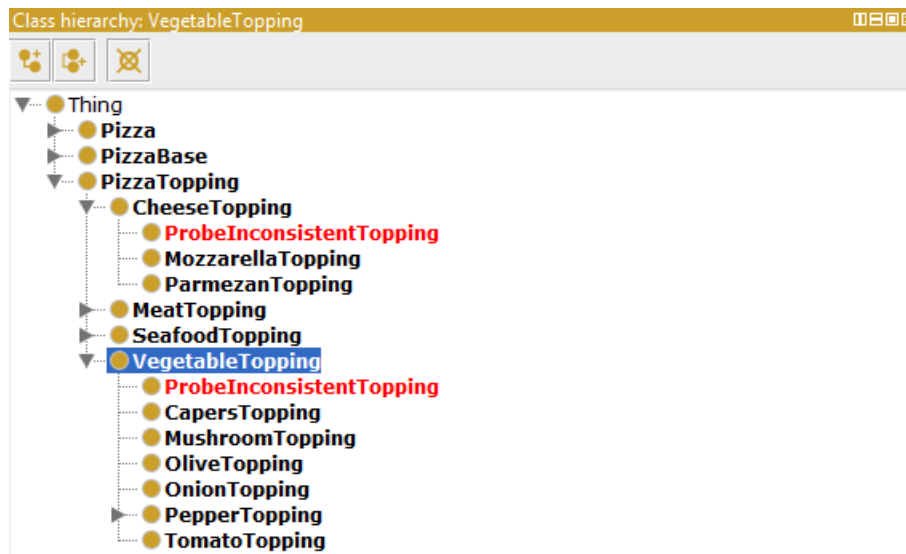


Figure 2.39: The Class ProbeInconsistentTopping Found to be Inconsistent by the Reasoner

After a few seconds the inferred hierarchy will have been computed and the inferred hierarchy window will pop open (You may need to check “Show Inferences” at the bottom right corner; the inferred hierarchy may not pop open but you can see inconsistency warning in the class hierarchy). The hierarchy should resemble that shown in Figure 2.39— notice that the class ProbeInconsistentTopping is highlighted in red, indicating that the reasoner has found this class to be inconsistent.

Exercise 26: Remove the disjoint statement between CheeseTopping and VegetableTopping to see what happens

1. Select the class CheeseTopping using the class hierarchy
2. The “Disjoints view” should contain CheeseTopping’s sibling classes: VegetableTopping, SeafoodTopping and MeatTopping. Select VegetableTopping in the Disjoints view.



3. Press the “Delete selected row” button on the Disjoints view to remove the disjoint axiom that states CheeseTopping and VegetableTopping are disjoint.
4. Press “start Reasoner” on the Reasoner drop down menu to send the ontology to the reasoner.

2.10 Necessary and Sufficient Conditions (Primitive and Defined Classes)

All of the classes that we have created so far have only used necessary conditions to describe them. Necessary conditions can be read as, “If something is a member of this class then it is necessary to fulfil these conditions”. With necessary conditions alone, we cannot say that, “If something fulfils these conditions then it must be a member of this class”.

Let’s illustrate this with an example. We will create a subclass of Pizza called CheesyPizza, which will be a Pizza that has at least one kind of CheeseTopping.

Exercise 27: Create a subclass of Pizza called CheesyPizza and specify that it has at least one topping that is a kind of CheeseTopping

1. Select Pizza in the class hierarchy on the ‘Classes’ tab
2. Press the “Add subclass” icon () to create a subclass of Pizza. Name it CheesyPizza
3. Make sure that CheesyPizza is selected in the class hierarchy. Select the “Add” icon () next to the “SubClass Of” header in the class description view. Select the “Class expression editor” tab.
4. Type hasTopping as the property to be restricted.
5. Type ‘some’ to create the existential restriction
6. Finally type CheeseTopping and press “OK” to close the dialog and create the restriction.

The “Class Description View” should now look like the picture shown in Figure 2.40, Our description of CheesyPizza states that if something is a member of the class CheesyPizza it is necessary for it to be a member of the class Pizza and it is necessary for it to have at least one topping that is a member of the class CheeseTopping.

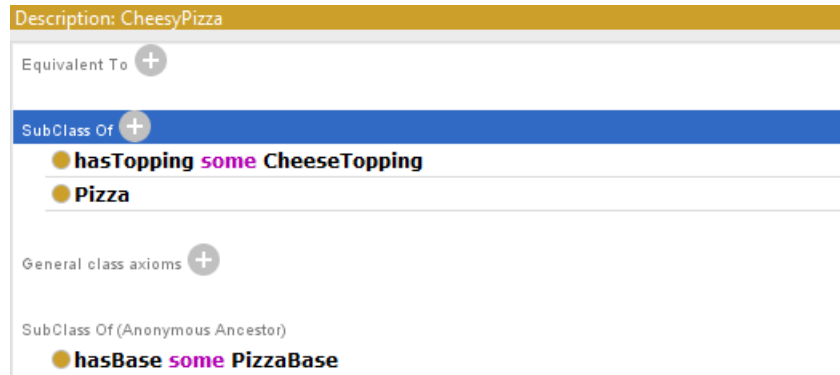


Figure 2.40: The Description of CheesyPizza

Our current description of CheesyPizza says that if something is a CheesyPizza it is necessarily a Pizza and it is necessary for it to have at least one topping that is a kind of CheeseTopping. We have used necessary conditions to say this. Now consider some (random) individual. Suppose that we know that this individual is a member of the class Pizza. We also know that this individual has at least one kind of CheeseTopping. However, given our current description of CheesyPizza this knowledge is not sufficient to determine that the individual is a member of the class CheesyPizza. To make this possible we need to change the conditions for CheesyPizza from necessary conditions to necessary and sufficient conditions. This means that not only are the conditions necessary for membership of the class CheesyPizza, they are also sufficient to determine that any (random) individual that satisfies them must be a member of the class CheesyPizza.

In order to convert necessary conditions to necessary and sufficient conditions, the conditions must be moved from under the “Subclass Of” header in the class description view to be under the “Equivalent classes” header. This can be done with the “Convert to defined class” option in the “Edit” menu.

Exercise 28: Convert the necessary conditions for CheesyPizza into necessary & sufficient conditions

1. Ensure that CheesyPizza is selected in the class hierarchy
2. In the “Edit” menu select “Convert to defined class”.

The ‘Class Description View’ should now look like the picture shown in Figure 2.41

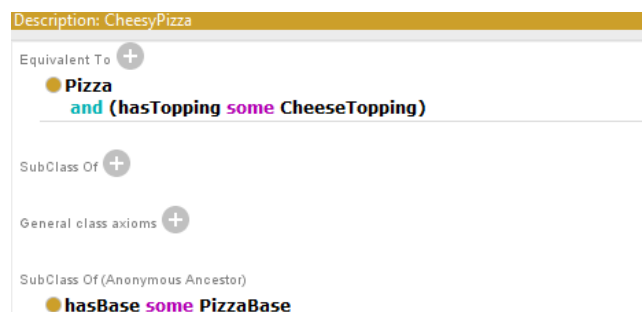


Figure 2.41: The Description of CheesyPizza (Using Necessary AND Sufficient Conditions)

We have converted our description of CheesyPizza into a definition. If something is a CheesyPizza then it is necessary that it is a Pizza and it is also necessary that at least one topping that is a member of the class CheeseTopping. Moreover, if an individual is a member of the class Pizza and it has at least one topping that is a member of the class CheeseTopping then these conditions are sufficient to determine that the individual must be a member of the class CheesyPizza. The notion of necessary and sufficient conditions is illustrated in Figure 2.42.

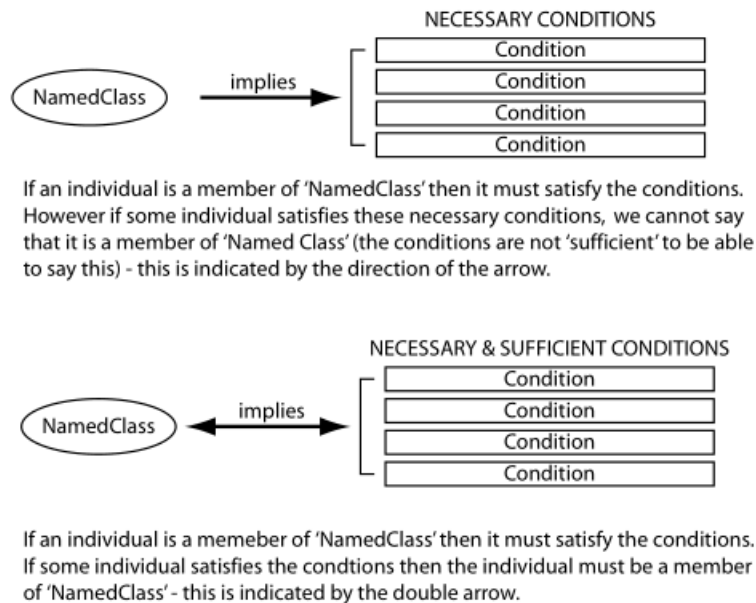


Figure 2.42: Necessary and Sufficient Conditions

To summarize: If class A is described using necessary conditions, then we can say that if an individual is a member of class A it must satisfy the conditions. We cannot say that any (random) individual that satisfies these conditions must be a member of class A. However, if class A is now defined using necessary and sufficient conditions, we can say that if an individual is a member of the class A it must satisfy the conditions and we can now say that if any (random) individual satisfies these conditions then it must be a member of class A. The conditions are not only necessary for membership of A but also sufficient to determine that something satisfying these conditions is a member of A.

How is this useful in practice? Suppose we have another class B, and we know that any individuals that are members of class B also satisfy the conditions that define class A. We can determine that class B is subsumed by class A — in other words, B is a subclass of A. Checking for class subsumption is a key task of a description logic reasoner and we will use the reasoner to automatically compute a classification hierarchy in this way.

2.10.1 Primitive and Defined Classes

Classes that have at least one set of necessary and sufficient conditions are known as defined classes — they have a definition, and any individual that satisfies the definition will belong to the class. Classes that do not have any sets of necessary and sufficient conditions (only have necessary conditions) are known as primitive classes. In Protégé 5 defined classes have a class icon with three horizontal white lines in them. Primitive classes have a class icon that has a plain yellow background. It is also important to understand that the reasoner can only automatically classify classes under defined classes - i.e. classes with at least one set of necessary and sufficient conditions.

2.11 Automated Classification

Being able to use a reasoner to automatically compute the class hierarchy is one of the major benefits of building an ontology. Indeed, when constructing very large ontologies the use of a reasoner to compute subclass-superclass relationships between classes becomes almost vital. Without a reasoner it is very difficult to keep large ontologies in a maintainable and logically correct state. In cases where ontologies can have classes that have many superclasses (multiple inheritance) it is nearly always a good idea to construct the class hierarchy as a simple tree. Classes in the asserted hierarchy (manually constructed hierarchy) therefore have no more than one superclass. Computing and maintaining multiple inheritance is the job of the reasoner. This technique helps to keep the ontology in a maintainable and modular state. Not only does this promote the reuse of the ontology by other ontologies and applications, it also minimizes human errors that are inherent in maintaining a multiple inheritance hierarchy.

Having created a definition of a CheesyPizza we can use the reasoner to automatically compute the subclasses of CheesyPizza.

Exercise 29: Use the reasoner to automatically compute the subclasses of CheesyPizza

1. Press the “start reasoned” button on the Reasoner drop down menu (See Figure 2.37).

After a few seconds the inferred hierarchy should have been computed and the inferred hierarchy window will pop open (if it was previously closed). The inferred hierarchy should appear similar to the picture shown in Figure 2.43. Figures 2.44 and 2.45 show the OWLViz display of the asserted and inferred hierarchies respectively.

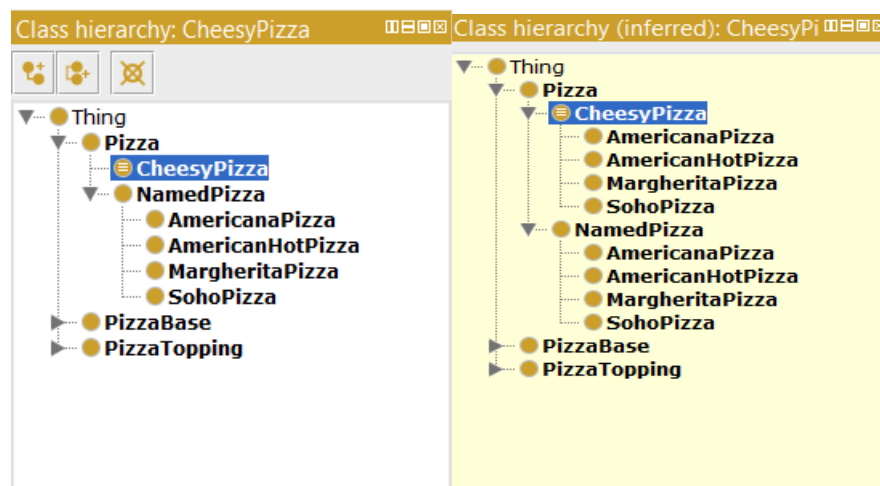


Figure 2.43: The Asserted and Inferred Hierarchies Displaying The Classification Results for CheesyPizza

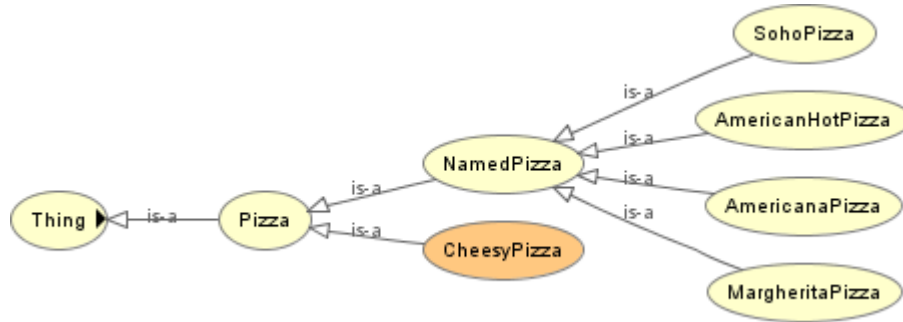


Figure 2.44: OWLViz Displaying the Asserted Hierarchy for CheesyPizza

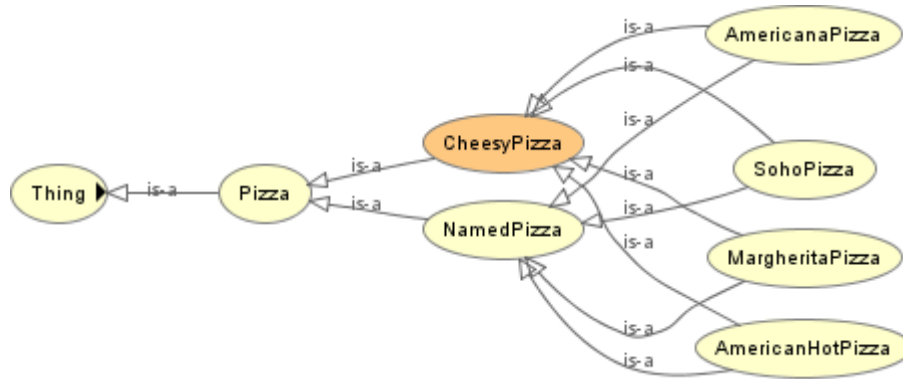


Figure 2.45: OWLViz Displaying the Inferred Hierarchy for CheesyPizza

2.12 Universal Restrictions

All of the restrictions we have created so far have been existential restrictions (some). Existential restrictions specify the existence of at least one relationship along a given property to an individual that is a member of a specific class (specified by the filler). However, existential restrictions do not mandate that the only relationships for the given property that can exist must be to individuals that are members of the specified filler class.


For example, we could use an existential restriction `hasTopping some MozzarellaTopping` to describe the individuals that have at least one relationship along the property `hasTopping` to an individual that is a member of the class `MozzarellaTopping`. This restriction does not imply that all of the `hasTopping` relationships must be to a member of the class `MozzarellaTopping`. To restrict the relationships for a given property to individuals that are members of a specific class we must use a universal restriction.

Universal restrictions are given the symbol \forall . They constrain the relationships along a given property to individuals that are members of a specific class. For example the universal restriction \forall `hasTopping MozzarellaTopping` describes the individuals all of whose `hasTopping` relationships are to members of the

class `MozzarellaTopping` — the individuals do not have a `hasTopping` relationships to individuals that aren't members of the class `MozzarellaTopping`.

Suppose we want to create a class called `VegetarianPizza`. Individuals that are members of this class can only have toppings that are `CheeseTopping` or `VegetableTopping`. To do this we can use a universal restriction.

Exercise 30: Create a class to describe a VegetarianPizza

1. Create a subclass of `Pizza`, and name it `VegetarianPizza`.
2. Making sure that `VegetarianPizza` is selected, click on the “Add” icon () next to the ‘Subclass Of’ header in the “Class Description View”.
3. Type `hasTopping` as the property to be restricted
4. Type “only” in order to create a universally quantified restriction.
5. For the filler we want to say `CheeseTopping` or `VegetableTopping`. We place this inside brackets so write an open bracket followed by the class `CheeseTopping` in the filler box. We now need to use the `unionOf` operator between the class names. We can add this operator by simply typing `or`. Next insert the class `VegetableTopping` by typing it. You should now have **hasTopping only (CheeseTopping or VegetableTopping)** in the text box.
6. Press “OK” to close the dialog and create the restriction — if there are any errors (due to typing errors etc.) they will be underlined in red.

At this point the class description view should look like the picture shown in Figure 2.46.

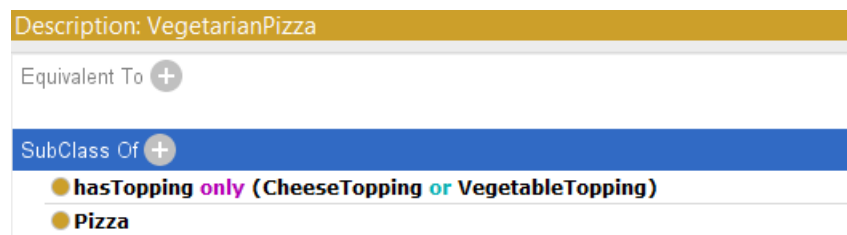


Figure 2.46: The Description of `VegetarianPizza` (Using Necessary Conditions)

Currently `VegetarianPizza` is described using necessary conditions. However, our description of a `VegetarianPizza` could be considered to be complete. We know that any individual that satisfies these conditions must be a `VegetarianPizza`. We can therefore convert the necessary conditions for `VegetarianPizza` into necessary and sufficient conditions. This will also enable us to use the reasoner to determine the subclasses of `VegetarianPizza`.

Exercise 31: Convert the necessary conditions for `VegetarianPizza` into necessary & sufficient conditions

1. Ensure that `VegetarianPizza` is selected in the class hierarchy.
2. In the ‘Edit’ menu select “Convert to defined class”.

The “Class Description View” should now look like the picture shown in Figure 2.47.

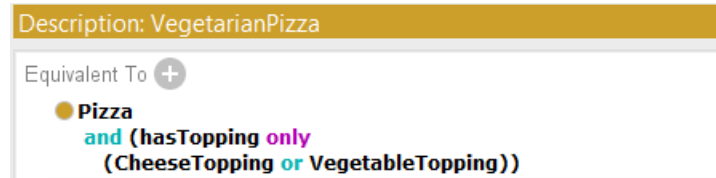


Figure 2.47: The Class Description View Displaying the Definition of VegetarianPizza (Using Necessary and Sufficient Conditions)

2.13 Automated Classification and Open World Reasoning

We want to use the reasoner to automatically compute the superclass-subclass relationship (subsumption relationship) between MargheritaPizza and VegetarianPizza and also, SohoPizza and VegetarianPizza. Recall that we believe that MargheritaPizza and SohoPizza should be vegetarian pizzas (they should be subclasses of VegetarianPizza). This is because they have toppings that are essentially vegetarian toppings — by our definition, vegetarian toppings are members of the classes CheeseTopping or VegetableTopping and their subclasses. Having previously created a definition for VegetarianPizza (using a set of necessary and sufficient conditions) we can use the reasoner to perform automated classification and determine the vegetarian pizzas in our ontology.

You will notice that MargheritaPizza and also SohoPizza have not been classified as subclasses of VegetarianPizza. This may seem a little strange, as it appears that both MargheritaPizza and SohoPizza have ingredients that are vegetarian ingredients, i.e. ingredients that are kinds of CheeseTopping or kinds of VegetableTopping. However, as we will see, MargheritaPizza and SohoPizza have something missing from their definition that means they cannot be classified as subclasses of VegetarianPizza.


Reasoning in OWL is based on what is known as the open world assumption (OWA). It is frequently referred to as open world reasoning (OWR). The open world assumption means that we cannot assume something doesn't exist until it is explicitly stated that it does not exist. In other words, because something hasn't been stated to be true, it cannot be assumed to be false — it is assumed that “the knowledge just hasn't been added to the knowledge base”. In the case of our pizza ontology, we have stated that MargheritaPizza has toppings that are kinds of MozzarellaTopping and also kinds of TomatoTopping. Because of the open world assumption, until we explicitly say that a MargheritaPizza only has these kinds of toppings, it is assumed (by the reasoner) that a MargheritaPizza could have other toppings. To specify explicitly that a MargheritaPizza has toppings that are kinds of MozzarellaTopping or kinds of MargheritaTopping and only kinds of MozzarellaTopping or MargheritaTopping, we must add what is known as a closure axiom on the hasTopping property.

4.13.1 Closure Axioms

A closure axiom on a property consists of a universal restriction that acts along the property to say that it can only be filled by the specified fillers. The restriction has a filler that is the union of the fillers that occur in the existential restrictions for the property. For example, the closure axiom on the hasTopping property for MargheritaPizza is a universal restriction that acts along the hasTopping property, with a filler that is

the union of MozzarellaTopping and also TomatoTopping. i.e. $\forall \text{ hasTopping } (\text{MozzarellaTopping} \cup \text{TomatoTopping})$.

Exercise 32: Add a closure axiom on the hasTopping property for MargheritaPizza

1. Make sure that MargheritaPizza is selected in the class hierarchy on the “Classes” tab.
2. Press the “Add” icon () next to the “Subclass Of” section of the “Class Description” view to open the edit text box.
3. Type hasTopping as the property to be restricted.
4. Type “only” to create the universal restriction.
5. Open brackets and type MozzarellaTopping or TomatoTopping close bracket.
6. Press “OK” to create the restriction and add it to the class MargheritaPizza.

The class description view should now appear as shown in Figure 2.48.

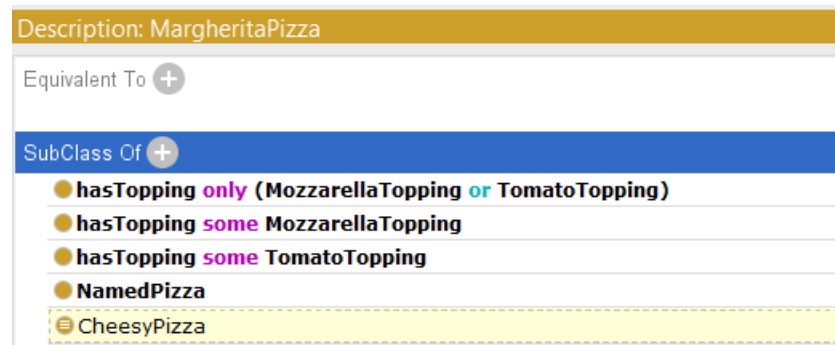



Figure 2.48: Class Description View: Margherita Pizza with a Closure Axiom for the hasTopping Property

Exercise 34: Add a closure axiom on the hasTopping property for SohoPizza

1. Make sure that SohoPizza is selected in the class hierarchy on the “Classes” tab.
2. Press the “Add” icon () on the “Subclass Of” section of the “Class Description” view to open the edit text box.
3. Type hasTopping as the property to be restricted.
4. Type “only” to create the universal restriction.
5. Open brackets and type MozzarellaTopping or TomatoTopping or ParmezanTopping or OliveTopping close bracket.
6. Press “OK” to create the restriction and add it to the class SohoPizza.

For completeness, we will add closure axioms for the hasTopping property to AmericanaPizza and also AmericanHotPizza. At this point it may seem like tedious work to enter these closure axioms by hand.

Exercise 35: Automatically create a closure axiom on the hasTopping property for AmericanaPizza

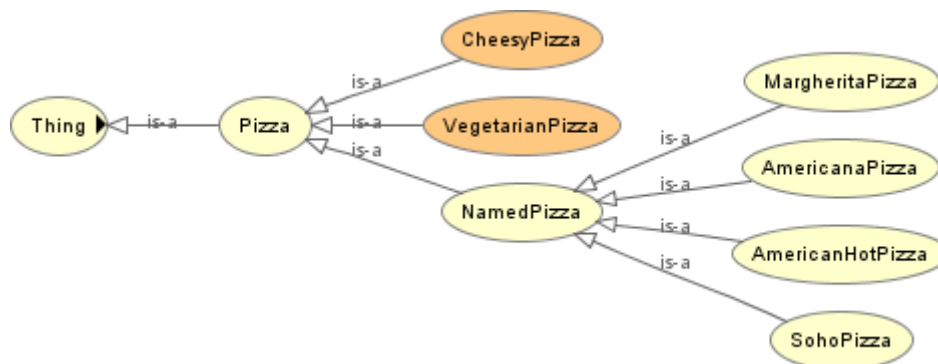
1. Select AmericanaPizza in the class hierarchy on the Classes tab.
2. In the class description view, select one of the restrictions on the hasTopping property, e.g., hasTopping some TomatoTopping.
3. Right click the restriction and select “Create closure axiom”.
4. A new closure axiom is created.

Exercise 36: Automatically create a closure axiom on the hasTopping property for AmericanHotPizza

1. Select AmericanHotPizza in the class hierarchy on the Classes tab.
2. In the class description view, select one of the restrictions on the hasTopping property, e.g., hasTopping some TomatoTopping.
3. Right click the restriction and select “Create closure axiom”.
4. A new closure axiom is created.

Having added closure axioms on the hasTopping property for our pizzas, we can now use the reasoner to automatically compute classifications for them.

After a short amount of time the ontology will have been classified and the “Inferred Hierarchy” pane will pop open (if it is not already open). This time, MargheritaPizza and also SohoPizza will have been classified as subclasses of VegetarianPizza. This has happened because we specifically “closed” the hasTopping property on our pizzas to say exactly what toppings they have and VegetarianPizza was defined to be a Pizza with only kinds of CheeseTopping and only kinds of VegetableTopping. Figure 2.49 shows the current asserted and inferred hierarchies. It is clear to see that the asserted hierarchy is simpler and “cleaner” than the “tangled” inferred hierarchy. Although the ontology is only very simple at this stage, it should be becoming clear that the use of a reasoner can help (especially in the case of large ontologies) to maintain a multiple inheritance hierarchy for us.



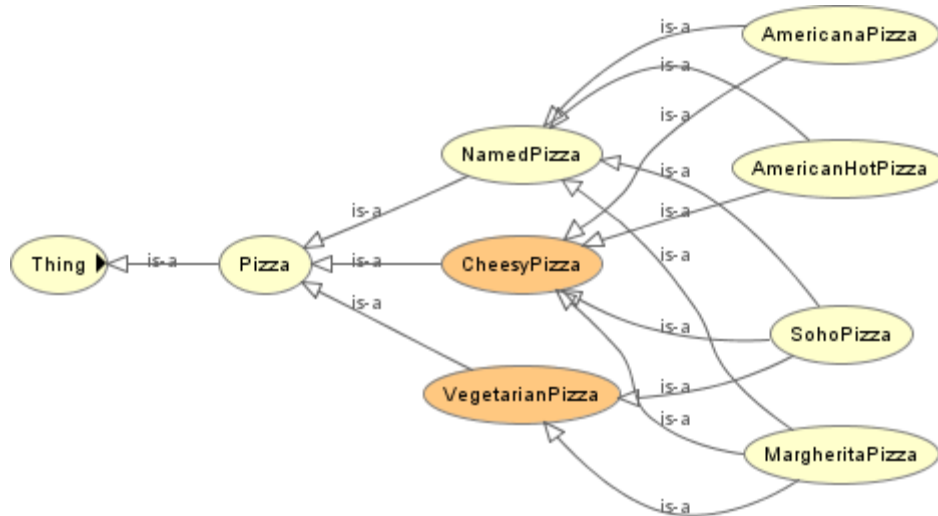


Figure 2.49: The Asserted and Inferred Hierarchies Showing the “before and after” Classification of Pizzas into CheesyPizzas and VegetarianPizzas.


2.14 Value Partitions

In this section we create some Value Partitions, which we will use to refine our descriptions of various classes. Value Partitions are not part of OWL, or any other ontology language, they are a “design pattern”. Design patterns in ontology design are analogous to design patterns in object oriented programming — they are solutions to modelling problems that have occurred over and over again. These design patterns have been developed by experts and are now recognized as proven solutions for solving common modelling problems. As mentioned previously, Value Partitions can be created to refine our class descriptions, for example, we will create a Value Partition called “SpicinessValuePartition” to describe the “spiciness” of PizzaToppings. Value Partitions restrict the range of possible values to an exhaustive list, for example, our “SpicinessValuePartition” will restrict the range to “Mild”, “Medium”, and “Hot”. Creating a ValuePartition in OWL consists of several steps:

1. Create a class to represent the ValuePartition. For example to represent a “spiciness” ValuePartition we might create the class SpicinessValuePartition.
2. Create subclasses of the ValuePartition to represent the possible options for the ValuePartition. For example we might create the classes Mild, Medium and Hot as subclasses of the SpicinessValuePartition class.
3. Make the subclasses of the ValuePartition class disjoint.
4. Provide a covering axiom to make the list of value types exhaustive.
5. Create an object property for the ValuePartition. For example, for our spiciness ValuePartition, we might create the property hasSpiciness.
6. Make the property functional.
7. Set the range of the property as the ValuePartition class. For example for the hasSpiciness property the range would be set to SpicinessValuePartition.

Let's create a ValuePartition that can be used to describe the spiciness of our pizza toppings. We will then be able to classify our pizzas into spicy pizzas and non-spicy pizzas. We want to be able to say that our pizza toppings have a spiciness of either “mild”, “medium” or “hot”. Note that these choices are mutually exclusive – something cannot be both “mild” and “hot”, or a combination of the choices.

Exercise 37: Create a ValuePartition to represent the spiciness of pizza toppings

1. Create a new class as a sub class of Thing called ValuePartition
2. Create a sub class of ValuePartition called SpicinessValuePartition.
3. Create three new classes as subclasses of SpicinessValuePartition. Name these classes Hot, Medium, and Mild.
4. Make the classes Hot, Medium, and Mild disjoint from each other. You can do this by selecting the class Hot, and selecting “Make all primitive siblings disjoint” from the “Edit” menu.
5. In the “Object Property Tab” create a new Object Property called hasSpiciness. Set the range of this property to SpicinessValuePartition. Make this new property functional by ticking the functional box.
6. Add a covering axiom to the SpicinessValuePartition. Highlight SpicinessValuePartition in the class hierarchy, in the “Equivalent classes” section of the class description view select the “Add” icon () and type Hot or Medium or Mild in the dialog box.

Let's look at the SpicinessValuePartition class (refer to Figure 2.50 and Figure 2.51):

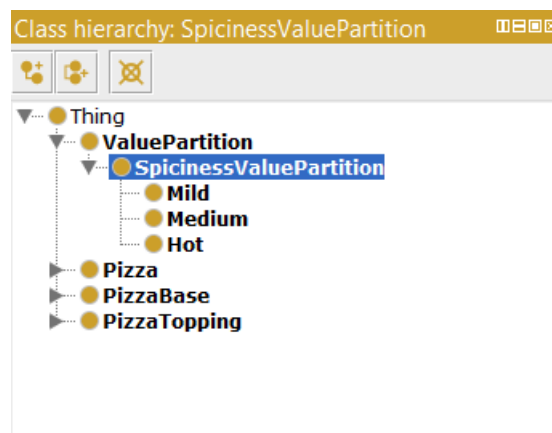


Figure 2.50: Classes Added by the “Create ValuePartition” Wizard

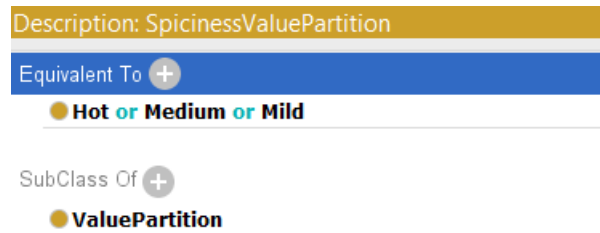


Figure 2.51: The Class Description View Displaying the Description of the SpicinessValuePartition Class

2.14.1 Covering Axioms

As part of the ValuePartition pattern we use a *covering axiom*. A covering axiom consists of two parts: the class that is being “covered”, and the classes that form the covering. For example, suppose we have three classes A, B and C. Classes B and C are subclasses of class A. Now suppose that we have a covering axiom that specifies class A is covered by class B and also class C. This means that a member of class A must be a member of B and/or C. If classes B and C are disjoint then a member of class A must be a member of either class B or class C. Remember that ordinarily, although B and C are subclasses of A, an individual may be a member of A without being a member of either B or C.

In Protégé 5 a covering axiom manifests itself as a class that is the union of the classes being covered, which forms a superclass of the class that is being covered. In the case of classes A, B and C, class A would have a superclass of $B \cup C$. The effect of a covering axiom is depicted in Figure 2.52.

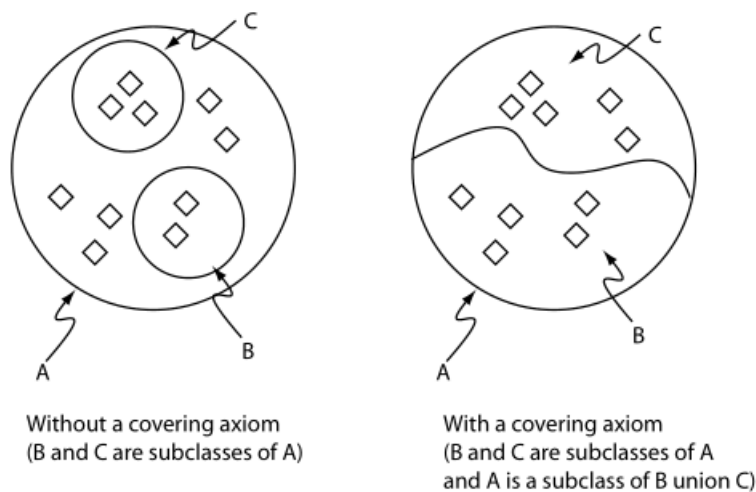


Figure 2.52: A Schematic Diagram that Shows the Effect of using a Covering Axiom to Cover Class A with Classes B and C

Our SpicinessValuePartition has a covering axiom to state that SpicinessValuePartition is covered by the classes Mild, Medium and Hot — Mild, Medium and Hot are disjoint from each other so that an individual cannot be a member of more than one of them. The class SpicinessValuePartition has a superclass that is

$\text{Mild} \cup \text{Medium} \cup \text{Hot}$. This covering axiom means that a member of `SpicinessValuePartition` must be a member of either `Mild` or `Medium` or `Hot`.

The difference between not using a covering axiom, and using a covering axiom is depicted in Figure 2.53. In both cases the classes `Mild`, `Medium` and `Hot` are disjoint — they do not overlap. It can be seen that in the case without a covering axiom an individual may be a member of the class `SpicinessValuePartition` and still not be a member of `Mild`, `Medium` or `Hot` — `SpicinessValuePartition` is not covered by `Mild`, `Medium` and `Hot`. Contrast this with the case when a covering axiom is used. It can be seen that if an individual is a member of the class `SpicinessValuePartition`, it must be a member of one of the three subclasses `Mild`, `Medium` or `Hot` — `SpicinessValuePartition` is covered by `Mild`, `Medium` and `Hot`.

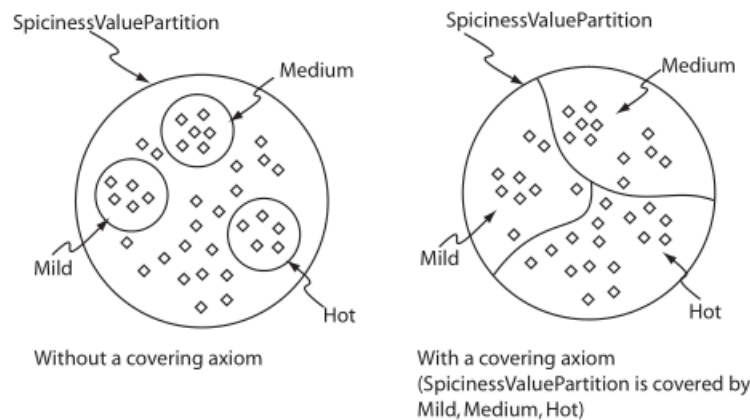



Figure 2.53: The Effect of Using a Covering Axiom on the `SpicinessValuePartition`

2.15 Adding Spiciness to Pizza Toppings

We can now use the `SpicinessValuePartition` to describe the spiciness of our pizza toppings. To do this we will add an existential restriction to each kind of `PizzaTopping` to state its spiciness. Restrictions will take the form, *hasSpiciness some SpicinessValuePartition*, where `SpicinessValuePartition` will be one of `Mild`, `Medium` or `Hot`.

Exercise 38: Specify `hasSpiciness` restrictions on `PizzaToppings`

1. Make sure that `JalapenoPepperTopping` is selected in the class hierarchy.
2. Use the “Add” icon () on the “Subclass Of” section of the “Class Description view” which opens a dialog.
3. Select the “Class expression editor” tab.
4. Type **hasSpiciness some Hot** to create the existential restriction. Remember you can use autocomplete to speed up the process.
5. Press “OK” to create the restriction — if there are any errors, the restriction will not be created, and the error will be highlighted in red.
6. Repeat this for each of the bottom level `PizzaToppings` (those that have no subclasses)

To complete this section, we will create a new class SpicyPizza, which should have pizzas that have spicy toppings as its subclasses. In order to do this we want to define the class SpicyPizza to be a Pizza that has at least one topping (hasTopping) that has a spiciness (hasSpiciness) that is Hot. This can be accomplished in more than one way, but we will create a restriction on the hasTopping property, that has a restriction on the hasSpiciness property as its filler.

Exercise 39: Create a SpicyPizza as a subclass of Pizza

1. Create a subclass of Pizza called SpicyPizza.
2. Ensure SpicyPizza is selected.
3. Press the “Add” icon (+) on the “Subclass Of” section of the class description view
4. Type hasTopping as the property to be restricted.
5. Type “some” as the type of restriction
6. The filler should be: PizzaTopping and hasSpiciness some Hot. This filler describes an anonymous class, which contains the individuals that are members of the class PizzaTopping and also members of the class of individuals that are related to the members of class Hot via the hasSpiciness property. In other words, the things that are PizzaToppings and have a spiciness that is Hot. To create this restriction in the text box type, (PizzaTopping and (hasSpiciness some Hot)), including the brackets.
7. Finally, select the “Convert to defined class” option in the “Edit” menu.

The class description view should now look like the picture shown in Figure 2.54

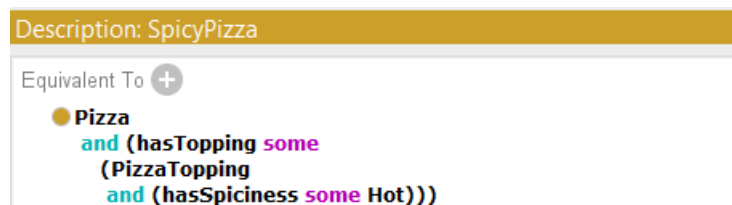


Figure 2.54: The Definition of SpicyPizza

We should now be able to invoke the reasoner and determine the spicy pizzas in our ontology. After the reasoner has finished, the “Inferred Hierarchy” class pane will pop open, and you should find that AmericanHotPizza has been classified as a subclass of SpicyPizza — the reasoner has automatically computed that any individual that is a member of AmericanHotPizza is also a member of SpicyPizza.

2.16 Cardinality Restrictions

In OWL we can describe the class of individuals that have at least, at most or exactly a specified number of relationships with other individuals or datatype values. The restrictions that describe these classes are known as Cardinality Restrictions. For a given property P, a Minimum Cardinality Restriction specifies the minimum number of P relationships that an individual must participate in. A Maximum Cardinality Restriction specifies the maximum number of P relationships that an individual can participate in. A Cardinality Restriction specifies the exact number of P relationships that an individual must participate in.

Relationships (for example between two individuals) are only counted as separate relationships if it can be determined that the individuals that are the fillers for the relationships are different to each other. For example, Figure 2.55 depicts the individual Matthew related to the individuals Nick and the individual Hai via the worksWith property. The individual Matthew satisfies a *minimum cardinality* restriction of 2 along the worksWith property if the individuals Nick and Hai are distinct individuals i.e. they are different individuals.

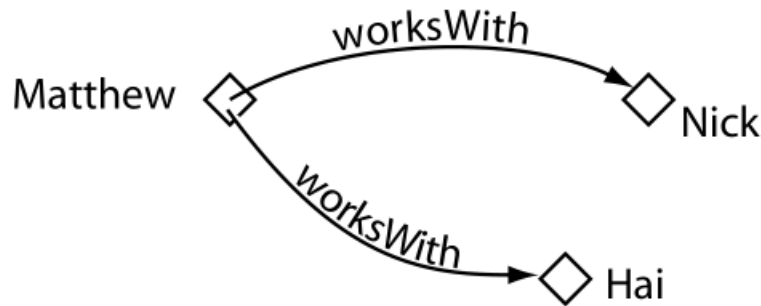



Figure 2.55: Cardinality Restrictions: Counting Relationships

Let's add a cardinality restriction to our Pizza Ontology. We will create a new subclass of Pizza called InterestingPizza, which will be defined to have three or more toppings.

Exercise 40: Create an InterestingPizza that has at least three toppings

1. Switch to the Classes tab and make sure that the Pizza class is selected.
2. Create a subclass of Pizza called InterestingPizza
3. Press the “Add” icon () on the “Subclass Of” section of the class description view.
4. Type hasTopping as a property to be restricted.
5. Type “min” to create a minimum cardinality restriction.
6. Specify a minimum cardinality of three by typing 3 into the text box.
7. Press the “Enter” to close the dialog and create the restriction. The class description view should now show Pizza and hasTopping min 3 under “Subclass Of”. The “Equivalent classes” section should be empty.
8. Select the “Convert to defined class” option in the “Edit” menu. The “Superclasses” section should be empty now while the “Equivalent classes” section should show **Pizza and hasTopping min 3**.

The class description view should now appear like the picture shown in Figure 2.56.

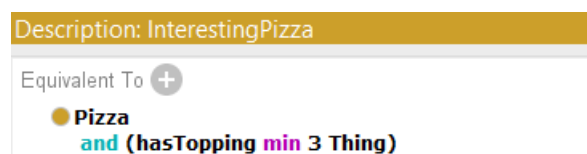



Figure 2.56: The Class Description View Displaying the Description of an InterestingPizza

After the reasoner has classified the ontology, the “Inferred Hierarchy” window will pop open. Expand the hierarchy so that InterestingPizza is visible. Notice that InterestingPizza now has subclasses AmericanaPizza, AmericanHotPizza and SohoPizza — notice MargheritaPizza has not been classified under InterestingPizza because it only has two distinct kinds of topping.

2.17 Qualified Cardinality Restrictions

In the previous section we described cardinality restrictions - specifies the exact number of P relationships that an individual must participate in. In this section we focus on Qualified Cardinality Restrictions (QCR), which are more specific than cardinality restrictions in that they state the class of objects within the restriction. Let’s add a Qualified Cardinality Restriction to our pizza ontology. To do this, we will create a subclass of NamedPizza, called Four Cheese Pizza, which will be defined as having exactly four cheese toppings.

Exercise 41: Create a Four Cheese Pizza that has exactly four cheese toppings

1. Create a subclass of Pizza called FourCheesePizza
2. Select the “Subclass Of” header in the class description view
3. Press the ‘Add’ icon () to open a text box
4. Type hasTopping for the property
5. Type exactly to create an exact cardinality restriction
6. Specify a QCR of four by typing 4 into the text box
7. Type CheeseTopping to specify the type of topping
8. Click OK and create the restriction

Our definition of a FourCheesePizza describes the set of individuals that are members of the class NamedPizza and that have exactly four hasTopping relationships with individuals of the CheeseTopping class. With this description a FourCheesePizza can still also have other relationships to other kinds of toppings. In order for us to say that we just want it to have four cheese toppings and no other toppings we must add the keyword “only” (the universal quantifier). This means that the only kinds of topping allowed are cheese toppings.

Chapter 3

Datatype Properties

In Section 2.4 of Chapter 2 we introduced properties in OWL, but only described object properties — that is, relationships between individuals. In this chapter we will discuss and show examples of Datatype properties. Datatype properties link an individual to an XML Schema Datatype value or an RDF literal. In other words, they describe relationships between an individual and data values. Most of the property characteristics described in Chapter 4 cannot be used with datatype properties. We will describe those characteristics of properties that are applicable to data properties later in this chapter.

Datatype properties can be created using the “Datatype Properties view” in either the “Entities” or “Datatype Properties” tab shown in Figure 3.1. If you cannot see the “Data Properties” tab, you can enable it by checking “Window|Tabs|Data Properties”.

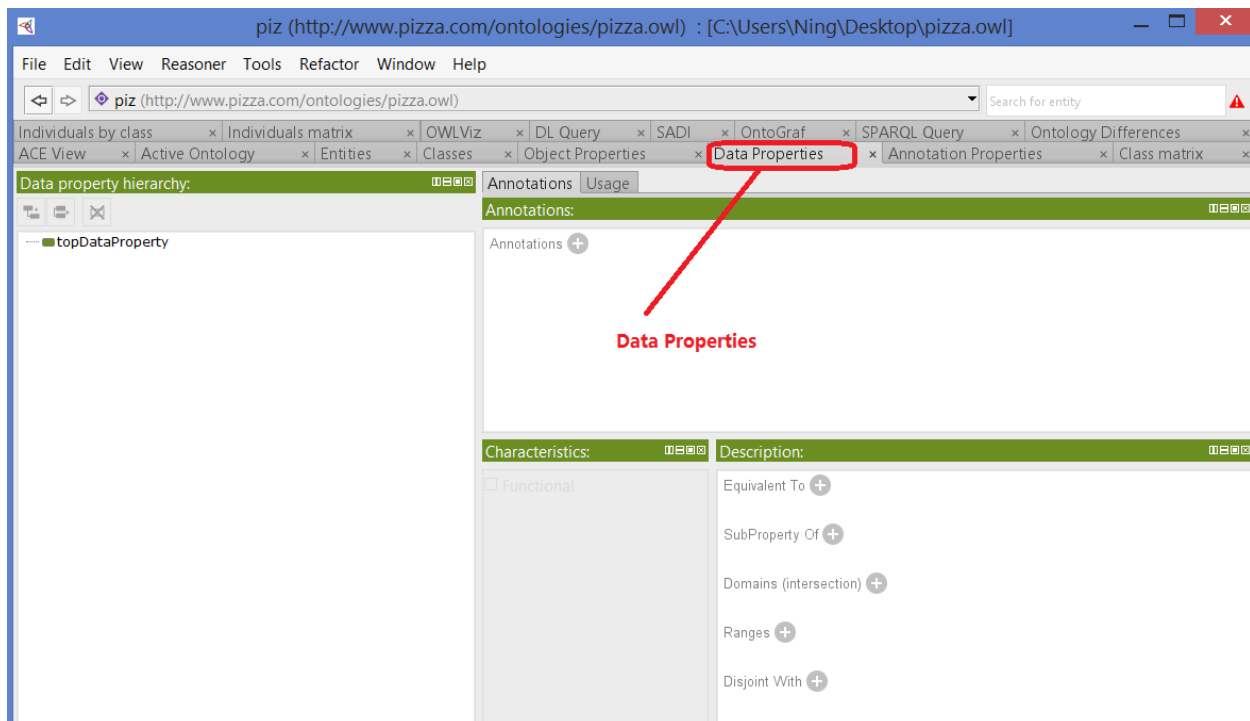



Figure 3.1: A Snapshot of the Datatype Properties Tab in Protégé 5

We will use datatype properties to describe the calorie content of pizzas. We will then use some numeric ranges to broadly classify particular pizzas as high or low calorie. In order to do this we need to complete the following steps:

- Create a datatype property `hasCalorificContentValue`, which will be used to state the calorie content of particular pizzas.
- Create several example pizza individuals with specific calorie contents.
- Create two classes broadly categorizing pizzas as low or high calorie


A datatype property can be used to relate an individual to a concrete data value that may be typed or untyped.

Exercise 42: Create a datatype property called `hasCalorificContentValue`

1. Switch to the “Datatype Properties” tab. Select “`topDataProperty`” which is the root data property for all data properties. Use the “Add sub property” button () to create a new Datatype property called `hasCalorificContentValue`.

There is nothing intrinsic to data properties to prevent us from performing class-level classification, but we will create some individuals as examples for classification as it is probably too strong to say that all `MargheritaPizzas` have exactly 263 calories.

Exercise 43: Create example pizza individuals

1. Ensure the “Entities” Tab is selected and that the “Individuals by type” sub tab is visible (by default in the entities tab it will be one of the views stacked at the bottom left of the tab).
2. Press the ‘Add individual’ button () and create an individual called `Example-Margherita`
3. In the “Individual Description view” add a type of `MargheritaPizza`. In the dialog that appears you can do this with either the “Class hierarchy” or the “Class expression editor”.
4. In the ‘Data property assertions view’ add a “Data property assertion” and in the dialog shown in Figure 3.2 ensure `hasCalorificContentValue` is selected as the property, integer is selected as the type and a value of 263 is entered.
5. Create several more example pizza individuals with different calorie contents including an instance of `QuattroFormaggio` with 723 calories.

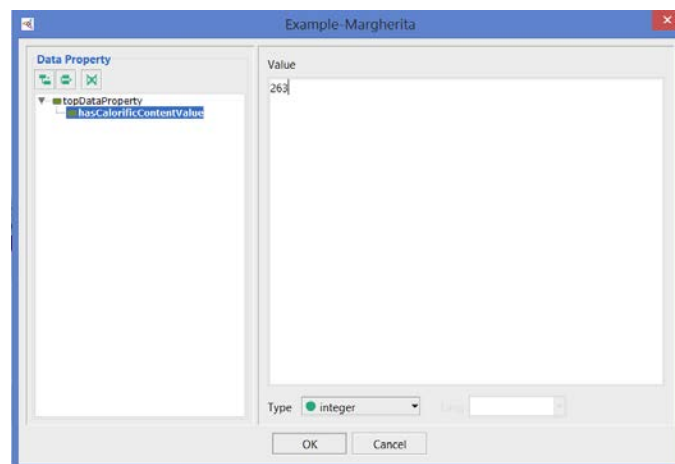


Figure 3.2: Creating a Data Property Assertion

A datatype property can also be used in a restriction to relate individuals to members of a given datatype. Built in datatypes are specified in the XML schema vocabulary and include integers, floats, strings, booleans etc.

Exercise 44: Create a datatype restriction to state that all Pizzas have a calorific value

1. Ensure the “Entities” or “Classes” Tab is selected.
2. Select Pizza and in the “Class hierarchy view”, and click the Add icon (+) next to “Subclass Of” to add a superclass. This brings up the editor dialog.
3. In the dialog select the “Data restriction creator” shown in Figure 3.3. This operates in the same way as the object restriction creator, but the filler is a named datatype.
4. Make sure the type of restriction is set to “some”.
5. Select hasCalorificContentValue as the property being restricted.
6. Finally, choose the datatype integer.
7. Press “OK”. The restriction **hasCalorificContentValue some integer** is now shown in the superclasses.

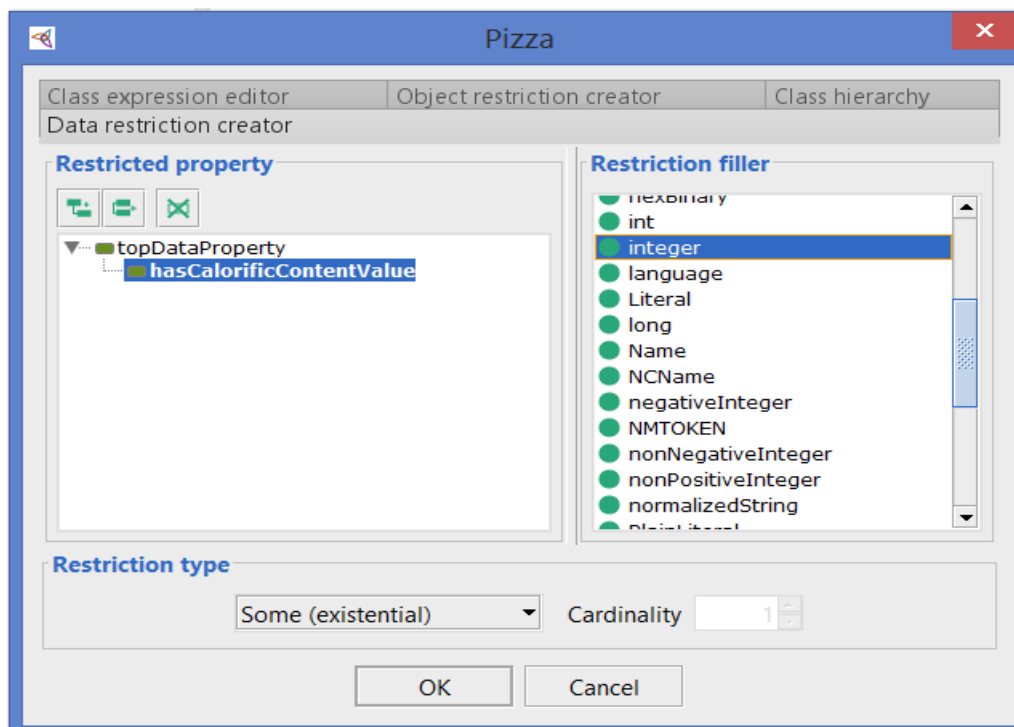



Figure 3.3: Creating a Some Restriction Using a Data Property

In addition to using the predefined set of datatypes we can further specialize the use of a datatype by specifying restrictions on the possible values. For example, it is easy to specify a range of values for a number. Using the datatype property we have created, we will now create defined classes that specify a range of interesting values. We will create definitions that use the `minInclusive` and `maxExclusive` facets that can be applied to numeric datatypes. `HighCaloriePizza` will be defined to be any pizza that has a calorific value equal to or higher than 400.

Exercise 45: Create a HighCaloriePizza that has a calorific value higher than or equal to 400

1. Ensure the “Classes Tab” or “Entities Tab” is selected.
2. Create a subclass of Pizza called HighCaloriePizza.
3. In the “Class Description view” click the “Add” icon () in the “Subclass Of” section to add a new restriction
4. In the “Class expression editor”, type “hasCalorificContentValue some integer[>= 400]” and click “OK”
5. Convert the class to a defined class (“Ctrl-D”, or “Command-D” on a Mac). You should now have a class defined as in Figure 3.4
6. Create a LowCaloriePizza in the same way, but define it as being equivalent to Pizza and **hasCalorificContentValue some integer[< 400]** (any pizza that has a calorific value less than 400). Notice that the definition does not overlap with HighCaloriePizza.

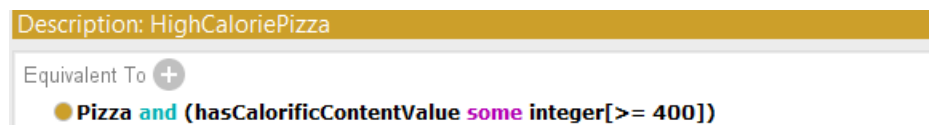


Figure 3.4: Using Datatype Restrictions to Define Ranges for HighCaloriePizza

You now have two categories of pizza that should cover any individual that has had its calorie content specified. We should now be able to test if the classification holds for the example individuals we created.


We will use the reasoner to perform instance classification.



Exercise 46: Inference pizza individuals based on their hasCalorificContentValue


1. Select a reasoner from the “Reasoner” menu or press “Start Reasoner” if one is already selected. The reasoner should classify and show the inferred class hierarchy.
2. Select HighCaloriePizza. You should be able to see inferences shown in the “Class Description” view in yellow with a dashed border.
3. Check the “Instance” section (see Figure 3.5). It should include your instance of “QuattroFormaggio” and perhaps other individuals which you specified as having a calorie value equal to or over 400.
4. Select LowCaloriePizza. Check the “Instance” section. It should include your instance of “ExampleMargherita” and perhaps other individuals which you specified as having a calorie value lower than 400.


Finally, think about how many different calorie values can be held by an individual pizza. Of course, the answer is only one. Remember that there is a property characteristic that states that a property with that characteristic can only be held by an individual once. By describing an object property as functional we state that any given individual can be related to at most one other individual along that property. We can also use the functional characteristic on data properties (this is currently the only characteristic it is possible to use on data properties). By making hasCalorificContentValue functional, we are saying that any one pizza can only ever have one calorie value.


Description: HighCaloriePizza

Equivalent To 





-  **Pizza** and (hasCalorificContentValue  integer[>= 400])


SubClass Of 



-  **Pizza**


General class axioms 


SubClass Of (Anonymous Ancestor)

-  **hasCalorificContentValue**  integer
-  **hasBase**  **PizzaBase**

Instances 

-  **Example-Ning**
-  **Example-QuattroFormaggio**

Target for Key 

Disjoint With 



-  **PizzaBase**
-  **PizzaTopping**

Figure 3.5: Individuals Classified as Being Members of HighCaloriePizza

Exercise 47: Making the hasCalorificContentValue datatype property functional

1. Go to the “Data Properties” tab and select hasCalorificContentValue
2. In the data type “Characteristics” pane, click the “functional” radio button.
3. Test that this works by creating a pizza individual that has two calorie values. This should cause the ontology to become inconsistent.



Chapter 4

More on Open World Reasoning

The examples in this chapter demonstrate the nuances of Open World Reasoning.

We will create a **NonVegetarianPizza** to complement our categorization of pizzas into **VegetarianPizzas**. The **NonVegetarianPizza** should contain all of the Pizzas that are *not* **VegetarianPizzas**. To do this we will create a class that is the complement of **VegetarianPizza**. A complement class contains all of the individuals that are not contained in the class that it is the complement to. Therefore, if we create **NonVegetarianPizza** as a subclass of **Pizza** and make it the complement of **VegetarianPizza** it should contain all of the **Pizzas** that are not members of **VegetarianPizza**.

Exercise 48: Create NonVegetarianPizza as a subclass of Pizza and make it disjoint to VegetarianPizza

1. Select **Pizza** in the class hierarchy on the “Classes” tab. Press the “Add subclass” icon () to create a new class as the subclass of **Pizza**.
2. Name the new class NonVegetarianPizza.
3. Make NonVegetarianPizza disjoint with VegetarianPizza — while NonVegetarianPizza is selected, go to the class “Description” view and press the ‘Add’ icon () on the “Disjoint With” section.

We now want to define a **NonVegetarianPizza** to be a Pizza that is not a **VegetarianPizza**.

Exercise 49: Make VegetarianPizza the complement of VegetarianPizza

1. Make sure that NonVegetarianPizza is selected in the class hierarchy on the ‘Classes tab’.
2. Double click on Pizza in the “Subclass Of” section of the class “Description” view, and edit in “Class expression editor” to create *Pizza and not VegetarianPizza*.
3. Press OK to create and assign the expression. If everything was entered correctly then the class expression editor will close and the expression will have been created. (If there are errors, check the spelling of VegetarianPizza).

The class description view should now resemble the picture shown in 4.2. However, we need to add Pizza to the necessary and sufficient conditions as at the moment our definition of NonVegetarianPizza says that an individual that is not a member of the class VegetarianPizza (everything else!) is a NonVegetarianPizza.

Exercise 50: Add Pizza to the necessary and sufficient conditions for NonVegetarianPizza

1. Make sure NonVegetarianPizza is selected in the class hierarchy on the “Clases” tab.

2. Convert the superclass to an equivalent class. Either select “Edit — Convert to defined class” or copy and paste the superclass you just edited from the superclasses to the equivalent classes section. Note that before pasting you should select the “Equivalent classes” header.

The “Class Description View” should now look like the picture shown in Figure 4.3.

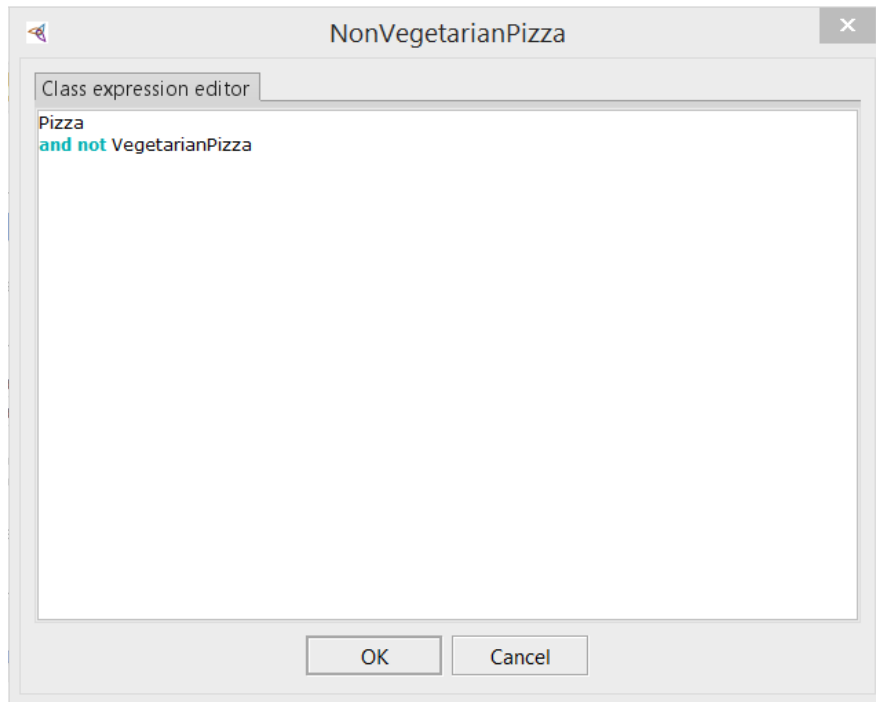


Figure 4.1: Class Description View: Expression Editor Auto Completion

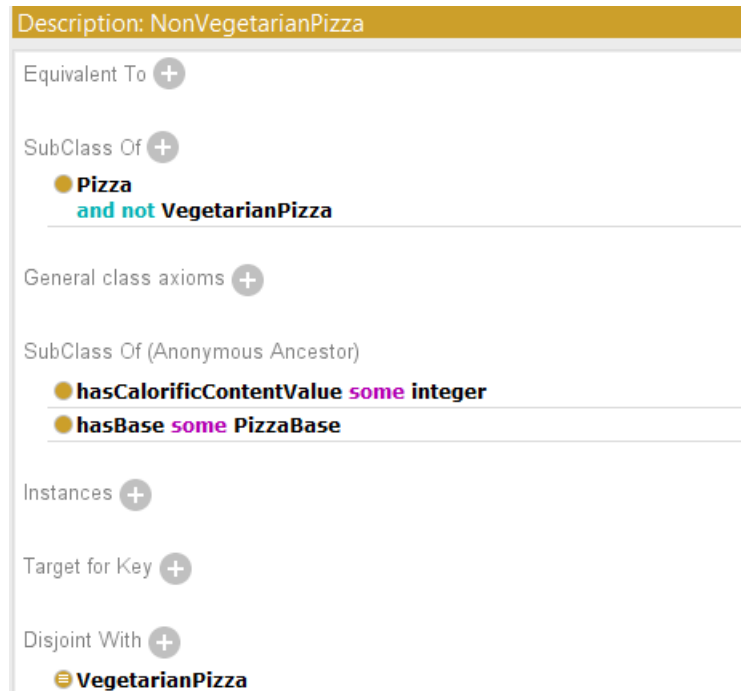


Figure 4.2: The Class Description View Displaying NonVegetarianPizza as a Primitive Class

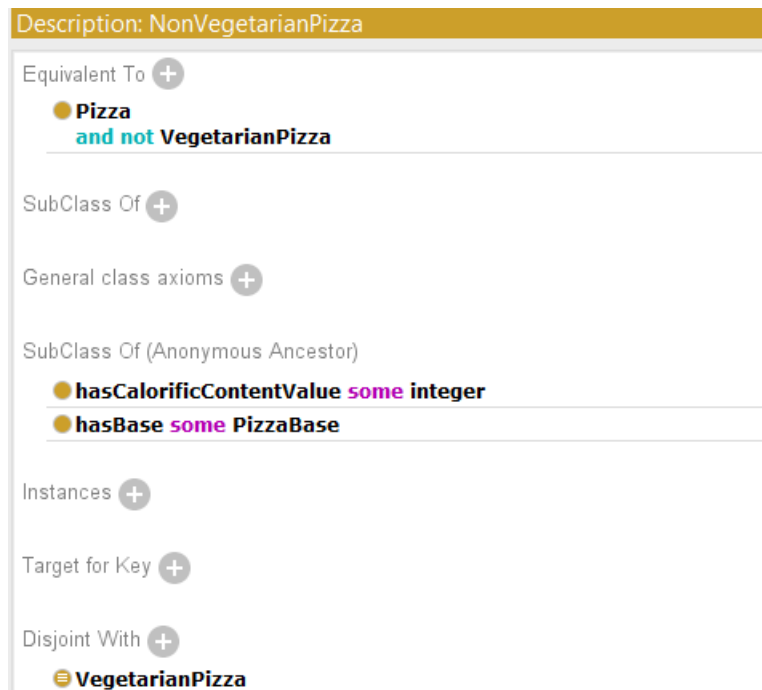



Figure 4.3: The Class Description View Displaying the Definition for NonVegetarianPizza

The inferred class hierarchy should resemble the picture shown in Figure 4.4. As can be seen, MargheritaPizza and SohoPizza have been classified as subclasses of VegetarianPizza. AmericanaPizza and AmericanHotPizza have been classified as NonVegetarianPizza. Things seemed to have worked. However, let's add a pizza that does not have a closure axiom on the hasTopping property.

Exercise 51: Create a subclass of NamedPizza with a topping of Mozzarella

1. Create a subclass of NamedPizza called UnclosedPizza.
2. Making sure that UnclosedPizza is selected.
3. In the class “Description” view, press the “Add” icon () next to “Subclass Of” to see the pop up window.
4. Select the “Class expression editor” tab to display the restriction text box.
5. Type **hasTopping** as the property to be restricted.
6. Type “some” in order to create an existential restriction.
7. Type **MozzarellaTopping** into text box to specify that the toppings must be individuals that are members of the class MozzarellaTopping.
8. Press “OK” to close the dialog and create the restriction

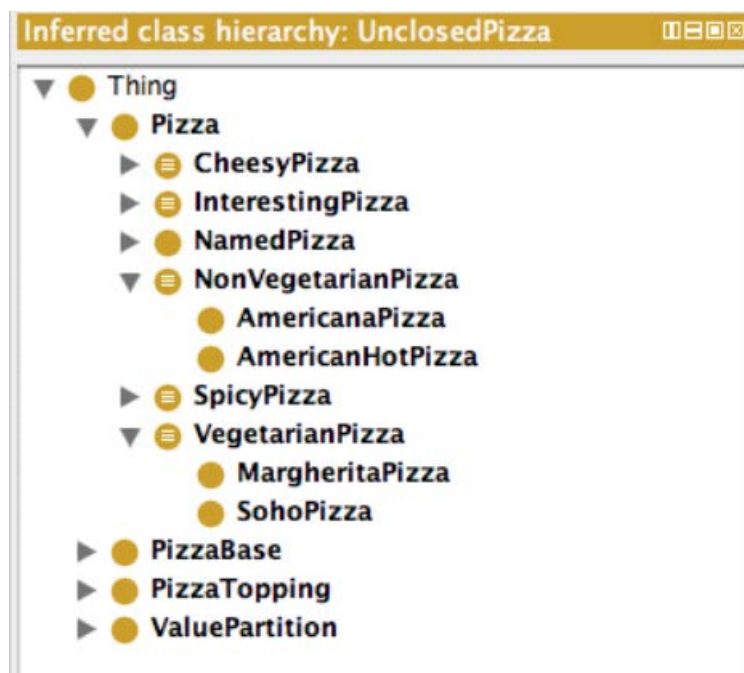


Figure 4.4: The Inferred Class Hierarchy Showing Inferred Subclasses of VegetarianPizza and NonVegetarianPizza

Appendix: How to Create Knowledge Graphs with New Relations in Pace Extended Protégé 5

Knowledge representation is at the core of any knowledge-powered intelligent systems. Currently Ontology and OWL are the industry standard methodology and representation respectively for knowledge representation. But ontology basically only supports one “first-class” relation (a relation that can be applied to any classes without domain/range specification, as we do for “is-a” in OWL) “is-a” or “inheritance” among any classes. For OWL to support additional relations among the classes, unnatural and complicated emulation is needed, leading to great difficulty in knowledge encoding and validation.

Most knowledge cannot be properly represented by ontologies or OWL. For example, algorithms represent most knowledge in computer science, and they heavily depend on the time order or temporal relation not supported by ontology. Most science and engineering knowledge heavily uses “part-of” in various flavors (physical or logical inclusion, intrinsic or optional inclusion, etc.; meronymy). Different knowledge domain needs different relations with various mathematical properties. Knowledge encoding must be executed by domain experts who usually have limited IT skills.

Pace University extended OWL with minimal syntax extension to allow domain experts to declare custom relations with various mathematical properties. The resulting knowledge representation is called Knowledge Graphs. Pace University has also extended Stanford University's project for Protégé v5 to allow domain experts to visually declare custom relations and encode knowledge. This appendix shows how you can use Pace extended Protégé to declare new custom relations among the classes and encode knowledge in Knowledge Graph. Pace University also has reusable software components to read-in Knowledge Graphs and support knowledge-based decision-making in software systems.

This appendix shows you how to declare new custom relations for creating Knowledge Graphs.

1. Open the Protégé IDE. If you are running the Pace revised version for the first time, please click “Window|Reset selected tab to default state” as shown in Figure A.1.

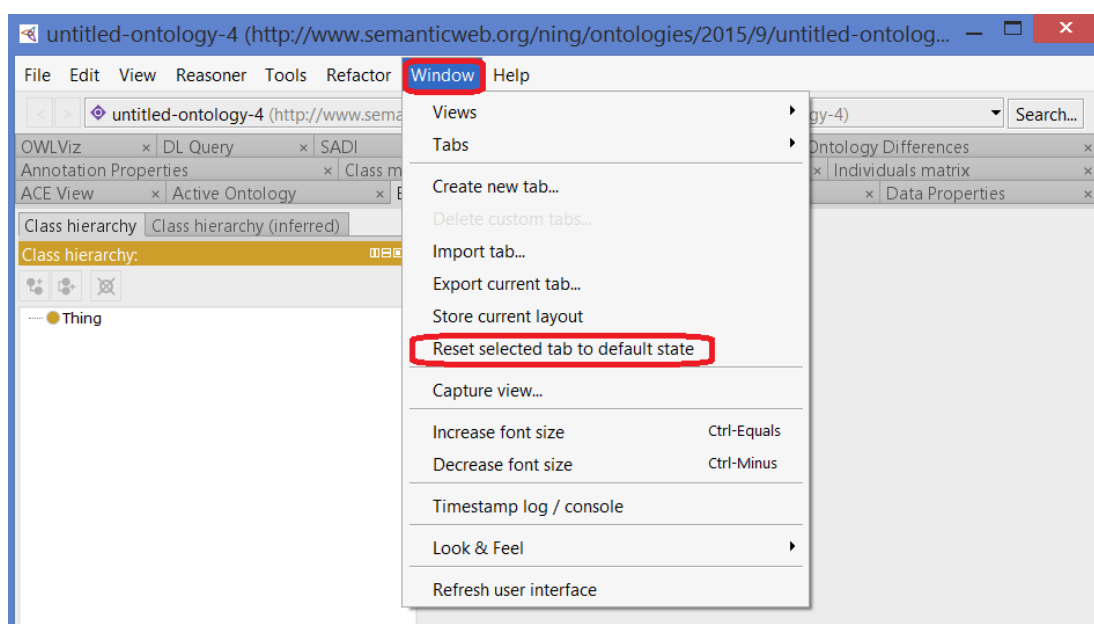




Figure A.1: Resetting the running environment

2. Switch to the “Entities” tab. In the “Class hierarchy” view, select root class **Thing**, and click the “Add sub class” icon () to create class “Finger”. While “Finger” is highlighted, click the “Add sibling class” icon () to create classes “Hand” and “Body” as shown in Figure A.2.

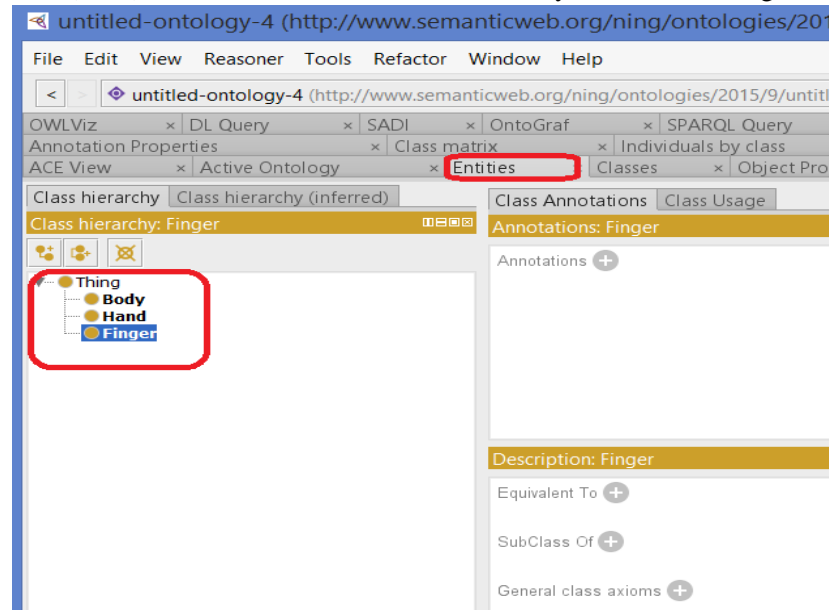



Figure A.2: Create the Necessary Classes

3. Choose the “Relations” tab in the left bottom corner (if you don’t see it, click “Window|Reset selected tab to default state”), and choose the “add relation” icon () to create the new relation “partOf” as shown in Figure 3.

You need to type the name of the new relation, **part of**, in the pop-up window and then click the “OK” button.

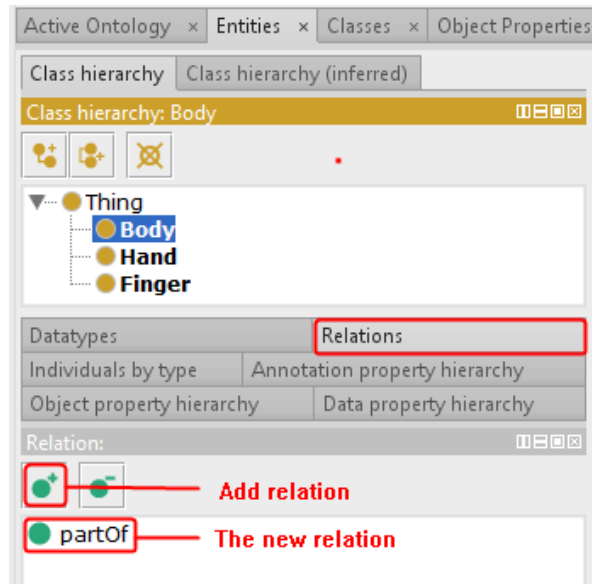



Figure 3: Create a New Relation partOf

4. Switch to “Classes” tab. You need to click “Window|Reset selected tab to default state” again, then click the “Related to” tab in the right pane. Make sure the class “Finger” is selected, and click the “Add relation” icon () as shown in Figure 4.

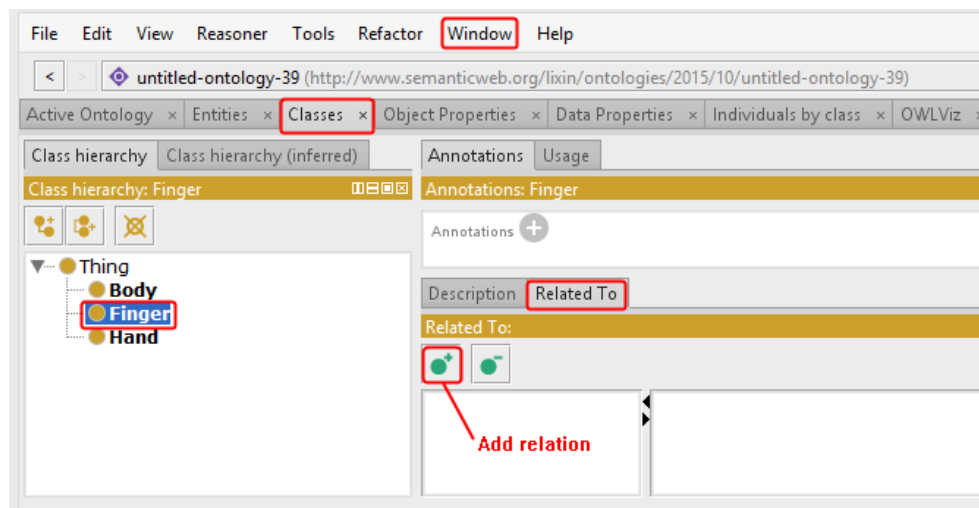


Figure 4: Preparing to Set the Relations between Classes

5. A new window would pop up. First select “partOf” in the left pane of relations, then select “Hand” from the right class list, as shown in Figure 5. Click “OK” to close the pop-up window.

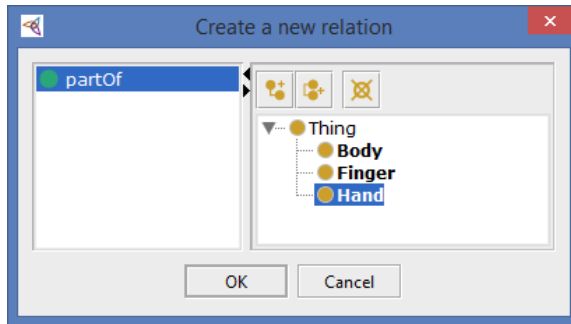



Figure 5: Declare Finger is partOf Hand

- Similar to steps 4 and 5, make sure the class “Hand” is selected in the class hierarchy, click “Add relation” icon () in the “Related To” tab to pop up the window for declaring a new relation, choose “partOf” in its left pane, then select “Body” in its right pane, as shown in Figure 6. Click the “OK” button to complete the declaration.

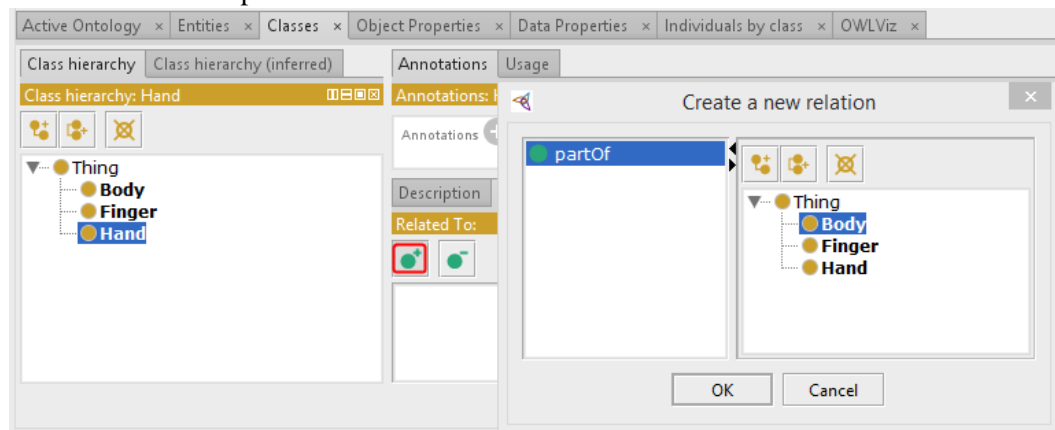


Figure 6: Declare Hand is partOf Body

- Switch to “OWLviz” tab and get the visualization of your OWL file as shown in Figure 7.

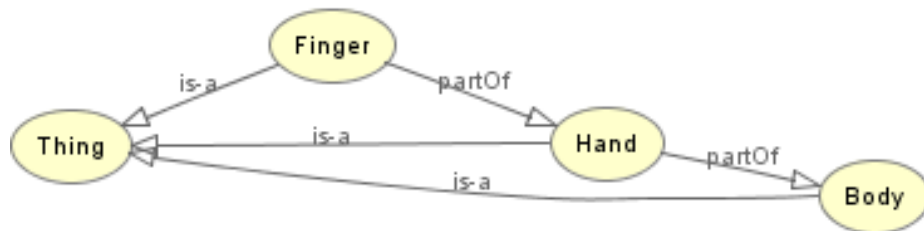


Figure 7: Visualization of the OWL