A Practical Guide to Building OWL Ontologies

Using Protégé 5.5 and Plugins

Edition 3.2

8 October 2021

Michael DeBellis

This is a revised version of the Protégé 4 Tutorial version 1.3 by Matthew Horridge. Previous versions of the tutorial were developed by Holger Knublauch , Alan Rector , Robert Stevens, Chris Wroe, Simon Jupp, Georgina Moulton, Nick Drummond, and Sebastian Brandt.

This work was conducted using the Protégé resource, which is supported by grant GM10331601 from the National Institute of General Medical Sciences of the United States National Institutes of Health.

Chapters 3-5 are based on the original tutorial. I have updated the tutorial to be consistent with Protégé 5. I have also made some changes to address some of the most common issues I've seen new users grapple with, to remove some of the dated information about older frame-based versions of Protégé, and various miscellaneous changes. Chapters 6-11 are new. I have added new sections for technologies such as SWRL, SPARQL and SHACL as well as some details on concepts such as IRIs and namespaces.

Thanks to Matthew Horridge and everyone who worked on the previous tutorials. Special thanks to Lorenz Buehmann who helped me work out a thorny problem as I developed the revised example and to André Wolski for help with the SHACL plugin. Special thanks to Dick Ooms, Colin Pilkington, and Livia Pinera for their excellent detailed feedback on previous versions of the tutorial. Special thanks to Jans Aasman, Yan Xu, and everyone at Franz Inc. for their help utilizing AllegroGraph and Gruff for this tutorial and for a new tutorial available at: https://www.michaeldebellis.com/post/the-people_example-ontology Also, thanks to everyone on the Protégé user support email list.

Note: this document may get updates frequently. It is a good idea to check my blog at: https://www.michaeldebellis.com/post/new-protege-pizza-tutorial to make sure you have the latest version.

If you have questions or comments, feel free to contact me at mdebellissf@gmail.com

Contents

Chapter 1 Introduction

This introduces Protégé 5 for creating OWL ontologies as well as various plugins. If you have questions specific to this tutorial, please feel free to email me directly: mdebellissf@gmail.com However, if you have general questions about Protégé, OWL, or plugins you should subscribe to and send an email to the User Support for Protégé and Web Protégé email list. This list has many people (including me) who monitor it and can contribute their knowledge to help you understand how to get the most out of this technology. To subscribe to the list, go to: https://protege.stanford.edu/support.php and click on the first orange Subscribe button. That will enable you to subscribe to the list and give you the email to send questions to.

This chapter covers licensing and describes conventions used in the tutorial. Chapter 2 covers the requirements for the tutorial and describes the Protégé user interface. Chapter 3 gives a brief overview of the OWL ontology language. Chapter 4 focuses on building an OWL ontology with classes and object properties. Chapter 4 also describes using a Description Logic Reasoner to check the consistency of the ontology and automatically compute the ontology class hierarchy.

Chapter 5 describes data properties. Chapter 6 describes design patterns and shows one design pattern: adding an order to an enumerated class. Chapter 7 describes the various concepts related to the name of an OWL entity.

Chapter 8 introduces an extended version of the Pizza tutorial developed in chapters 1-7. This ontology has a small number of instances and property values already created which can be used to illustrate the tools in the later chapters for writing rules, doing queries, and defining constraints.

Chapter 9 describes two tools for doing queries: Description Logic queries and SPARQL queries. Chapter 10 introduces the Semantic Web Rule Language (SWRL) and walks you through creating SWRL and SQWRL rules. Chapter 11 introduces the Shapes Constraint Language (SHACL) and discusses the difference between defining logical axioms in Description Logic and data integrity constraints in SHACL. Chapter 12 has some concluding thoughts and opinions and Chapter 13 provides a bibliography.

1.1 Licensing

1.2 Conventions

Class, property, rule, and individual names are written in `Consolas` font `like this`. The term used for any such construct in Protégé and in this document is an *Entity*. Individuals and classes can also be referred to as objects.

Names for user interface tabs, views, menu selections, buttons, and text entry are highlighted like this.

Any time you see highlighted text such as File>Preferences or OK or PizzaTopping it refers to something that you should or optionally could view or enter into the user interface. If you ever aren't sure what to do to accomplish some task look for the highlighted text. Often, as with `PizzaTopping` the text you enter into a field in the Protégé UI will be the name of a class, property, etc. In those cases, where the

name is meant to be entered into a field it will only be highlighted rather than highlighted and printed in `Consolas` font.

Menu options are shown with the name of the top-level menu, followed by a > followed by the next level down to the desired selection. For example, to indicate how to open the Individuals by class tab under the Tabs section in the Window menu the following text would be used: Window>Tabs> Individuals by class.

When a word or phrase is emphasized, it is *shown in italics like this.*

Exercises are presented like this:

Exercise 1: Accomplish this

---

1. Do this.
2. Then do this.
3. Then do this.

Potential pitfalls and warnings are presented like this.

Tips and suggestions related to using Protégé are presented like this.

Explanations as to what things mean are presented like this.

General notes are presented like this.

Vocabulary explanations and alternative names are presented like this.

Chapter 2 Requirements and the Protégé User Interface

In order to follow this tutorial, you must have Protégé 5, which is available from the Protégé website,[1] and some of the Protégé Plugins which will be described in more detail below. For now, just make sure you have the latest version of Protégé. At the time this is being written the latest version is 5.5 although the tutorial should work for later versions as well.

The Protégé user interface is divided up into a set of major tabs. These tabs can be seen in the Window>Tabs option. This option shows all the UI tabs that are currently loaded into the Protégé environment. Any tabs that are currently opened have a check mark next to them. To see a tab that is not visible just select it from the menu and it will be added to the top with the other major tabs and its menu item will now be checked. You can add additional major tabs to your environment by loading plugins. For example, when we load the SHACL4Protégé plugin the SHACLEditor will be added to the menu.

Each major tab consists of various panes or as Protégé calls them views. Each view can be resized or closed using the icons in the top right corner of every view. The views can also be nested as sub-tabs within each major tab. When there could potentially be confusion between a tab that is a screen all its own (is under the Window>Tabs option) and a view that is a sub-tab we will call the screen tab a major tab. There are many views that are not in the default version of Protégé that can be added via the Window>Views option. The additional views are divided into various categories such as Window>Views>Individual views. Section 5.2 will show an example of adding a new view to a major tab.

Chapter 3 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). A good primer on the basic concepts of OWL can be found at: https://www.w3.org/TR/owl2-primer/

OWL makes it possible to describe concepts in an unambiguous manner based on set theory and logic. Complex concepts can be built up out of simpler concepts. The logical model allows the use of a reasoner which can check whether 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. The reasoner can also infer additional information. For example, if two properties are inverses only one value needs to be asserted by the user and the inverse value will be automatically inferred by the reasoner.

3.1 Components of OWL Ontologies

An OWL ontology consists of Classes, Properties, and Individuals. OWL ontologies are an implementation of Description Logic (DL) which is a decidable subset of First Order Logic. A class in OWL is a set, a property is a binary relation, and an individual is an element of a set. Other concepts from set theory are also implemented in OWL such as Disjoint sets, the Empty set (`owl:Nothing`), inverse

[1] http://protege.stanford.edu

relations, transitive relations, and many more. An understanding of the basic concepts of set theory will help the user get the most out of OWL but is not required. One of the benefits of Protégé is that it presents an intuitive GUI that enables domain experts to define models without a background in set theory. However, developers are encouraged to refresh their knowledge on logic and set theory. A good source is the first 3 chapters in Elements of the Theory of Computation by Lewis and Papadamitrious. Another good source is the PDF document *Overview of Set Theory* available at:
https://www.michaeldebellis.com/post/owl-theoretical-basics

3.1.1 Individuals

Individuals represent objects in the domain of interest. An important difference between OWL and most programming and knowledge representation languages is that 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 from each other. Figure 3.1 shows a representation of some individuals in a domain of people, nations, and relations — in this tutorial we represent individuals as diamonds.

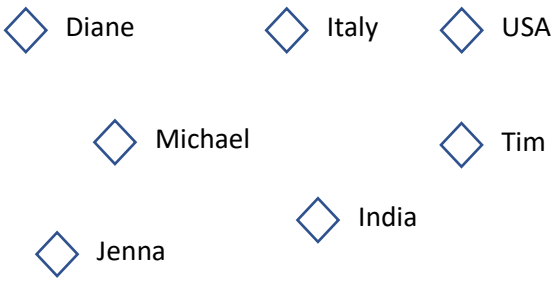


Figure 3.1: Representation of Individuals

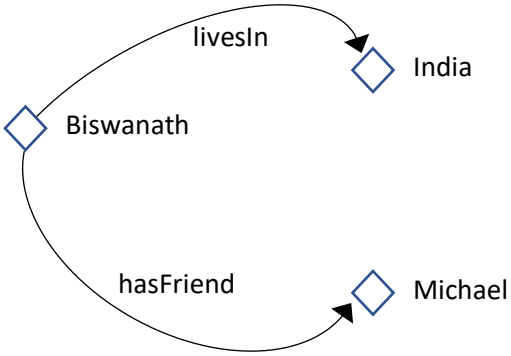


Figure 3.2: Representation of Properties

Individuals are also known as *instances*. Individuals can be referred to as *instances of classes*.

3.1.2 Properties

Properties are binary relations between individuals. I.e., properties link two individuals together. For example, the property `hasFriend` might link the individual `Biswanath` to the individual `Michael`, or the property `hasChild` might link the individual `Michael` to the individual `Oriana`. Properties can have inverses. For example, the inverse of `hasChild` is `hasParent`. Properties can be limited to having a single value – i.e., to being functional. They can also be transitive or symmetric. These property characteristics are explained in detail in Section 4.8. Figure 3.2 shows a representation of some properties.

Properties are similar to properties in Object-Oriented Programming (OOP). However, there are important differences between properties in OWL and OOP. The most important difference is that OWL properties are first class entities that exist independent of classes. OOP developers are encouraged to read: https://www.w3.org/2001/sw/BestPractices/SE/ODSD/

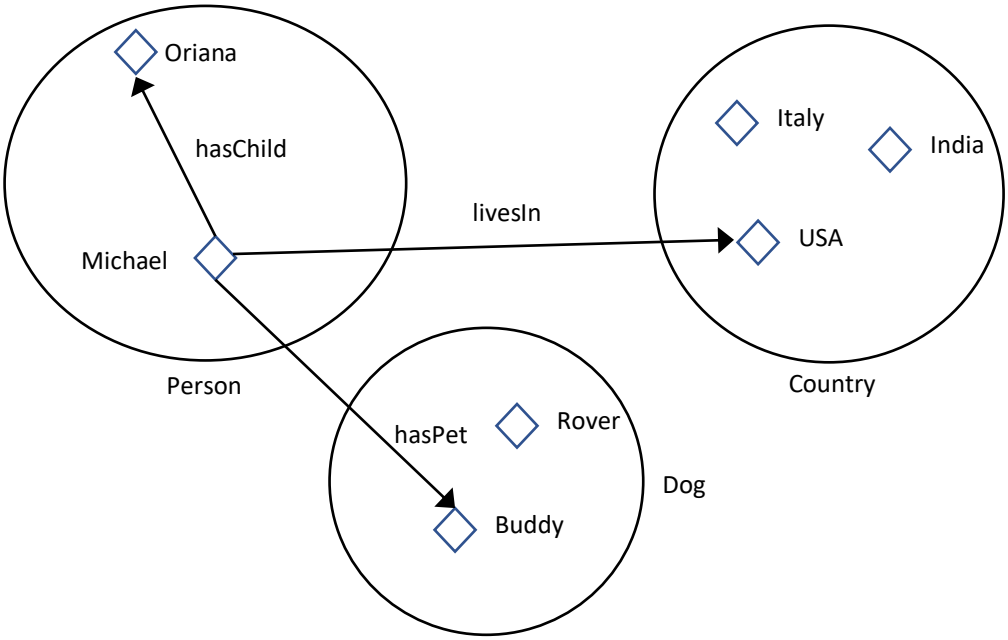


Figure 3.3: Representation of Classes containing Individuals

3.1.3 Classes

OWL classes are sets that contain individuals. They are described using formal (mathematical) descriptions that rigorously define 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.[2] Classes may be organized into a superclass-subclass hierarchy, which is also known as a taxonomy. However, taxonomies are often trees. I.e., each node has only one parent node. Class hierarchies in OWL are not restricted to be trees and multiple inheritance can be a powerful tool to represent data in an intuitive manner.

Subclasses specialize (aka *are subsumed by*) their superclasses. For example, consider the classes `Animal` and `Dog` – `Dog` might be a subclass of `Animal` (so `Animal` is the superclass of `Dog`). This says that *All dogs are animals*, *All members of the class* `Dog` *are members of the class* `Animal`. OWL and Protégé

[2] Individuals can belong to more than one class and classes can have more than one superclass. Unlike OOP where multiple inheritance is typically unavailable or discouraged it is common in OWL.

provide a language that is called Description Logic or DL for short. One of the key features of DL is that these superclass-subclass relationships (aka subsumption relationships) can be computed automatically by a reasoner – more on this later. Figure 3.3 shows a representation of some classes containing individuals – classes are represented as ovals, like sets in Venn diagrams.

In OWL classes can be 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.

Chapter 4 Building an OWL Ontology

This chapter describes how to create an ontology of Pizzas. We use Pizzas because it is something almost everyone is familiar with.

Exercise 1: Create a new OWL Ontology

---

1. Start Protégé. When Protégé opens for the first time each day it puts up a screen of all the available plugins. You can also bring this up at any time by using File>Check for plugins. You won't need any plugins at this point of the tutorial so just click the Not now button.

2. The Protégé user-interface consists of several tabs such as Active ontology, Entities, etc. When you start Protégé you should be in the Active Ontology tab. This is for overview information about the entire ontology. Protégé always opens with a new untitled ontology you can start with. Your ontology should have an IRI something like: http://www.semanticweb.org/yourname/ontologies/2020/4/untitled-ontology-27 Edit the name of the ontology (the part after the last "/" in this case untitled-ontology-27) and change it to something like PizzaTutorial. Note: the Pizza ontology IRIs shown below (e.g., figure 4.3) show the IRI after I edited the default that Protégé generated for me. Your IRI will look different and will be based on your name or the name of your organization.

3. Now you want to save your new ontology. Select File>Save. This should bring up a window that says: Choose a format to use when saving the 'PizzaTutorial' ontology. There is a drop down menu of formats to use. The default RDF/XML Syntax should be selected by clicking the OK button. This should bring up the standard dialog your operating system uses for saving files. Navigate to the folder you want to use and then type in the file name, something like Pizza Tutorial and select Save.

As with any file you work on it is a good idea to save your work at regular intervals so that if something goes wrong you don't lose your work. At certain points in the tutorial where saving is especially important the tutorial will prompt you to do so but it is a good idea to save your work often, not just when prompted.

The next step is to set some preferences related to the names of new entities. Remember than in Protégé any class, individual, object property, data property, annotation property, or rule is referred to as an entity. The term name in OWL can actually refer to two different concepts. It can be the last part of the IRI[3] or it can refer to the annotation property (usually `rdfs:label`) used to provide a more user friendly name for the entity. We will discuss this in more detail below in chapter 7. For now, we just want to set the parameters correctly so that future parts of the tutorial (especially the section on SPARQL queries) will work appropriately.

[3] An IRI is similar to a URL. This will be discussed in detail below in chapter 7.

Exercise 2: Set the Preferences for New Entities and Rendering

---

1. Go to File>Preferences in Protégé. This will bring up a new window with lots and lots of different tabs. Click on the New entities tab. This will bring up a tab that looks similar to figure 4.1. The top part of that tab is a box labeled Entity IRI. It should be set with the parameters as shown in figure 4.1. I.e., Starts with Active ontology IRI. Followed by #. Ends with User supplied name. If the last parameter is set to Auto-generated name change it to User supplied name. That is the parameter most likely to be different but also check the other two as well.

2. Now select the Renderer tab. It should look like figure 4.2. Most importantly, check that Entity rendering is set to Render by entity IRI short name (ID) rather than Render by annotation property. Don't worry if this doesn't completely make sense at this point. The issues here are a bit complex and subtle so we defer them until after you have an understanding of the basic concepts of what an OWL ontology is. We will have a discussion of these details below in chapter 7. For now you just need to make sure that the preferences are set appropriately to work with the rest of the tutorial.

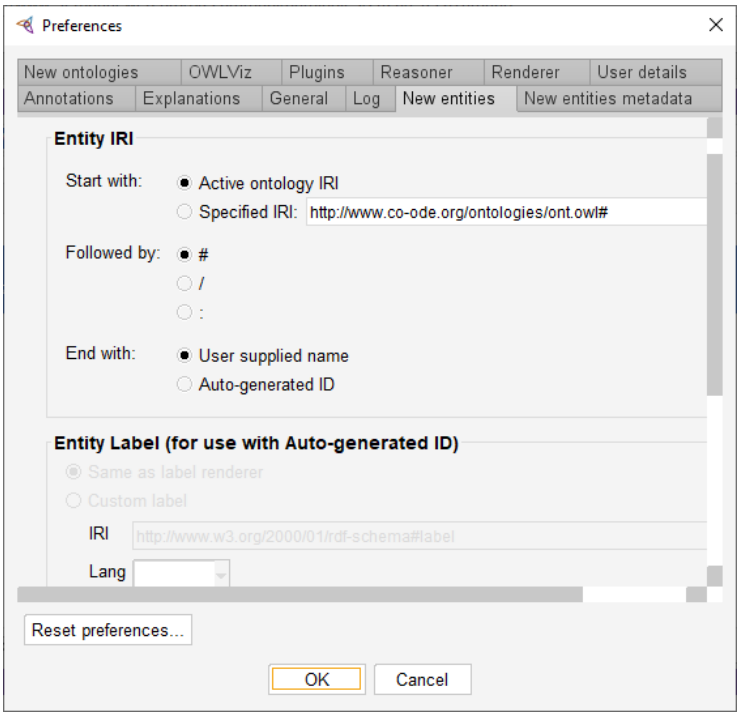


Figure 4.1: The New entities tab

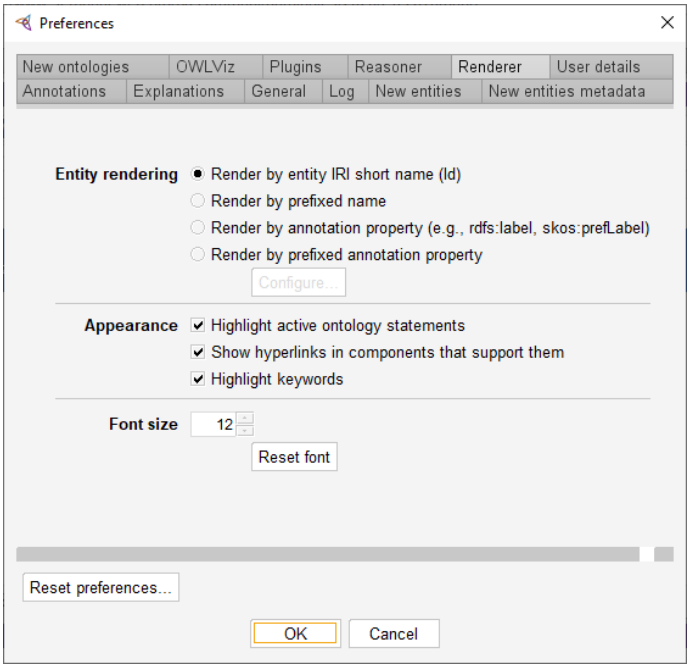


Figure 4.2 Renderer tab

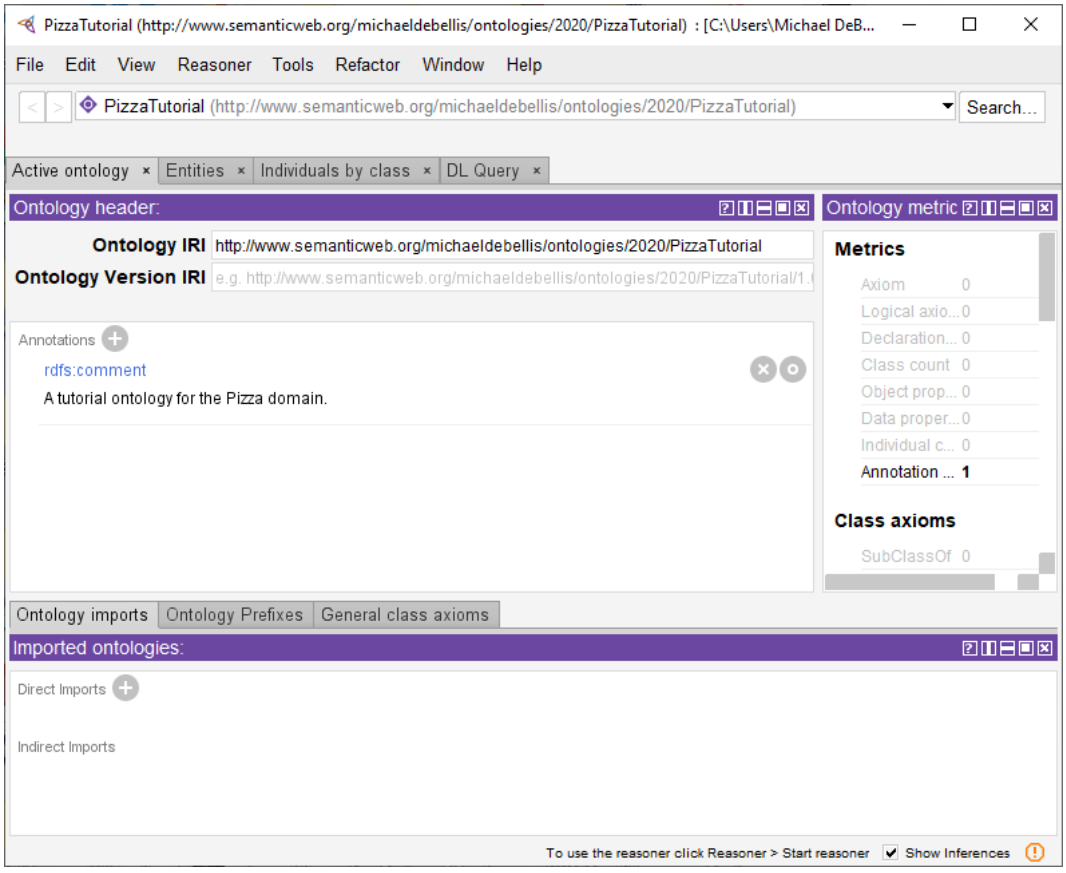


Figure 4.3: The Active Ontology Tab with a New Comment

Exercise 3: Add a Comment Annotation to Your Ontology

---

1. Make sure you are in the Active Ontology tab. In the view just below the Ontology IRI and Ontology Version IRI fields find the Annotations option and click on the + sign. This will bring up a menu to create a new annotation on the ontology.

2. The rdfs:comment annotation should be highlighted by default. If it isn't highlighted click on it. Then type a new comment into the view to the right. Something like A tutorial ontology for the Pizza domain.

3. Click OK. Your Active Ontology tab should like Figure 4.3.

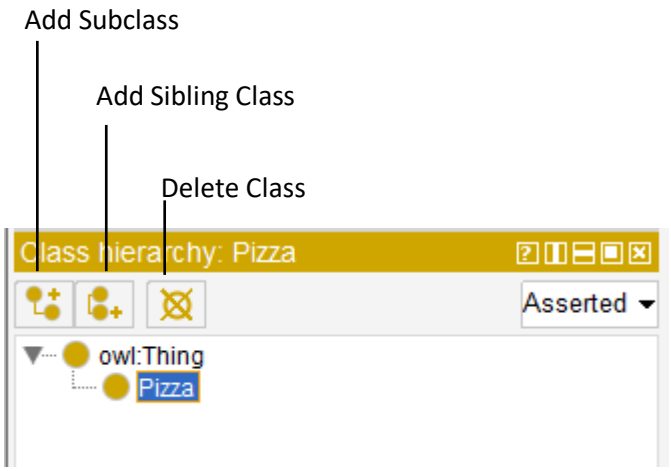


Figure 4.4: The Class Hierarchy View Options

4.1 Named Classes

The main building blocks of an OWL ontology are classes. In Protégé 5, editing of classes can be done in the Entities tab. The Entities tab has a number of sub-tabs. When you select it, the default should be the Class hierarchy view as shown in Figure 4.5.[4] All empty ontologies contains one class called `owl:Thing`. OWL classes are sets of individuals. The class `owl:Thing` is the class that represents the set containing all individuals. Because of this all classes are subclasses of `owl:Thing`.

[4] Each of the sub-tabs in the Entities tab also exists as its own major tab. In the tutorial we will refer to tabs like the Class hierarchy tab or Object properties tab and it is up to the user whether to access them from the Entities tab or to create them as independent tabs.

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

---

1. Navigate to the Entities tab[5] with the Class hierarchy view selected. Make sure owl:Thing is selected.

2. Press the Add Subclass icon shown in figure 4.4. This button creates a new subclass of the selected class. In this case we want to create a subclass of `owl:Thing`.

3. This should bring up a dialog titled Create a new class with a field for the name of the new class. Type in Pizza and then select OK.

4. Repeat the previous steps to add the classes `PizzaTopping` and `PizzaBase` ensuring that `owl:Thing` is selected before using the add subclass icon so that all your classes are subclasses of `owl:Thing`. Your user interface should now look like figure 4.5. Don't worry that some of the classes are highlighted in red. That is because the reasoner hasn't run yet. We will address this shortly.

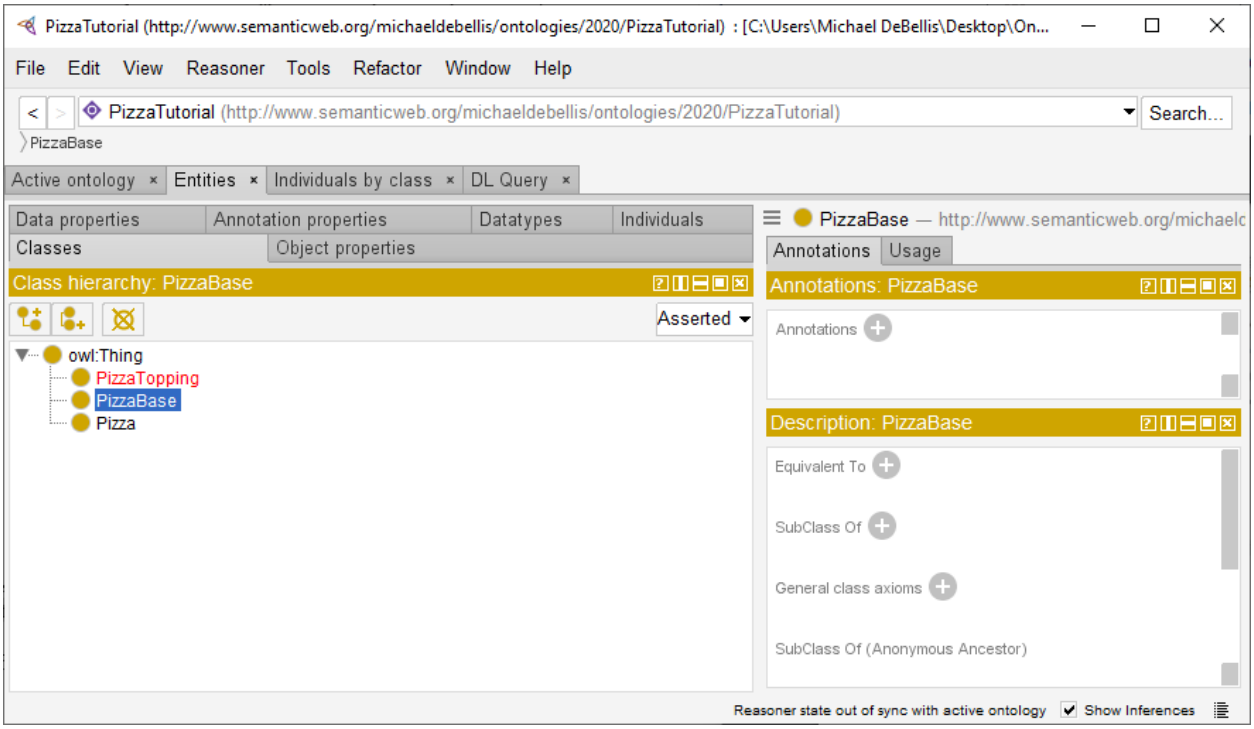


Figure 4.5 The Classes Sub-Tab in the Entities Tab

---

[5] The Entities tab is a big tab that has tabs like Classes, Object properties, Data properties, etc. as sub-tabs. Each of these sub-tabs is also a major tab (a tab accessible from the Window>Tabs option) that can be created on its own. Since I took screen snapshots at various times I wasn't always completely consistent. Sometimes I used Classes as a sub-tab of the Entities tab and sometimes as a major tab on its own. Also, at different points in time I had other tabs open depending on what other work I was doing. Thus, your UI won't look identical to the figures. There may be additional tabs in the figure that aren't in your UI or vice versa.

There are no mandatory naming conventions for OWL entities. In chapter 7, we will discuss names and labels in more detail. A best practice is to select one set of naming conventions and then abide by that convention across your organization. For this tutorial we will follow the standard where class and individual names start with a capital letter for each word and do not contain spaces. This is known as CamelBack notation. For example: `Pizza`, `PizzaTopping`, etc. Also, we will follow the standard that class names are always singular rather than plural. E.g., `Pizza` rather than `Pizzas`, `PizzaTopping` rather than `PizzaToppings`.

4.2 Using a Reasoner

You may notice that one or more of your classes is highlighted in red as in Figure 4.5. This is because we haven't run the reasoner yet so Protégé has not been able to verify that our new classes have no inconsistencies. When just creating classes and subclasses in a new ontology there is little chance of an inconsistency. However, it is a good idea to run the reasoner often. When there is an inconsistency the sooner it is discovered the easier it is to fix. One common mistake that new users make is to do a lot of development and then run the reasoner only to find that there are multiple inconsistencies which can make debugging significantly more difficult. So let's get into the good habit of running the reasoner often. Protégé comes with some reasoners bundled in and others available as plugins. Since we are going to write some SWRL rules later in the tutorial, we want to use the Pellet reasoner. It has the best support for SWRL at the time this tutorial is being written.

Exercise 5: Install and Run the Pellet Reasoner

---

1. Check to see if the Pellet reasoner is installed. Click on the Reasoner menu. At the bottom of the menu there will be a list of the installed reasoners such as Hermit and possibly Pellet. If Pellet is visible in that menu then select it and skip to step 3.

2. If Pellet is not visible then do File>Check for plugins and select Pellet from the list of available plugins and then select Install. This will install Pellet and you should get a message that says it will take effect the next time you start Protégé. Do a File>Save to save your work then quit Protégé and restart it. Then go to File>Open recent. You should see your saved Pizza tutorial in the list of recent ontologies. Select it to load it. Now you should see Pellet under the Reasoner menu and be able to select it so do so.

3. With Pellet selected in the Reasoner menu execute the command Reasoner>Start reasoner. The reasoner should run very quickly since the ontology is so simple. You will notice that the little text message in the lower right corner of the Protégé window has changed to now say Reasoner active. The next time you make a change to the ontology that text will change to say: Reasoner state out of sync with active ontology. With small ontologies the reasoner runs very quickly, and it is a good idea to get into the habit of running it often, as much as after every change.

4. It is possible that one or more of your classes will still be highlighted in red after you run the reasoner. If that happens do: Window>Refresh user interface and any red highlights should go away. Whenever your user interface seems to show something you don't expect the first thing to do is to try this command.

5. One last thing we want to do is to configure the reasoner. By default, the reasoner does not perform all possible inferences because some inferences can take a long time for large and complex ontologies. In this tutorial we will always be dealing with small and simple ontologies so we want to see everything the reasoner can do. Go to: Reasoner>Configure. This will bring up a dialog with several check boxes of inferences that the reasoner can perform. If they aren't all checked then check them all. You may receive a warning that some inferences can take a lot of time, but you can ignore those since your ontology will be small.

4.3 Disjoint Classes

Having added the classes `Pizza`, `PizzaTopping`, and `PizzaBase` to the ontology, we now want to say that these classes are *disjoint*. I.e., no individual can be an instance of more than one of those classes. In set theory terminology the intersection of these three classes is the empty set: `owl:Nothing`.

Exercise 6: Make Pizza, PizzaTopping, and PizzaBase disjoint from each other

---

1. Select the class Pizza in the class hierarchy.

2. Find the Disjoint With option in the Description view and select the (+) sign next to it. See the red circle in figure 4.6.

3. This should bring up a dialog with two tabs: Class hierarchy and Expression editor. You want Class hierarchy for now (we will use the expression editor later). This gives you an interface to select a class that is identical to the Class hierarchy view. Use it to navigate to PizzaBase. Hold down the shift key and select PizzaBase and PizzaTopping. Select OK.

4. Do a Reasoner>Synchronize reasoner. Then look at `PizzaBase` and `PizzaTopping`. You should see that they each have the appropriate disjoint axioms defined to indicate that each of these classes is disjoint with the other two.

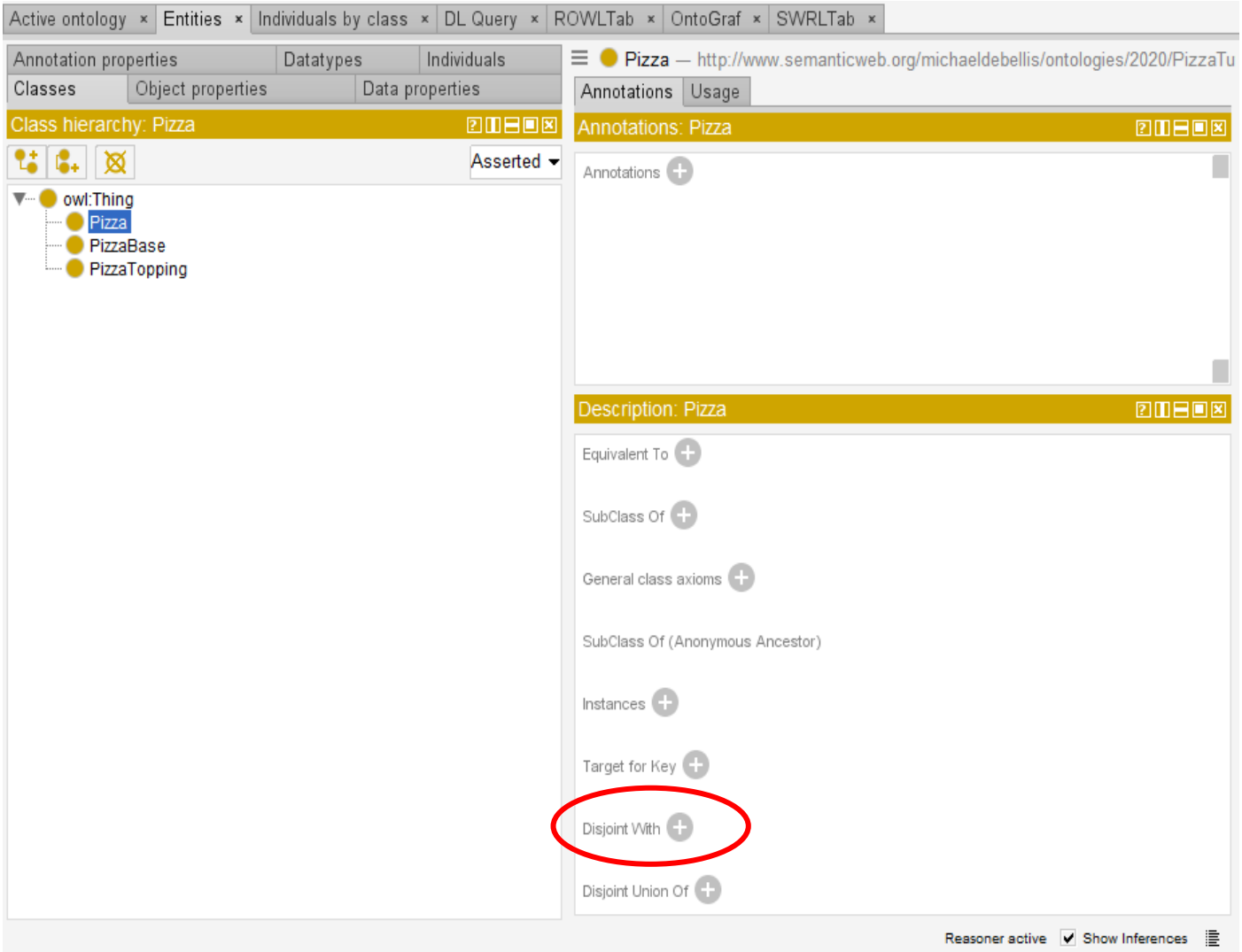


Figure 4.6: The Disjoint Option in the Class Description View

OWL classes are assumed to overlap, i.e., by default they are not disjoint. This is often useful because in OWL, unlike in most object-oriented models, multiple inheritance is not discouraged and can be a powerful tool to model data. If we want classes to be disjoint, we must explicitly declare them to be so. It is often a good development strategy to start with classes that are not disjoint and then make them disjoint once the model is more fully fleshed out as it is not always obvious which classes are disjoint from the beginning.

4.4 Using Create Class Hierarchy

In this section we will use Tools>Create class hierarchy to create multiple classes at once.

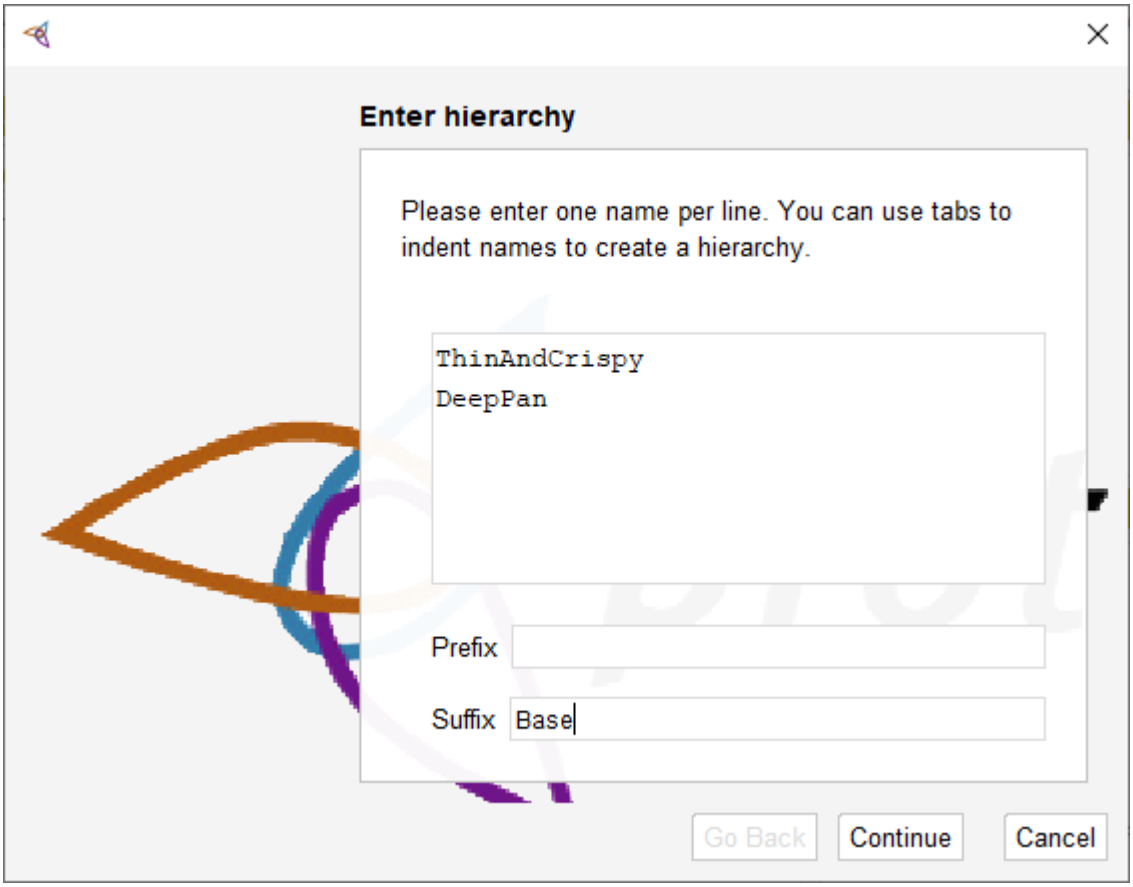


Figure 4.7: The Create class hierarchy wizard

Exercise 7: Use the Create class hierarchy tool to create subclasses of PizzaBase

---

1. Select the class PizzaBase in the class hierarchy.

2. With PizzaBase selected use the Tools>Create class hierarchy menu option.

3. This should bring up a wizard that enables you to create a nested group of classes all at once. You should see a window labeled Enter hierarchy where you can enter one name on each line. You can also use the tab key to indicate that a class is a subclass of the class above it. For now we just want to enter two subclasses of `PizzaBase`: `ThinAndCrispyBase` and `DeepPanBase`. One of the things the wizard does is to automatically add a prefix or suffix for us. So just enter ThinAndCrispy, hit return and enter DeepPan. Then in the Suffix field add Base. Your window should look like figure 4.7.

4. Select Continue. This will take you to a window that asks if you want to make sibling classes disjoint. The default should be checked (make them disjoint) which is what we want in this case (a base can't be both deep pan and thin) so just select Finish. Synchronize the reasoner. Your class hierarchy should now look like figure 4.8.

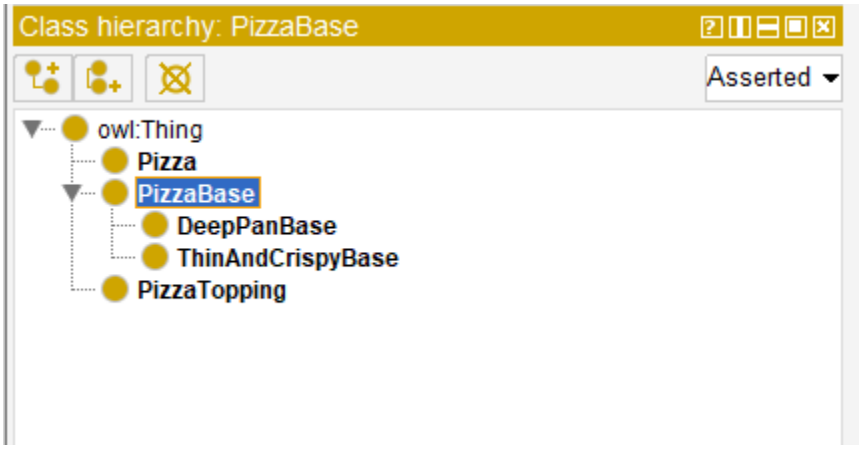


Figure 4.8: The New Class Hierarchy

4.5 Create a PizzaTopping Hierarchy

We will use Tools>Create class hierarchy again but this time to create a more interesting hierarchy with additional subclasses to model the subclasses of `PizzaTopping`.

Exercise 8: Create subclasses of PizzaTopping

---

1. Select the class PizzaTopping in the class hierarchy.

2. With PizzaTopping selected use the Tools>Create class hierarchy menu option.

3. This will once again bring up the wizard. We want all our toppings to end in Topping so enter Topping in the Suffix field. Then create the nested structure as shown in figure 4.9. Use the Tab key to indent classes where needed.

4. Select Continue. This will take you to the window that asks if you want to make sibling classes disjoint. We do want this so leave the box checked and click Finish. Synchronize the reasoner. Your class hierarchy should now look like figure 4.10.

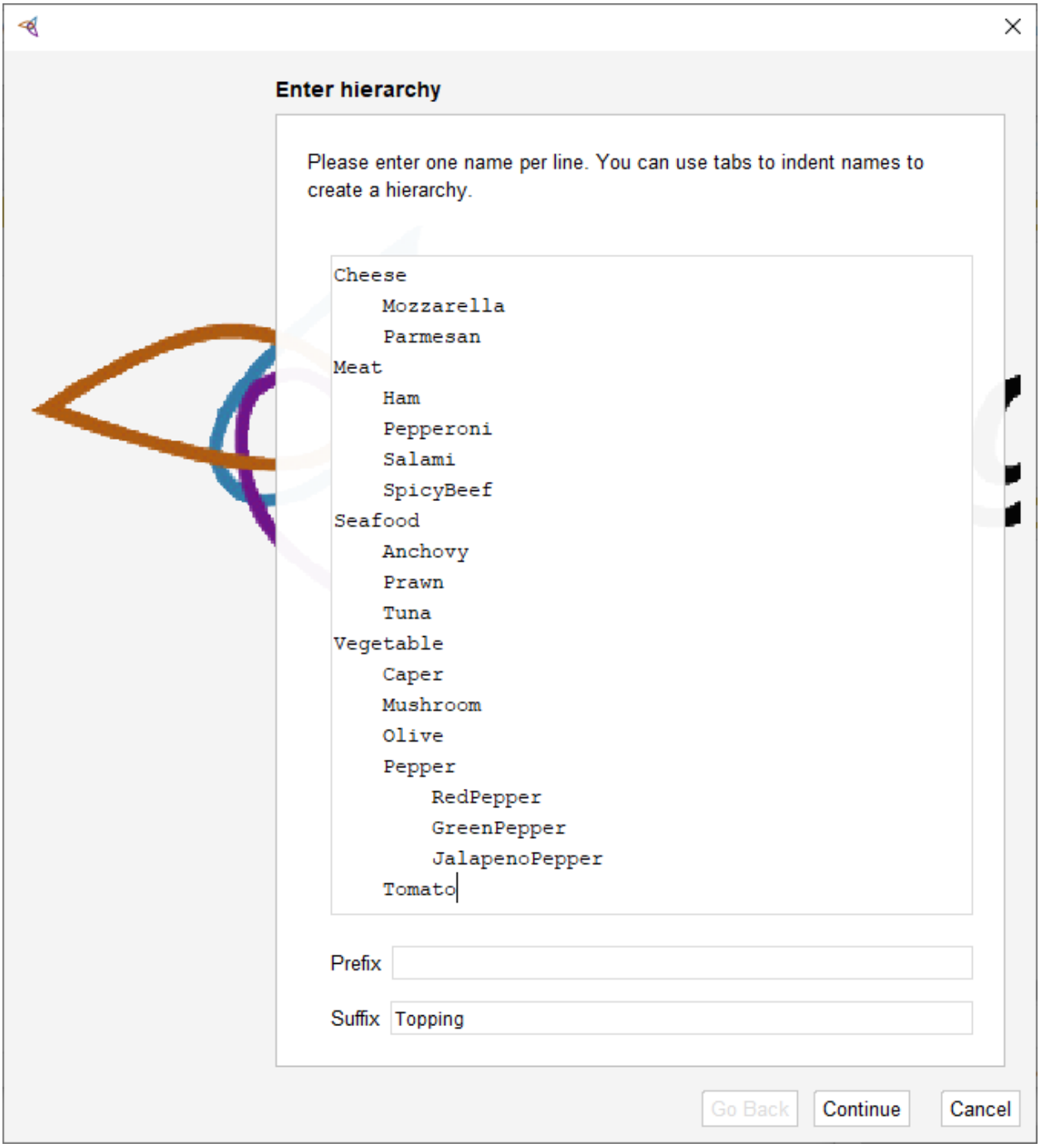


Figure 4.9 Using Create class hiearchy to create `PizzaTopping` subclasses

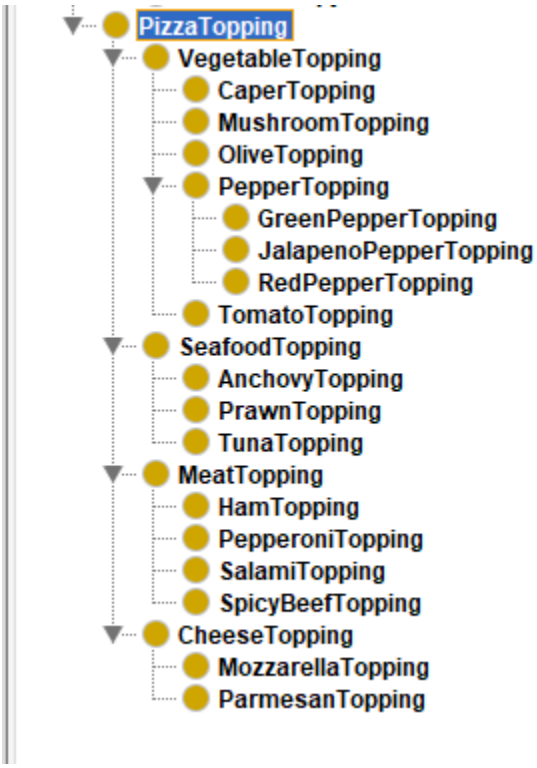
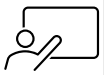


Figure 4.10 The New `PizzaTopping` Class Hierarchy



So far, we have created some simple named classes and subclasses which hopefully seem intuitive and obvious. However, what does it actually mean to be a subclass of something in OWL? For example, what does it mean for `VegetableTopping` to be a subclass of `PizzaTopping`? In OWL subclass means *necessary implication*. I.e., if `VegetableTopping` is a subclass of `PizzaTopping` then *all* instances of `VegetableTopping` are also instances of `PizzaTopping`. It is for this reason that we try to have standards such as having all `PizzaTopping` classes end with the word "Topping". Otherwise, it might seem we are saying that anything that is a kind of *Ham* like the *Ham* in your sandwich is a kind of `MeatTopping` or `PizzaTopping` which is not what we mean. For large ontologies strict attention to the naming of classes and other entities can prevent potential confusion and bugs.

4.6 OWL Properties

OWL Properties represent relationships. There are three types of properties, Object properties, Data properties and Annotation properties. Object properties are relationships between two individuals. Data properties are relations between an individual and a datatype such as xsd:string or xsd:dateTime. Annotation properties also usually have datatypes as values although they can have objects. An annotation property is usually meta-data such as a comment or a label. In OWL only individuals can have values for object and data properties, but any entity can have an annotation property value since meta-data applies to all entities. Annotation properties usually can't be reasoned about. For example, SWRL rules which we will cover later cannot view or change the value of annotation properties. In this chapter we will focus on Object properties. Data properties are described in Chapter 5. In the current version of the tutorial we are only discussing the annotation property `rdfs:label` (see chapter 7) however they are fairly intuitive.

Properties may be created using the Object Properties sub-tab of the Entities tab shown in figure 4.11. Just as all OWL classes ultimately are a subclass of `owl:Thing`, all properties are ultimately a sub-property of `owl:topObjectProperty`. A sub-property is similar to a subclass except it is about the tuples in a property. For example, `hasFather` would be a sub-property of `hasParent` because all the tuples in `hasFather` are in `hasParent` but not vice versa. E.g., if `Sasha hasFather Barack` then she also `hasParent Barack`. However, she also `hasParent Michelle` but it is not the case that she `hasFather Michelle`. Rather she `hasMother Michelle`, i.e., `hasMother` is also a sub-property of `hasParent`.

The GUI for entering properties is also similar to that for entering classes. The first icon with one box under another creates a sub-property of the selected property. The second icon showing two boxes at the same level creates a sibling property to the selected property and the icon with an X through a box deletes the selected property.

Exercise 9: Create some properties

---

1. Select the Object properties sub-tab of the Entities tab (see figure 4.11).

2. Make sure `owl:topObjectProperty` is selected**.** Click on the nested box icon at the left to create a new sub-property of `owl:topObjectProperty.` When prompted for the name of the new property type in hasIngredient.

3. Just as you can use a wizard to create multiple classes you can also use one to create multiple properties. Select hasIngredient and then select Tools>Create object property hierarchy. Enter the new property names hasTopping and hasBase. Select Continue and accept the default that the object properties are *not* disjoint.

4. Synchronize the reasoner. Your window should now look like figure 4.11.

For those familiar with the Entity-Relationship model, OWL object properties are similar to relations and data properties are similar to attributes. Object properties are similar to properties with a range of some class in OOP and data properties are similar to OOP properties with a range that is a datatype.

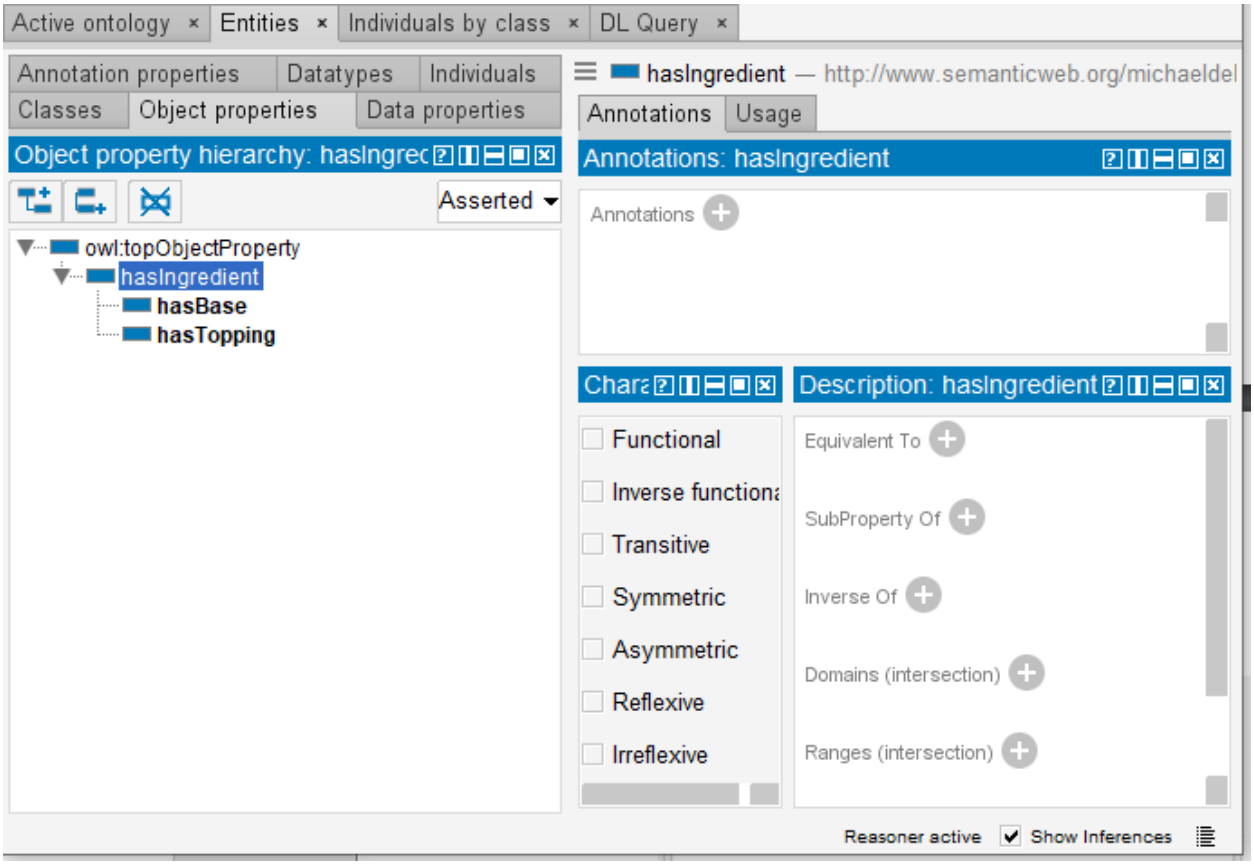


Figure 4.11 Adding Some Object Properties

4.7 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, in figure 3.3 the individual `Michael hasPet Buddy`. In this example `hasPet` is an object property that maps from a `Person` to their `Pet` which are known as the domain and range of the property. `Michael` is an instance of the `Person` class and `Buddy` is an instance of the `Pet` class. The `hasPet` property points from a `Person` to that person's `Pet`. The inverse property could be `isPetOf` which would be represented by a link between the two individuals going the other way, from `Buddy` to `Michael`. Whenever possible it is desirable to adhere to this type of naming standard with properties. Properties going in one direction as *hasProperty* and their inverses as *isPropertyOf*.

Exercise 10: Create some inverse properties

---

1. Use the Object properties tab to create a new object property called `isIngredientOf` (this will be the inverse property of `hasIngredient`). Make sure that `isIngredientOf` is a sibling property if `hasIngredient` and a sub-property of `owl:topObjectProperty.`

2. Click on the Add icon (+) next to Inverse Of in the Description view for `hasIngredient.` You will be presented with a window that shows a nested view of all the current properties. Select hasIngredient to make it the inverse of `isIngredientOf`.

3. Select isIngredientOf and then Tools>Create object property hierarchy. Enter isToppingOf then on a new line enter isBaseOf. As before, select Continue and leave the box for disjoint properties unchecked and select Finish. Repeat step 2 to make `isToppingOf` the inverse of `hasTopping` and `isBaseOf` the inverse of `hasBase`.

4. Synchronize the reasoner. Your window should now look like figure 4.12.

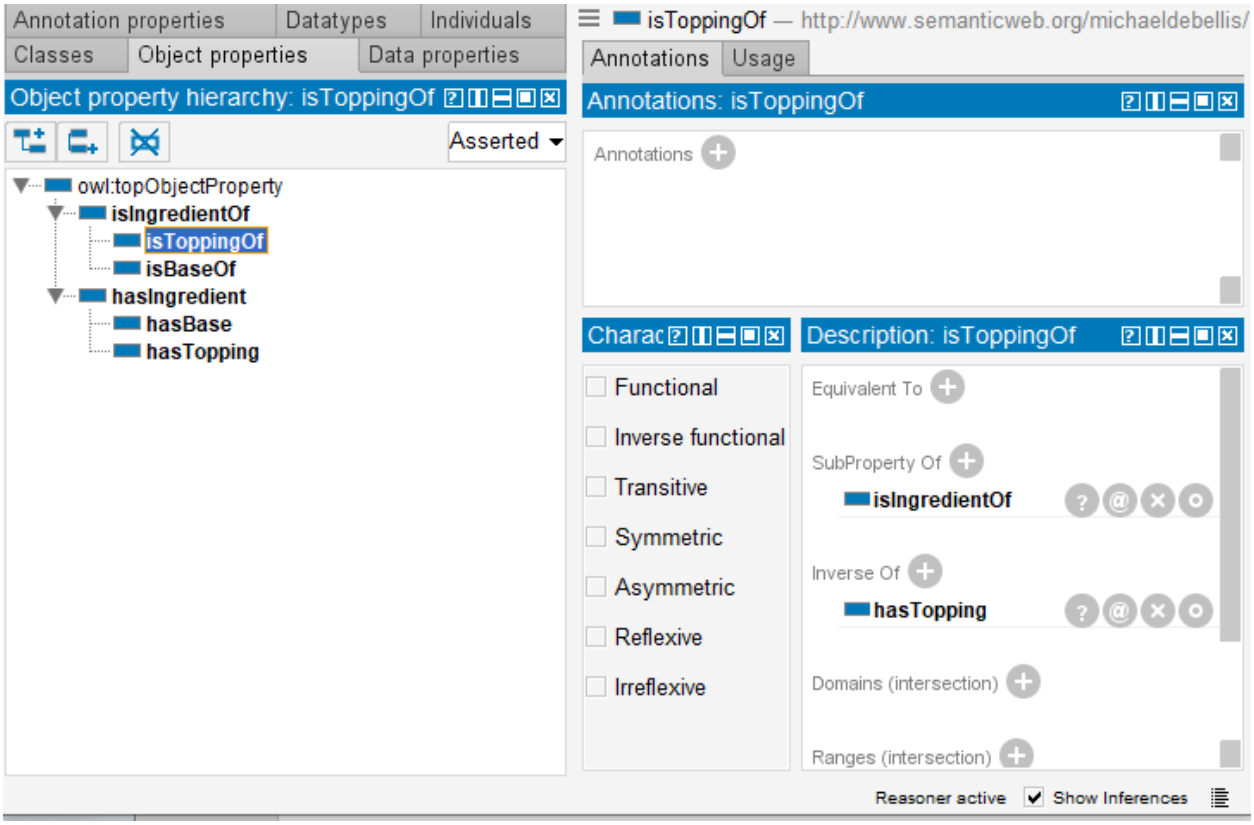


Figure 4.12 Inverse Properties

4.8 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. If you are familiar with basic concepts of relations in set theory these characteristics will already be familiar to you. In figure 4.12 you can see the Characteristics: view for a property as a list of check boxes: Functional, Inverse functional, Transitive, etc.

4.8.1 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. For example the property `hasBirthMother` -- someone 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. This can happen in OWL because

unlike many languages it does not have a unique names assumption. Unless specifically stated otherwise, the reasoner can infer that two individuals with different names are actually 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 the reasoner to infer that there was an inconsistency in the ontology. We will discuss names more in chapter 7.

In section 4.16 we will discuss cardinality restrictions on properties. E.g., that the `hasWheel` property of the `Bicycle` class has a minimum of 2 (allowing for training wheels) whereas `hasWheel` for the `Unicycle` class is defined to be exactly 1. A functional property is equivalent to a property with a cardinality restriction that says it has a maximum of 1 value. The term functional is from mathematics where a function is defined as a relation where each member of the domain has at most one value. For example, the `greaterThan` relation is not functional since for any number `X` many (in fact an infinite number) can be `greaterThan` X but the `plusOne` relation is functional since for any number `X` `plusOne` always results in one unique value.

4.8.2 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. Following our example from section 4.8.1 the inverse of `hasBirthMother` would be `isBirthMotherOf`. The `isBirthMotherOf` property would not be functional since a woman can be the birth mother of several children. However, it would be inverse functional since each person has exactly one mother.

4.8.3 Transitive Properties

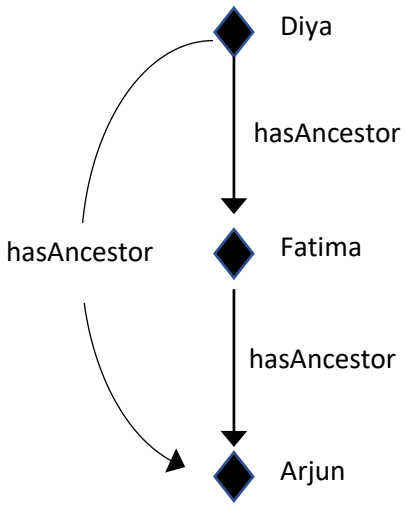


Figure 4.13 Transitive Properties

If a property P is transitive, and P 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 4.13 shows an example of the transitive property `hasAncestor`. If the individual `Diya` has an ancestor that is `Fatima`, and `Fatima` has an ancestor that is `Arjun`, then we can infer that `Diya` has an ancestor that is `Arjun` – this is indicated by the curved line in Figure 4.13.

An example of the transitive property in mathematics is the > relation. If x > y and y > z then x > z.

Note that if a property is transitive it cannot be functional. Also, if a property is transitive then its inverse property must also be transitive. E.g., the inverse of > is < and < is also transitive. We will see an example of this in chapter 6.

4.8.4 Symmetric and Asymmetric 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. The `hasSibling` property or `hasSpouse` are examples of

symmetric properties. If Michelle hasSpouse Barack, then Barack hasSpouse Michelle. A symmetric property is its own inverse.

An Asymmetric property is a property that can never have symmetric values. If a property P is asymmetric then if a is related to b via that property b cannot be related to a via that property. An example of an asymmetric property is hasBirthMother. If Diya hasBirthMother Fatima, then it can't be the case that Fatima hasBirthMother Diya.

4.8.5 Reflexive and Irreflexive Properties

A reflexive property is a property that always relates an individual to itself. If a property P is reflexive then for all individuals a P will always relate a to a. Equality is the most common example of a reflexive property. For any object a, a is always equal to a. An irreflexive property is… you guessed it… a property that can never relate an individual to itself. The property hasBirthMother is an example of an irreflexive property since no person can be their own mother. Note: you should use reflexive properties with care. The domain of a reflexive property is *always* owl:Thing. The reasons are complex, see the W3C Owl 2 Specification in the bibliography for more details. The important thing is that if you make a property reflexive that means its domain is owl:Thing. For example, if you have a reflexive property and declare its domain to be some class such as Person the reasoner will infer that Person is equivalent to owl:Thing which can cause problems.

4.8.6 Reasoners Automatically Enforce Property Characteristics

The reasoners that work with Protégé automatically enforce all the characteristics that are described above. For example, if the user enters the fact that Diya hasBirthMother Fatima and isBirthMotherOf is the inverse of hasBirthMother, the reasoner will infer that Fatima isBirthMotherOf Diya. These types of characteristics can significantly reduce the amount of effort needed to populate an ontology with data about individuals.

4.9 OWL Property Domains and Ranges

Properties may have a *domain* and *range* defined. These terms have the same meaning in OWL as they do in mathematics and set theory. The domain of a property is the set of all objects that can have that property asserted about it. The range is the set of all objects that can be the value of the property. Both the domain and range are optional. In general, it is a good idea to define them because doing so can catch modeling mistakes while defining the model rather than at run time when trying to use it. The domain for an object property must always be a class. For data properties the range is a simple datatype such as xsd:decimal. The most common predefined datatypes already exist in Protégé. It is also possible to define new data types although most users will seldom need to do that. For most cases if you are considering defining a new datatype you should probably consider making the property an object property instead and defining a class as the range. For people familiar with Entity-Relation modeling an object property is similar to a relation and a data property is similar to an attribute. For those familiar with set theory a property is identical to a binary relation in set theory.

As an example, in our pizza ontology, the property hasTopping would link individuals belonging to the class Pizza to individuals belonging to the class PizzaTopping. The domain of hasTopping is Pizza and the range is PizzaTopping. Inverse properties have their domains and range swapped. In this example, the inverse of hasTopping will be called isToppingOf. Thus, the domain for isToppingOf is the range of hasTopping (PizzaTopping) and the range for isToppingOf is the domain of hasTopping (Pizza).

Exercise 11: Define the domain and range of the hasTopping property

---

1. Navigate to the Object properties tab. Select the hasTopping property.

2. Click on the Add icon (+) next to Domains (intersection) in the Description view for `hasTopping`. You will be presented with a window that shows several tabs. There are multiple ways to define domain and range. For now we will use the simplest method (and the one most often used). Select the ClassHierarchy tab. Then select Pizza from the class hierarchy. Your UI should look like figure 4.14. Click on OK. You should now see Pizza underneath the Domains in the Description view.

3. Repeat step 2 but this time start by using the (+) icon next to the Ranges (intersection) in the Description for `hasTopping`. This time select the class PizzaTopping as the range.

4. Synchronize the reasoner. Now select isToppingOf. You should see that the Domain and Range for `isToppingOf` have been filled in by the reasoner (see figure 4.15). Since the two properties are inverses the reasoner knows that the domain for one is the range for the other and vice versa. This is another example of why frequently running the reasoner can save time and help maintain a valid model. Note that these values are highlighted in yellow. Any information supplied by the reasoner rather than by the user is highlighted in this way.

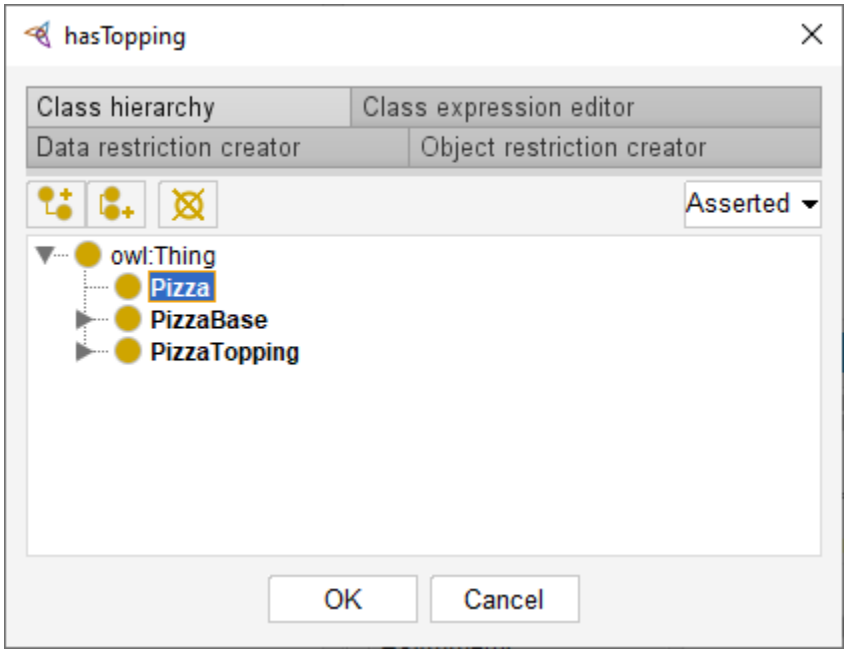


Figure 4.14 Defining the Domain for hasTopping

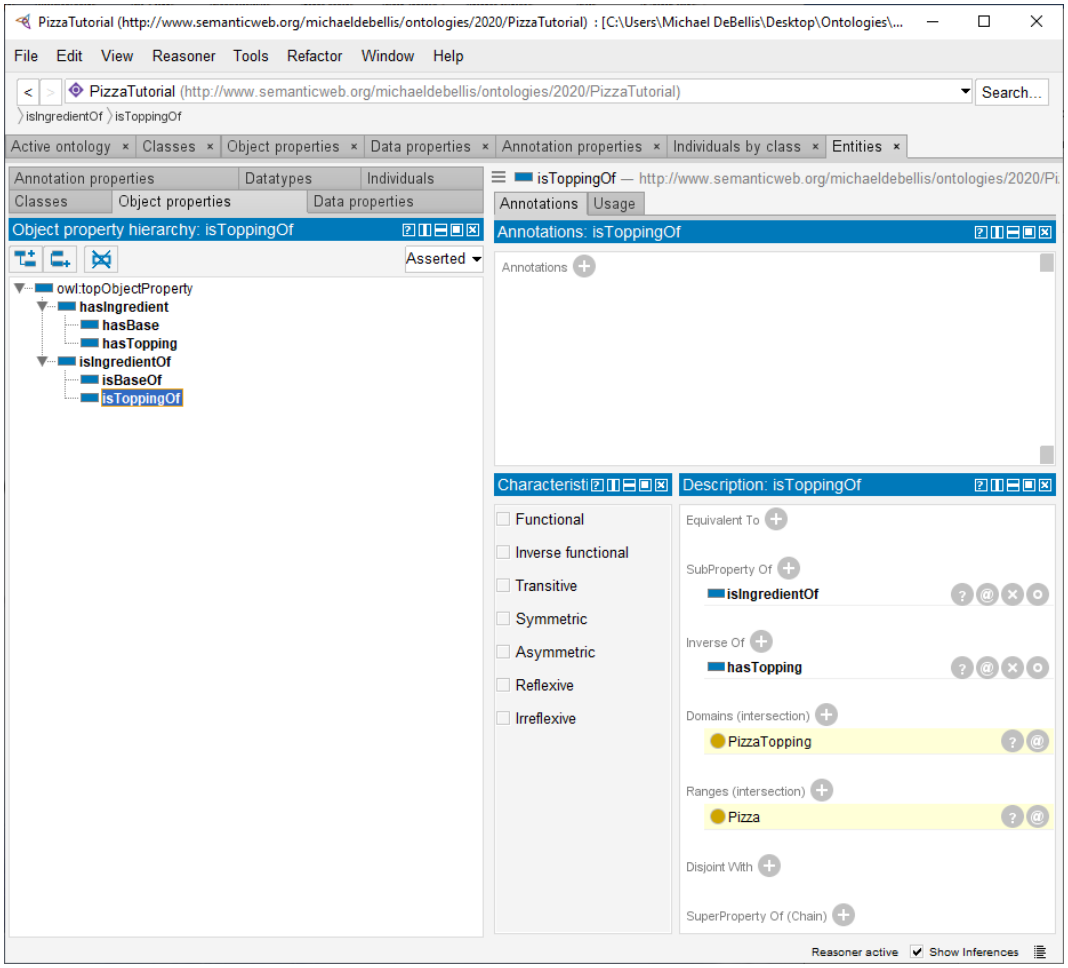


Figure 4.15 Domain and Range inferred by the reasoner

It is possible to specify more than one class as the domain or range of a property. One of the most common mistakes of new users is to do this and expect that the resulting domain/range is the union of the two classes. However, note that next to the Domain and Range in the Description view it says (intersection). This is because the semantics of having 2 or more classes as the domain or range is the *intersection* of those classes *not* the union. E.g., if one defined the domain for a property to be `Pizza` and then added another domain `IceCream` that would mean that for something to be in the domain of that property it would have to be an instance of *both* `Pizza` *and* `IceCream` not (as people often expect) the *union* of those two sets which would be *either* the class `Pizza` *or* the class `IceCream`. Also, note that the domain and range are for inferencing, they are not data integrity constraints. This distinction will be explained in more detail below in the section on SHACL.

Exercise 12: Define the domain and range for the hasBase property

---

1. Now we are going to repeat the same activities as in the previous exercise but for another property: `hasBase`. Make sure you are still on the Object properties tab. Select the hasBase property.

2. Click on the Add icon (+) next to Domains (intersection) in the Description view for `hasBase.` Select the ClassHierarchy tab. Then select Pizza from the class hierarchy..

3. Repeat step 2 but this time start by using the (+) icon next to the Ranges (intersection) in the Description for `hasBase`. This time select the class PizzaBase as the range.

4. Synchronize the reasoner. Now select isBaseOf You should see that the Domain and Range for `isBaseOf` have been filled in by the reasoner.

4.10 Describing and Defining Classes

Now that we have defined some properties, we can use these properties to define some more interesting classes. There are 3 types of classes in OWL:

1. Primitive classes. These are classes that are defined by conditions that are *necessary* (but not sufficient) to hold for any individuals that are instances of that class or its subclasses. The condition may be as simple as: *Class A is a subclass of class B*. To start with we will define primitive classes first and then defined classes. When the reasoner encounters an individual that is an instance of a primitive class it infers that all the conditions defined for that class must hold for that individual.
2. Defined classes. These are classes that are defined by both *necessary* and *sufficient* conditions. When the reasoner encounters an individual that satisfies all the conditions for a defined class it will make the inference that the individual is an instance of that class. The reasoner can also use the conditions defined on classes to change the class hierarchy, e.g., to infer that *Class A is a subclass of Class B*. We will see examples of this later in the tutorial.
3. Anonymous classes. These are classes that you won't encounter much and that won't be discussed much in this tutorial, but it is good to know about them. They are created by the reasoner when you use class expressions. For example, if you define the range of a property to be `PizzaTopping or PizzaBase` then the reasoner will create an anonymous class representing the intersection of those two classes.

4.10.1 Property restrictions

In OWL properties define binary relations with the same semantics and characteristics as binary relations in First Order Logic. There are two types of OWL properties for describing a domain: Object properties and Data properties. Object properties have classes as their domain and range. Data properties have classes as their domain and simple datatypes such as `xsd:string` or `xsd:dateTime` as their range. In figure 3.3 the individual `Michael` is related to the individual `USA` by the property `livesIn`. Consider all the individuals who are an instance of `Person` and also have the same relation, that each `livesIn` the `USA`. This group is a set or OWL class such as `USAResidents`. In OWL a class can be defined by describing the various properties and values that hold for all individuals in the class. Such definitions are called *restrictions* in OWL.

The following are some examples of classes of individuals that we might want to define via property restrictions:

- The class of individuals with at least one `hasChild` relation.
- The class of individuals with 2 or more `hasChild` relations.
- 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 a mozzarella topping.
- The class of individuals that are `Pizzas` and only have `hasTopping` relations to instances of the class `VegetableTopping` (i.e., `VegetarianPizza`).

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

1. Quantifier restrictions. These describe that a property must have some or all values that are of a particular class.
2. Cardinality restrictions. These describe the number of individuals that must be related to a class by a specific property.
3. hasValue restrictions. These describe specific values that a property must have.

We will initially use quantifier restrictions. Quantifier restrictions can be further categorized as *existential* restrictions and *universal* restrictions[6]. Both types of restrictions will be illustrated with examples in this tutorial.

- Existential restrictions describe classes of individuals that participate in at least one relation along a specified property. For example, the class of individuals who have at least one (or some) `hasTopping` relation to instances of `VegetableTopping`. In OWL the keyword `some` is used to denote existential restrictions.
- Universal restrictions describe classes of individuals that for a given property *only* have relations along a property to individuals that are members of a specific class. For example, the class of individuals that only have `hasTopping` relations to instances of the class `VegetableTopping`. In OWL they keyword `only` is used for universal restrictions.

Let's take a closer look at an example of an existential restriction. The restriction `hasTopping some MozzarellaTopping` is an existential restriction (as indicated by the `some` keyword), which restricts 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`.

A restriction always describes a class. Sometimes (as we will soon see) it can be a defined class. Other times it may be an anonymous class. In all cases the class contains all of the individuals that satisfy the restriction, i.e., all of the individuals that have the relationships required to be a member of the class. In section 9.2 one of our SPARQL queries will return several anonymous classes.

[6] These have the same meaning as existential and universal quantification in First Order Logic.

The restrictions for a class are displayed and edited using the Class Description View shown in Figure 4.17. The Class Description View holds most of the information used to describe a class. The Class Description View is a powerful way of describing and defining classes. It is one of the most important differences between describing classes in OWL and in other models such as most object-oriented programming languages. In other models there is no formal definition that describes why one class is a subclass of another, in OWL there is. Indeed, the OWL classifier can actually redefine the class hierarchy based on the logical restrictions defined by the user. We will see an example of this later in the tutorial.

Restrictions are also called axioms in OWL. This has the same meaning as in logic. An axiom is a logical formula defined by the user rather than deduced by the reasoner. As described above, in Protégé all axioms are shown in normal font whereas all inferences inferred by the reasoner are highlighted in yellow.

4.10.2 Existential Restrictions

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 or datatype. 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 13: 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. Click on the Add icon (+) next to the SubClass Of field in the Description view for Pizza.

3. This will bring up a new window with several tab options to define a new restriction. Select the Object restriction creator. This tab has the Restricted property on the left and the Restriction filler on the right.

4. Expand the property hierarchy on the left and select hasBase as the property to restrict. Then in the Restriction filler on the right select the class PizzaBase. Finally, the Restriction type at the bottom should be set to Some (existential). This should be the default so you shouldn't have to change anything but double check that this is the case. Your window should look like figure 4.16 now.

5. When your UI looks like figure 4.16 click on the OK button. That should close the window. Run the reasoner to make sure things are consistent. Your main window should now look like figure 4.17.
