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

---

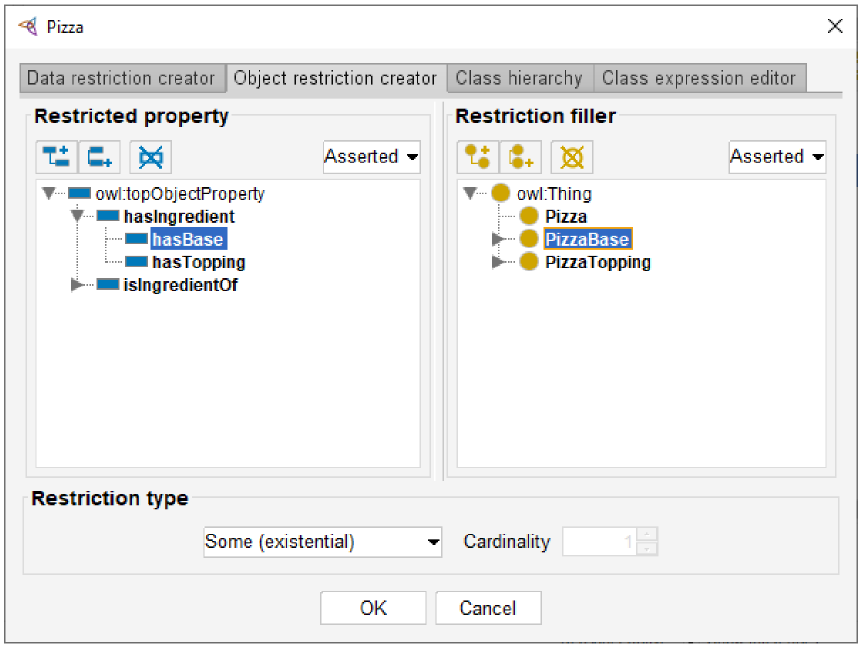
1. Select Pizza from the class hierarchy on the Classes tab.

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.



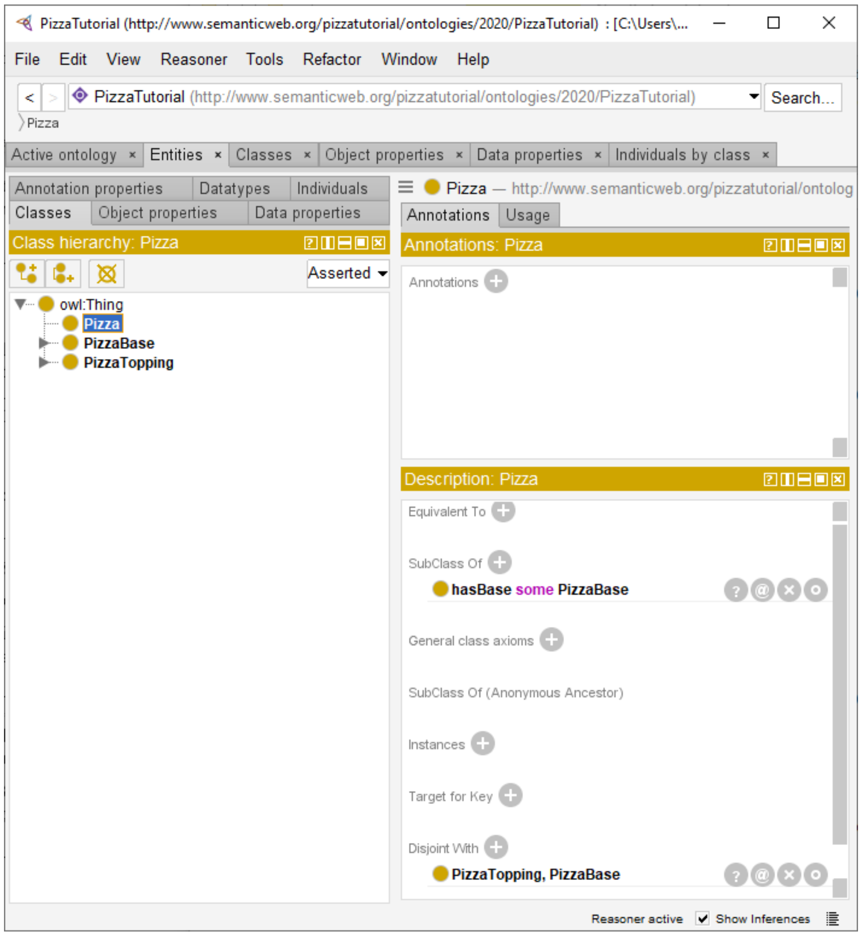
Restricted property: owl:topObjectProperty, hasIngredient, hasBase, hasTopping, isIngredientOf (Asserted)

Restriction filler: owl:Thing, Pizza, PizzaBase, PizzaTopping (Asserted)

Restriction type: Some (existential)

Cardinality: 1

Figure 4.16 The Object Restriction Creator Tab



SubClass Of: hasBase some PizzaBase

Disjoint With: PizzaTopping, PizzaBase

Figure 4.17 The Pizza Class with hasBase Restriction

We have described the class Pizza to be to be a subclass of Thing and a subclass of the things that have a base which is some kind of PizzaBase. Notice that these are necessary conditions — if something is a Pizza it is *necessary* for it to be a member of the class Thing (in OWL, everything is a member of the class Thing) and *necessary* for it to have a kind of PizzaBase. More formally, for something to be a Pizza it is necessary for it to be in a relationship with an individual that is a member of the class PizzaBase via the property hasBase.

4.10.3 Creating Subclasses of Pizza

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 14: Create Subclasses of Pizza: NamedPizza and MargheritaPizza

---

1. Select Pizza from the class hierarchy on the Classes tab.

2. Click on the Add subclass icon at the top left of the Classes tab (look back at figure 4.4 if you aren't certain). You can also move your mouse over the icons and you will see a little pop-up hint for each icon.

3. Protégé will prompt you for the name of the new subclass. Call it NamedPizza.

4. Repeat steps 1-3 this time starting with NamedPizza to create a subclass of NamedPizza. Call it MargheritaPizza.

5. Add a comment to the class MargheritaPizza using the Annotations view. This is above the Description view. Add the comment: A pizza that only has Mozzarella and Tomato toppings. Remember that annotation properties are meta-data that can be asserted about any entity whereas object and data properties can only be asserted about individuals. There are a few predefined annotation properties that are included in all Protégé ontologies such as the comment property.

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 15: Create Restrictions that define a MargheritaPizza

---

1. Select MargheritaPizza 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.

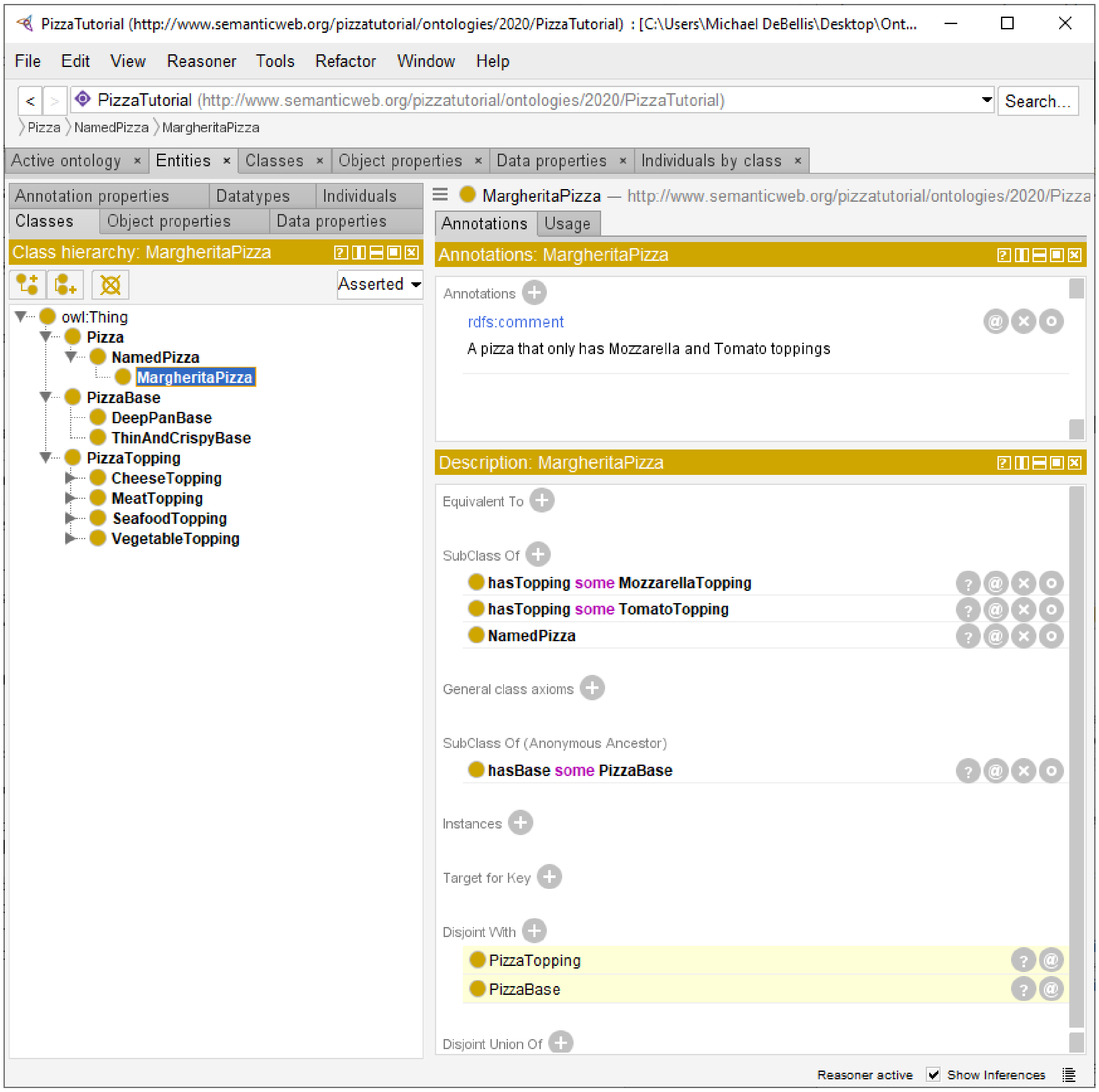
3. This again brings up the restriction dialogue. This time rather than use the Object restriction creator we will use the Class expression editor tab. Select that tab.

4. Type hasTopping some Mo into the field. Rather than type the rest of the name of the topping now hit <control><space> (hold down the control key and hit the space bar). Protégé should auto-complete the name for you and the field should now contain: hasTopping some MozzarellaTopping. This is a useful technique for any part of the Protégé UI. Whenever you enter the name of some entity you can do

<control><space>. If there is only one possible completion for the string then Protégé will fill in the appropriate name. If there are multiple possible completions Protégé will create a menu with all the possible completions and allow you to select the one you want.

5. Click on OK to enter the new restriction.

6. Repeat steps 1-5 only this time add the restriction hasTopping some TomatoTopping. Remember to use <control><space> to save time typing. Synchronize the reasoner to make sure things are consistent. Your UI should now look similar to figure 4.18.


PizzaTutorial (http://www.semanticweb.org/pizzatutorial/ontologies/2020/PizzaTutorial) : [C:\Users\Michael DeBellis\Desktop\Ont...

File Edit View Reasoner Tools Refactor Window Help

PizzaTutorial (http://www.semanticweb.org/pizzatutorial/ontologies/2020/PizzaTutorial)

Pizza 〉 NamedPizza 〉 MargheritaPizza

Active ontology | Entities | Classes | Object properties | Data properties | Individuals by class

Annotation properties | Datatypes | Individuals | Classes | Object properties | Data properties

Class hierarchy: MargheritaPizza (Asserted)

```
owl:Thing
  Pizza
    NamedPizza
      MargheritaPizza
  PizzaBase
    DeepPanBase
    ThinAndCrispyBase
  PizzaTopping
    CheeseTopping
    MeatTopping
    SeafoodTopping
    VegetableTopping
```

MargheritaPizza — http://www.semanticweb.org/pizzatutorial/ontologies/2020/Pizza

Annotations | Usage

Annotations: MargheritaPizza

Annotations
rdfs:comment
A pizza that only has Mozzarella and Tomato toppings

Description: MargheritaPizza

Equivalent To

SubClass Of
- hasTopping some MozzarellaTopping
- hasTopping some TomatoTopping
- NamedPizza

General class axioms

SubClass Of (Anonymous Ancestor)
- hasBase some PizzaBase

Instances

Target for Key

Disjoint With
- PizzaTopping
- PizzaBase

Disjoint Union Of

Reasoner active ✓ Show Inferences

Figure 4.18 Definition for the class MargheritaPizza

Note in figure 4.18 the two classes listed under Disjoint With and highlighted in yellow. This is an example of an inference from the reasoner. When we defined Pizza, PizzaBase, and PizzaTopping we made those 3 classes disjoint. I.e., no individual can be a member of more than one of those classes. Since MargheritaPizza is a subclass of Pizza it is also disjoint with PizzaBase and PizzaTopping, so the reasoner has added this information to the definition of MargheritaPizza and as with all inferences from the reasoner highlighted the new information in yellow.

We will now create the class to represent an AmericanaPizza, which has toppings of pepperoni, mozzarella and tomato. Because the class AmericanaPizza is similar to the class MargheritaPizza (i.e., an AmericanaPizza is almost the same as a MargheritaPizza 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 16: Create AmericanaPizza by Cloning MargheritaPizza and Adding Additional Restrictions

---

1. Select MargheritaPizza from the class hierarchy on the Classes tab.

2. Select Edit>Duplicate selected class. This will bring up a dialogue for you to duplicate the class. The default is the name of the existing class so there will be a red error message when you start because you need to enter a new name. Change the name from MargheritaPizza to AmericanaPizza. Leave all the other options as they are and then select OK.

3. Make sure that AmericanaPizza is still selected. Click on the Add icon (+) next to the SubClass Of field in the Description view for AmericanaPizza.

4. Use either the Object restriction creator tab or the Class expression editor tab to add the additional restriction: hasTopping some PepperoniTopping.

5. Click on OK to enter the new restriction.

6. Edit the comment annotation on AmericanaPizza. It should currently be: A pizza that only has Mozzarella and Tomato toppings since it was copied over from MargheritaPizza. Note that at the top right of the comment there are three little icons, an @ sign, an X and an O. Click on the O. This icon is the one you use to edit any existing data in Protégé. This should bring up a window where you can edit the comment. Change it to something appropriate such as: A pizza that only has Mozzarella, Tomato, and Pepperoni toppings. Then click on OK to enter the edit to the comment.

Exercise 17: Create AmericanaHotPizza and SohoPizza

---

1. An AmericanaHotPizza 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.

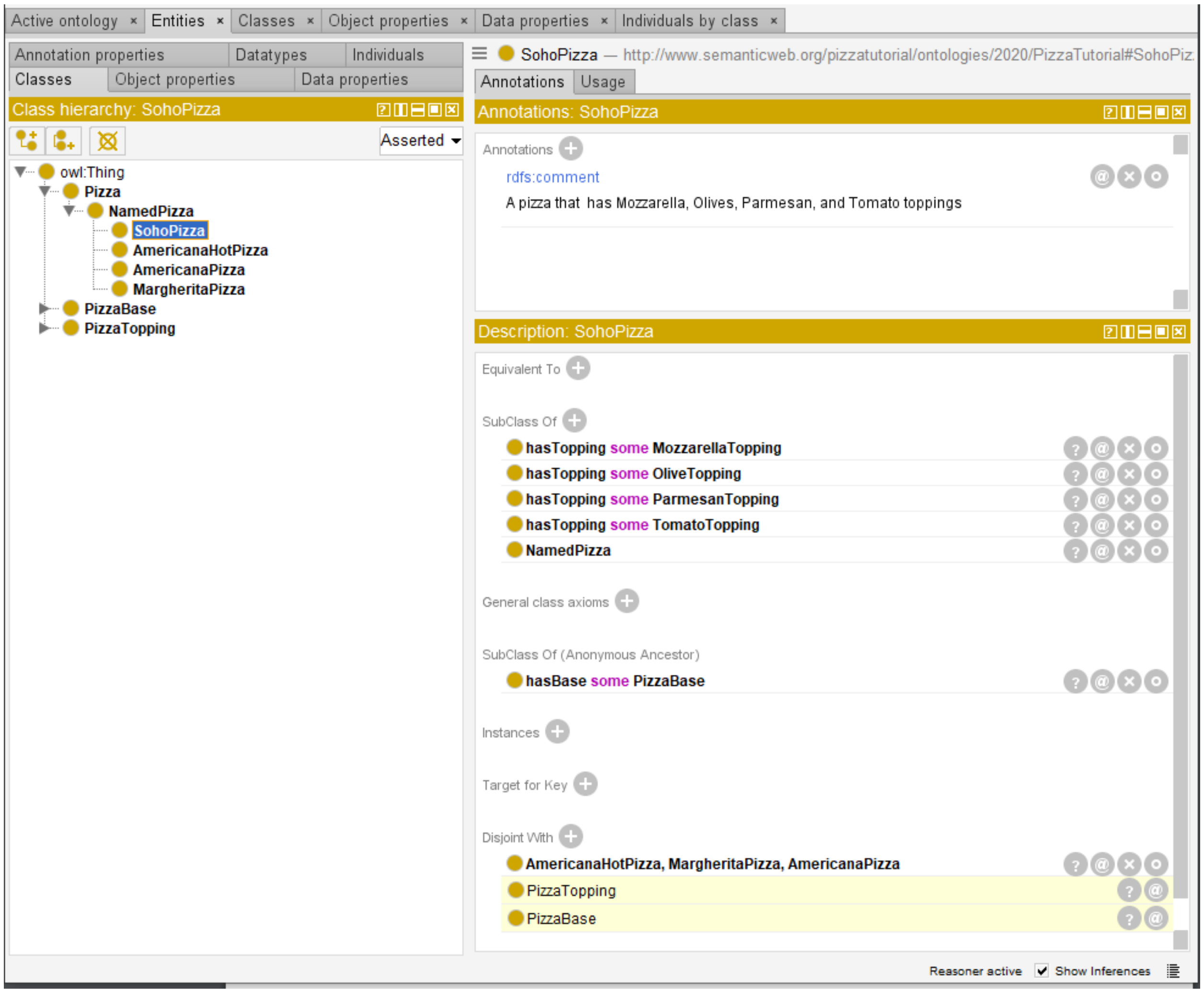
2. A SohoPizza is almost the same as a MargheritaPizza but has additional toppings of olives and parmesan cheese — create this by cloning MargheritaPizza and adding two existential restrictions along the property hasTopping, one with a filler of OliveTopping, and one with a filler of ParmesanTopping.

Exercise 18: Make Subclasses of NamedPizza Disjoint

---

1. We want to make these subclasses of NamedPizza disjoint from each other. I.e., any individual can belong to at most one of these classes. To do that first select MargheritaPizza (or any other subclass of NamedPizza).

2. Click on the (+) sign next to Disjoint With near the bottom of the Description view. This will bring up a Class hierarchy view. Use this to navigate to the subclasses of NamedPizza and use <control><left click> to select all of the other sibling classes to the one you selected. Then select OK. You should now see the appropriate disjoint axioms showing up on each subclass of NamedPizza. Synchronize the reasoner. Your UI should look similar to figure 4.19 now.



A pizza that has Mozzarella, Olives, Parmesan, and Tomato toppings

hasTopping some MozzarellaTopping
hasTopping some OliveTopping
hasTopping some ParmesanTopping
hasTopping some TomatoTopping
NamedPizza

hasBase some PizzaBase

AmericanaHotPizza, MargheritaPizza, AmericanaPizza
PizzaTopping
PizzaBase

Figure 4.19 Subclasses of NamedPizza are Disjoint

4.10.4 Detecting a Class that can't Have Members

Next, we are going to use the reasoner to detect a class with a definition that means it can never have any members. In the current version of Protégé when the reasoner detects an inconsistency or problem on some operating systems the UI can occasionally lock up and be hard to use. So to make sure you don't lose any of your work save your ontology using File>Save.

Sometimes it can be useful to create a class that we think should be impossible to instantiate to make sure the ontology is modeled as we think it is. Such a class is called a Probe Class.

Exercise 19: Add a Probe Class called ProbeInconsistentTopping

---

1. Select the class CheeseTopping from the class hierarchy.

2. Create a subclass of CheeseTopping called ProbeInconsistentTopping.

3. Click on the Add icon (+) next to the SubClass Of field in the Description view for `ProbeInconsistentTopping`.

4. Select the Class hierarchy tab from the dialogue that pops up. This will bring up a small view that looks like the class hierarchy tab you have been using to add new classes. Use this to navigate to and select the class VegetableTopping. Click on OK.

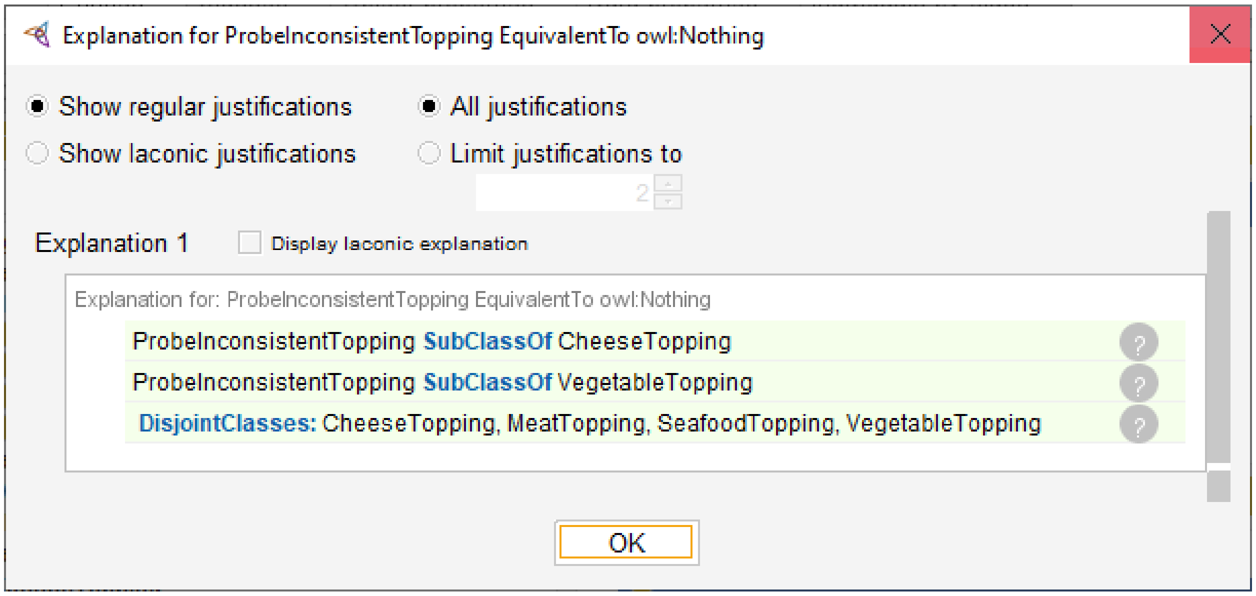
5. Make sure to save your current ontology file. Now run the reasoner. You should see that `ProbeInconsistentTopping` is now highlighted in red indicating it is inconsistent.

6. Click on ProbeInconsistentTopping to see why it is highlighted in red. Notice that at the top of the Description view you should now see owl:Nothing under the Equivalent To field. This means that the probe class is equivalent to `owl:Nothing`. The `owl:Nothing` class is the opposite of `owl:Thing`. Whereas all individuals are instances of `owl:Thing`, no individual can ever be an instance of `owl:Nothing`. The `owl:Nothing` class is equivalent to the empty set in set theory.

7. There should be a ? icon just to the right of owl:Nothing. As with any inference of the reasoner it is possible to click on the new information and generate an explanation for it. Do that now, click on the ? icon. This should generate a new window that looks like figure 4.20. The explanation is that `ProbeInconsistentTopping` is a subclass of `CheeseTopping` and `VegetableTopping` but those two classes are disjoint.

8. Click OK to dismiss the window. Delete the class ProbeInconsistentTopping by selecting it and then clicking on the delete class icon at the top of the classes view (see figure 4.4).

9. Synchronize the reasoner.


Explanation for ProbeInconsistentTopping EquivalentTo owl:Nothing

● Show regular justifications ● All justifications

○ Show laconic justifications ○ Limit justifications to 2

Explanation 1 ☐ Display laconic explanation

Explanation for: ProbeInconsistentTopping EquivalentTo owl:Nothing

ProbeInconsistentTopping SubClassOf CheeseTopping

ProbeInconsistentTopping SubClassOf VegetableTopping

DisjointClasses: CheeseTopping, MeatTopping, SeafoodTopping, VegetableTopping

Figure 4.20 Explanation for why `ProbeInconsistentTopping` is equivalent to `owl:Nothing`

4.11 Primitive and Defined Classes (Necessary and Sufficient Axioms)

All of the classes that we have created so far have only used necessary axioms to describe them. Necessary axioms can be read as, *If something is a member of this class then it is necessary to fulfil these conditions*. With necessary axioms 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 20: Create the CheesyPizza class

---

1. Select Pizza in the class hierarchy on the Classes tab.

2. Select the Add Subclass icon (see figure 4.4). Name the new subclass CheesyPizza.

3. Make sure CheesyPizza is selected. Click on the Add icon (+) next to the SubClass Of field in the Description view.

4. Select the Class expression editor tab. Type in the new axiom: hasTopping some CheeseTopping. Remember you can use <control><space> to auto-complete each word in the axiom, e.g., type hasT and then <control><space> to auto-complete the rest. If you haven't typed enough for Protégé to unambiguously choose one entity or Description Logic keyword you will be prompted with a menu of possible completions. Click OK to enter the new restriction axiom.

⚠ Note that if you just type a few characters, the number of possible completions may be large resulting in an unwieldy menu. Also, Protégé doesn't do things like type checking on possible completions. For example, if you type "Chee" and do <control><space> you will be prompted with `CheeseTopping` and `CheesyPizza` as possible completions even though a `Pizza` is not in the range of `hasTopping`. This is where the reasoner can also help. If you enter a class that is not in the range of `hasTopping` the reasoner will signal an inconsistency.

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`. 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`.

A class (such as all the classes we have defined so far) that only has necessary conditions is called a *primitive class*. A class that has necessary and sufficient conditions is known as a *defined class*. 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 To header. This can be done with the menu option: Edit>Convert to defined class.

Exercise 21: Convert CheesyPizza from a Primitive Class to a Defined Class

---

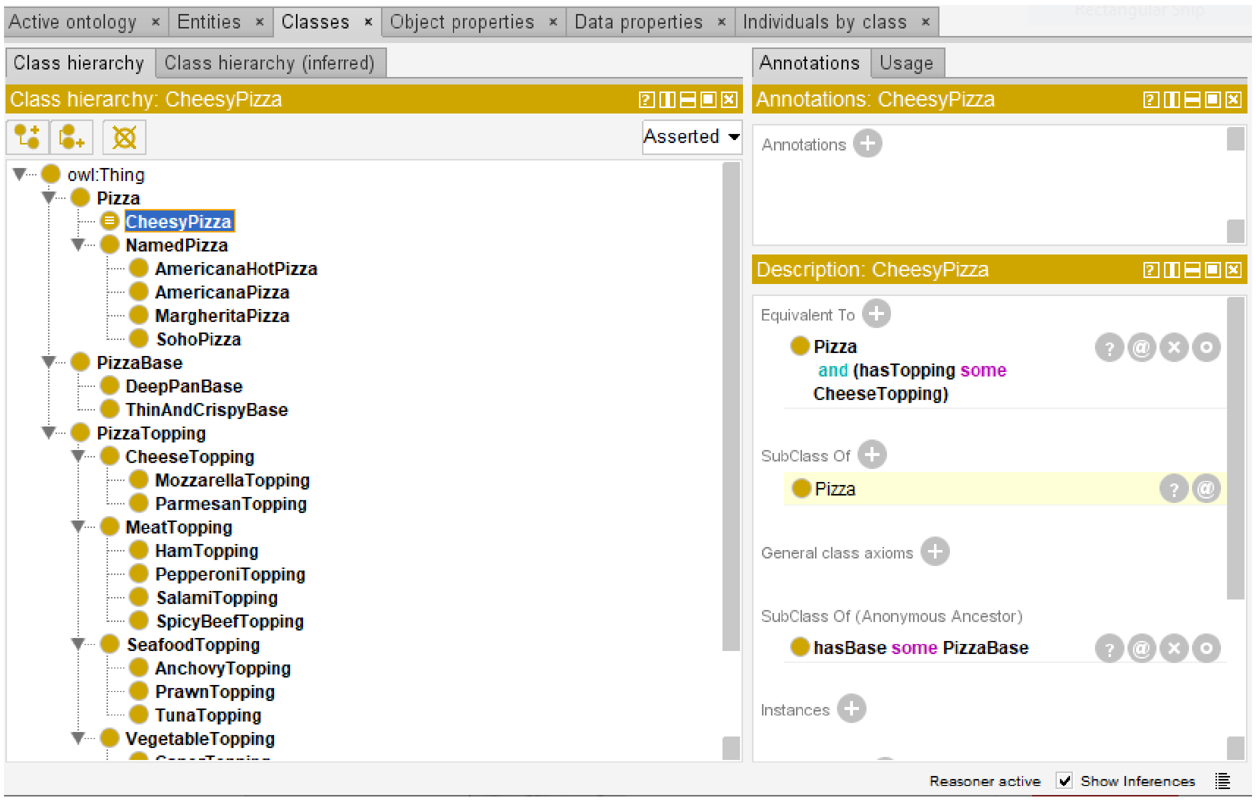
1. Make sure CheesyPizza is selected.

2. Select the menu option: Edit>Convert to defined class.

3. Synchronize the reasoner.

Your screen should now look similar to figure 4.21. Note that when a class is a defined class it is shown in the UI with three horizontal stripes in the circle next to its name.

So far we have seen the reasoner do simple things such as propagate disjoint axioms from super classes down to subclasses. However, the reasoner is capable of doing much more. Now that we have a defined class we can see an example of this. Notice that there are two tabs in the Class hierarchy view. The one shown in figure 4.21 is the asserted hierarchy. This is the hierarchy as defined by user declared axioms. The other tab is the Class hierarchy (inferred) tab. This is the hierarchy as inferred by the reasoner. Up until we created a defined class the two tabs would be identical because we had only primitive classes in the ontology. Now that we have a defined class the inferred hierarchy will look different. Select the Class hierarchy (inferred) tab. Make sure that the reasoner is synchronized (it should say Reasoner active as in figure 4.21). Also, make sure to expand the `CheesyPizza` class in this tab. You should see a screen similar to figure 4.22. As you should see in the inferred tab the reasoner has inferred that all the `Pizza` classes with a cheese topping are subclasses of `CheesyPizza`.



Class hierarchy: CheesyPizza — Asserted

```
owl:Thing
  Pizza
    CheesyPizza
    NamedPizza
      AmericanaHotPizza
      AmericanaPizza
      MargheritaPizza
      SohoPizza
  PizzaBase
    DeepPanBase
    ThinAndCrispyBase
  PizzaTopping
    CheeseTopping
      MozzarellaTopping
      ParmesanTopping
    MeatTopping
      HamTopping
      PepperoniTopping
      SalamiTopping
      SpicyBeefTopping
    SeafoodTopping
      AnchovyTopping
      PrawnTopping
      TunaTopping
    VegetableTopping
```

Annotations: CheesyPizza

Annotations

Description: CheesyPizza

Equivalent To
Pizza
and (hasTopping some
CheeseTopping)

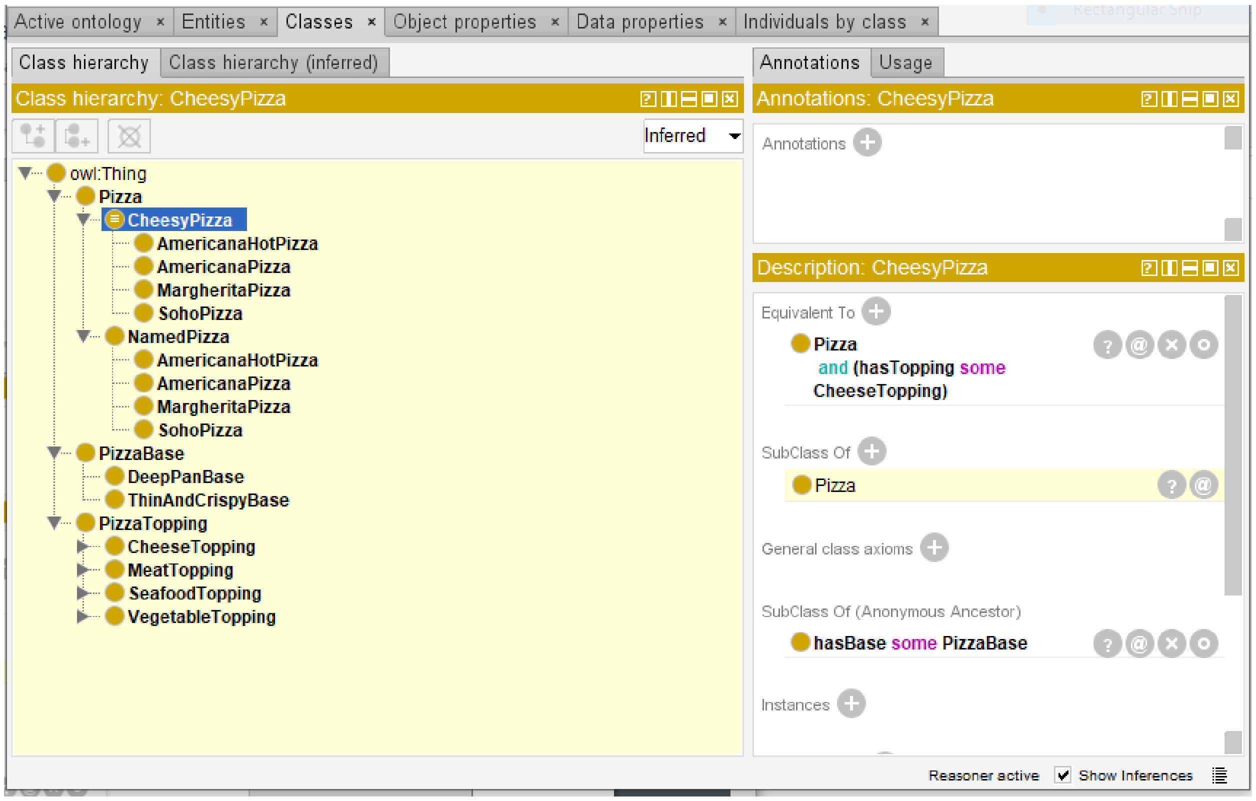
SubClass Of
Pizza

General class axioms

SubClass Of (Anonymous Ancestor)
hasBase some PizzaBase

Instances

Figure 4.21 CheesyPizza as a Defined Class



Class hierarchy: CheesyPizza — Inferred

```
owl:Thing
  Pizza
    CheesyPizza
      AmericanaHotPizza
      AmericanaPizza
      MargheritaPizza
      SohoPizza
    NamedPizza
      AmericanaHotPizza
      AmericanaPizza
      MargheritaPizza
      SohoPizza
  PizzaBase
    DeepPanBase
    ThinAndCrispyBase
  PizzaTopping
    CheeseTopping
    MeatTopping
    SeafoodTopping
    VegetableTopping
```

Annotations: CheesyPizza

Annotations

Description: CheesyPizza

Equivalent To
Pizza
and (hasTopping some
CheeseTopping)

SubClass Of
Pizza

General class axioms

SubClass Of (Anonymous Ancestor)
hasBase some PizzaBase

Instances

Figure 4.22 Classes Inferred by the Reasoner to be subclasses of CheesyPizza

4.12 Universal Restrictions

All of the restrictions we have created so far have been existential restrictions (defined using the `some` DL keyword). 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 correspond to the symbol ∀ in First Order Logic. They constrain the relationships along a given property to individuals that are members of a specific class. For example, the universal restriction `∀ hasTopping VegetableTopping` describes the individuals all of whose `hasTopping` relationships are to members of the class `VegetableTopping` — the individuals do not have a `hasTopping` relationship to individuals that aren't members of the class `VegetableTopping`.

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

Exercise 22: Create a Defined Class called VegetarianPizza

---

1. Select the Pizza in the Classes tab. Create a subclass of `Pizza` and name it `VegetarianPizza`.

2. Make sure VegetarianPizza is selected. Click on the Add icon (+) next to the SubClass Of field in the Description view.

3. Select the Class expression editor tab from the pop-up window. Type in the Description Logic axiom: hasTopping only (VegetableTopping or CheeseTopping). Click on OK.

4. Make sure VegetarianPizza is still selected. Run the Edit>Convert to defined class command.

5. `VegetarianPizza` should now have three horizontal lines through it just as `CheesyPizza` does. Also, the Equivalent To field in the Description view should have: Pizza and (hasTopping only (CheeseTopping or VegetableTopping)). Note that another way to create defined classes is to enter the Description Logic axiom directly into the Equivalent To field.

6. Synchronize the reasoner.

This means that if something is a member of the class `VegetarianPizza` it is necessary for it to be a kind of `Pizza` and it is necessary for it to only (∀ universal quantifier) have toppings that are kinds of `CheeseTopping` or kinds of `VegetableTopping`. In other words, all `hasTopping` relationships that individuals which are members of the class `VegetarianPizza` participate in must be to individuals that are either members of the class `CheeseTopping` or `VegetableTopping`. The class `VegetarianPizza` also contains individuals that are Pizzas and do not participate in any `hasTopping` relationships.

In situations like the above example, a common mistake is to use an intersection instead of a union. For example, CheeseTopping *and* VegetableTopping. Although CheeseTopping and Vegetable might be a natural thing to say in English, this logically means something that is simultaneously a kind of CheeseTopping and VegetableTopping. This is incorrect because we have stated that CheeseTopping and VegetableTopping are disjoint classes and hence no individual can be an instance of both. If we used such a definition the reasoner would detect the inconsistency.

In the above example it might have been tempting to create two universal restrictions — one for CheeseTopping (∀ hasTopping CheeseTopping) and one for VegetableTopping (∀ hasTopping VegetableTopping). However, when multiple restrictions are used (for any type of restriction) the total description is taken to be the intersection of the individual restrictions. This would have therefore been equivalent to one restriction with a filler that is the intersection of MozzarellaTopping *and* TomatoTopping — as explained above this would have been logically incorrect.

4.13 Automated Classification and Open World Reasoning

Make sure that the reasoner is synchronized (the little text in the lower right corner should say Reasoner active). Now switch from the Class hierarchy tab to the Class hierarchy (inferred) tab. You may notice something that seems perplexing. The classes MargheritaPizza and SohoPizza both only have vegetable and cheese toppings. So one might expect that the reasoner would classify them as subclasses of VegetarianPizza as it recently (in section 4.11) classified them and others as subclasses of CheesyPizza. The reason this didn't happen is something called the Open World Assumption (OWA). This is one of the concepts of OWL that can be most confusing for new and even experienced users because it is different than the Close World Assumption (CWA) used in most other programming and knowledge representation languages.

In most languages using the CWA we assume that everything that is currently known about the system is already in the database. However, OWL was meant to be a language to bring semantics to the Internet so the language designers chose the OWA. The open world assumption means that we cannot assume something doesn't exist just because it isn't currently in the ontology. The Internet is an open system. The information could be out there in some data source that hasn't yet been integrated into our ontology. Thus, we can't conclude some information doesn't exist unless 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 TomatoTopping and only kinds of MozzarellaTopping or TomatoTopping, we must add what is known as a closure axiom on the hasTopping property.

A closure axiom on a property consists of a universal restriction that says that a property can only be filled by 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. `hasTopping only (MozzarellaTopping or TomatoTopping)`.

Exercise 23: Add a Closure Axiom on the hasTopping Property for MargheritaPizza

---

1. Make sure that MargheritaPizza is selected in the class hierarchy in the Classes tab.

2. Click on the Add icon (+) next to the SubClass Of field in the Description view.

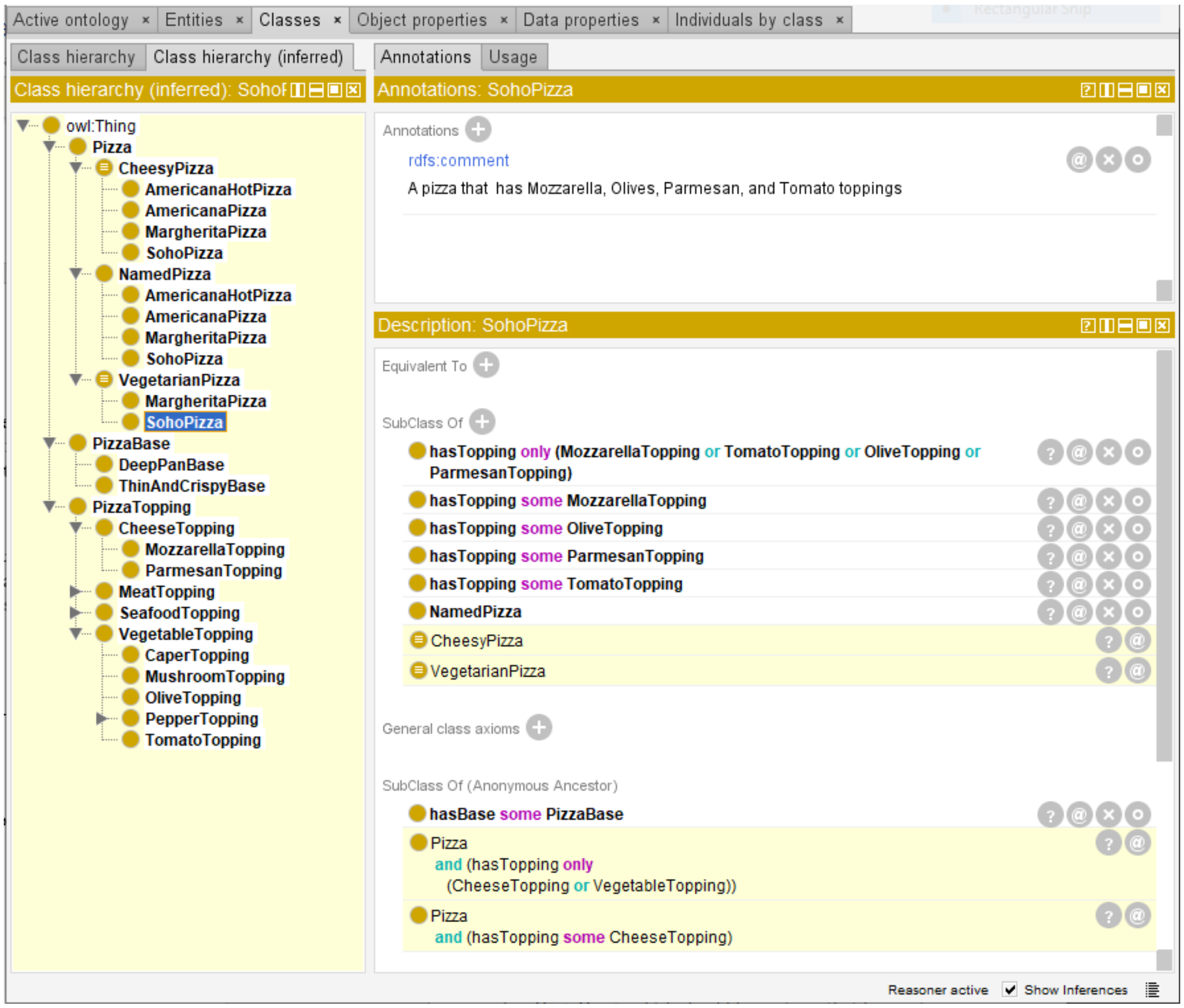
3. Select the Class expression editor tab from the pop-up window. Type in the Description Logic axiom: hasTopping only (MozzarellaTopping or TomatoTopping).

4. Click on OK.

5. Repeat steps 1-4 but this time click on SohoPizza and use the axiom: hasTopping only (MozzarellaTopping or TomatoTopping or ParmesanTopping or OliveTopping).

6. Synchronize the reasoner.

The previous axioms said that for example that it was necessary for any `Pizza` that was a `MargheritaPizza` to have a `MozzarellaTopping` and a `TomatoTopping`. The new axioms say that a `MargheritaPizza` can *only* have these toppings and similarly for `SohoPizza` and its toppings. This should supply the needed information for the reasoner to now make them both subclasses of `VegetarianPizza`. Go to the Class hierarchy (inferred) tab. You should now see that `MargheritaPizza` and `SohoPizza` are both classified as subclasses of `VegetarianPizza`. Your UI should now look similar to figure 4.23. Note the various axioms highlighted in yellow. Those are all additional inferences supplied by the reasoner. For experience you might want to click on some of the ? icons next to these inferences to see the explanations generated by the reasoner. As you develop more complex ontologies this is a powerful tool to debug and design your ontology.



Class hierarchy (inferred): SohoPizza

```
owl:Thing
  Pizza
    CheesyPizza
      AmericanaHotPizza
      AmericanaPizza
      MargheritaPizza
      SohoPizza
    NamedPizza
      AmericanaHotPizza
      AmericanaPizza
      MargheritaPizza
      SohoPizza
    VegetarianPizza
      MargheritaPizza
      SohoPizza
  PizzaBase
    DeepPanBase
    ThinAndCrispyBase
  PizzaTopping
    CheeseTopping
      MozzarellaTopping
      ParmesanTopping
    MeatTopping
    SeafoodTopping
    VegetableTopping
      CaperTopping
      MushroomTopping
      OliveTopping
      PepperTopping
      TomatoTopping
```

Annotations: SohoPizza

Annotations

rdfs:comment

A pizza that has Mozzarella, Olives, Parmesan, and Tomato toppings

Description: SohoPizza

Equivalent To

SubClass Of

- hasTopping only (MozzarellaTopping or TomatoTopping or OliveTopping or ParmesanTopping)
- hasTopping some MozzarellaTopping
- hasTopping some OliveTopping
- hasTopping some ParmesanTopping
- hasTopping some TomatoTopping
- NamedPizza
- CheesyPizza
- VegetarianPizza

General class axioms

SubClass Of (Anonymous Ancestor)

- hasBase some PizzaBase
- Pizza and (hasTopping only (CheeseTopping or VegetableTopping))
- Pizza and (hasTopping some CheeseTopping)

Figure 4.23 The Reasoner Inferred that Margherita and Soho Pizzas are subclasses of VegetarianPizza

4.14 Defining an Enumerated Class

A powerful tool in the object-oriented programming (OOP) community is the concept of design patterns. The idea of a design pattern is to capture a reusable model that is at a higher level of abstraction than a specific code library. One of the first and most common design patterns was the Model-View-Controller pattern first used in Smalltalk and now almost the default standard for good user interface design. Since there are significant differences between OWL and standard OOP the many excellent books on OOP design patterns don’t directly translate into OWL design patterns. Also, since the use of OWL is more recent than OOP there does not yet exist the excellent documentation of OWL patterns that the OOP community has. However, there are already many design patterns that have been documented for OWL and that can provide users with ways to save time and to standardize their designs according to best practices.

One of the most common OWL design patterns is an enumerated class. When a property has only a few possible values it can be useful to create a class to represent those values and to explicitly define the class by listing each possible value. We will show an example of such an enumerated class by creating a new