

A Practical Guide To Building OWL Ontologies Using Protégé 4
and CO-ODE Tools
Edition 1.1

Matthew Horridge,
Simon Jupp, Georgina Moulton, Alan Rector, Robert Stevens, Chris Wroe

THE UNIVERSITY OF MANCHESTER

Copyright © The University Of Manchester

October 16, 2007

Contents

1	Introduction	7
1.1	Conventions	7
2	Requirements	9
3	What are OWL Ontologies?	10
3.1	The Three Species Of OWL	10
3.1.1	OWL-Lite	10
3.1.2	OWL-DL	11
3.1.3	OWL-Full	11
3.1.4	Choosing The Sub-Language To Use	11
3.2	Components of OWL Ontologies	11
3.2.1	Individuals	11
3.2.2	Properties	12
3.2.3	Classes	12
4	Building An OWL Ontology	14
4.1	Named Classes	16
4.2	Disjoint Classes	19
4.3	Using The OWL Tools To Create Classes	19
4.4	OWL Properties	24
4.5	Inverse Properties	28

4.6	OWL Object Property Characteristics	29
4.6.1	Functional Properties	29
4.6.2	Inverse Functional Properties	31
4.6.3	Transitive Properties	31
4.6.4	Symmetric Properties	31
4.6.5	Antisymmetric properties	34
4.6.6	Reflexive properties	34
4.6.7	Irreflexive properties	35
4.7	Property Domains and Ranges	35
4.8	Describing And Defining Classes	38
4.8.1	Property Restrictions	39
4.8.2	Existential Restrictions	42
4.9	Using A Reasoner	50
4.9.1	Determining the OWL Sub-Language	50
4.9.2	Invoking The Reasoner	50
4.9.3	Inconsistent Classes	52
4.10	Necessary And Sufficient Conditions (Primitive and Defined Classes)	55
4.10.1	Primitive And Defined Classes	59
4.11	Automatic Classification	59
4.11.1	Classification Results	62
4.12	Universal Restrictions	62
4.13	Automatic Classification and Open World Reasoning	66
4.13.1	Closure Axioms	67
4.14	Value Partitions	71
4.14.1	Covering Axioms	73
4.15	Using the Matrix Wizard	74
4.16	Cardinality Restrictions	78

4.17	Qualified Cardinality Restrictions	80
5	Datatype Properties	82
6	More On Open World Reasoning	86
7	Creating Other OWL Constructs In Protégé 4	92
7.1	Creating Individuals	92
7.2	hasValue Restrictions	94
7.3	Enumerated Classes	96
7.4	Annotation Properties	97
7.5	Multiple Sets Of Necessary & Sufficient Conditions	99
7.6	Role Chains	101

Copyright

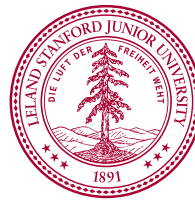
Copyright The University Of Manchester 2007

Acknowledgements

I would like to acknowledge and thank my colleagues at the University Of Manchester and also Stanford University for proof reading this tutorial/guide and making helpful comments and suggestions as to how it could be improved. In particular I would like to thank my immediate colleagues: Alan Rector, Nick Drummond, Hai Wang and Julian Seidenberg at the University Of Manchester, who suggested changes to early drafts of the tutorial in order to make things clearer and also ensure the technical correctness of the material. Alan was notably helpful in suggesting changes that made the tutorial flow more easily. I am grateful to Chris Wroe and Robert Stevens who conceived the original idea of basing the tutorial on an ontology about pizzas. Finally, I would also like to thank Natasha Noy from Stanford University for using her valuable experience in teaching, creating and giving tutorials about Protégé to provide detailed and useful comments about how initial drafts of the tutorial/guide could be made better.

This work was supported in part by the CO-ODE project funded by the UK Joint Information Services Committee and the HyOntUse Project (GR/S44686) funded by the UK Engineering and Physical Science Research Council and by 21XS067A from the National Cancer Institute.

<http://www.co-ode.org>



Chapter 1

Introduction

This guide introduces Protégé 4 for creating OWL ontologies. Chapter 3 gives a brief overview of the OWL ontology language. Chapter 4 focuses on building an OWL-DL ontology and using a Description Logic Reasoner to check the consistency of the ontology and automatically compute the ontology class hierarchy. Chapter 7 describes some OWL constructs such as hasValue Restrictions and Enumerated classes, which aren't directly used in the main tutorial.

1.1 Conventions


Class, property and individual names are written in a sans serif font **like this**.

Names for user interface views are presented in a style '**like this**'.

Where exercises require information to be typed into Protégé 4 a type writer font is used **like this**.

Exercises and required tutorial steps are presented like this:

Exercise 1: Accomplish this

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



Tips and suggestions related to using Protégé 4 and building ontologies are presented like this.



Explanation as to what things mean are presented like this.



Potential pitfalls and warnings are presented like this.



General notes are presented like this.



Vocabulary explanations and alternative names are presented like this.

Chapter 2

Requirements

In order to follow this tutorial you must have Protégé 4, which is available from the Protégé website ¹, and the Protégé Plugins which are available via the CO-ODE web site ². It is also recommended (but not necessary) to use the OWLViz plugin, which allows the asserted and inferred classification hierarchies to be visualised, and is available from the CO-ODE web site, or can be installed when Protégé 4 is installed. For installation steps, please see the documentation for each component.

¹<http://protege.stanford.edu>

²<http://www.co-ode.org>

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)¹. Like Protégé, OWL makes it possible to describe concepts but it also provides new facilities. It has a richer set of operators - e.g. intersection, union and negation. It is based on a different logical model which makes it possible for concepts to be defined as well as described. Complex concepts can therefore be built up in definitions out of simpler concepts. Furthermore, the logical model allows the use of a reasoner which can check whether or not all of the statements and definitions in the ontology are mutually consistent and can also recognise 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.

3.1 The Three Species Of OWL

OWL ontologies may be categorised into three species or sub-languages: OWL-Lite, OWL-DL and OWL-Full. A defining feature of each sub-language is its expressiveness. OWL-Lite is the least expressive sub-language. OWL-Full is the most expressive sub-language. The expressiveness of OWL-DL falls between that of OWL-Lite and OWL-Full. OWL-DL may be considered as an extension of OWL-Lite and OWL-Full an extension of OWL-DL.

3.1.1 OWL-Lite

OWL-Lite is the syntactically simplest sub-language. It is intended to be used in situations where only a simple class hierarchy and simple constraints are needed. For example, it is envisaged that OWL-Lite will provide a quick migration path for existing thesauri and other conceptually simple hierarchies.

¹<http://www.w3.org/TR/owl-guide/>

3.1.2 OWL-DL

OWL-DL is much more expressive than OWL-Lite. OWL-DL and OWL-Lite are based on *Description Logics* (hence the suffix DL). Description Logics are a decidable fragment of First Order Logic² and are therefore amenable to automated reasoning. It is therefore possible to automatically compute the classification hierarchy³ and check for inconsistencies in an ontology that conforms to OWL-DL. **This tutorial focuses on OWL-DL.**

3.1.3 OWL-Full

OWL-Full is the most expressive OWL sub-language. It is intended to be used in situations where very high expressiveness is more important than being able to guarantee the decidability or computational completeness of the language. It is therefore not possible to perform automated reasoning on OWL-Full ontologies.

3.1.4 Choosing The Sub-Language To Use

For a more detailed synopsis of the three OWL sub-languages see the OWL Web Ontology Language Overview⁴. Although many factors come into deciding the appropriate sub-language to use, there are some simple rules of thumb.

- The choice between OWL-Lite and OWL-DL may be based upon whether the simple constructs of OWL-Lite are sufficient or not.
- The choice between OWL-DL and OWL-Full may be based upon whether it is important to be able to carry out automated reasoning on the ontology or whether it is important to be able to use highly expressive and powerful modelling facilities such as meta-classes (classes of classes).

Protégé 4 does not make the distinction between editing OWL-Lite and OWL-DL ontologies.

3.2 Components of OWL Ontologies

OWL ontologies have similar components to Protégé frame based ontologies. However, the terminology used to describe these components is slightly different from that used in Protégé. An OWL ontology consists of Individuals, Properties, and Classes, which roughly correspond to Protégé frames Instances, Slots and Classes.

3.2.1 Individuals

Individuals, represent objects in the domain that we are interested in⁵. An important difference between Protégé and OWL is that OWL does not use the Unique Name Assumption (UNA). This means that

²Logics are *decidable* if computations/algorithms based on the logic will terminate in a *finite* time.

³Also known as *subsumption reasoning*.

⁴<http://www.w3.org/TR/owl-features>

⁵Also known as *the domain of discourse*.



Figure 3.1: Representation Of Individuals

two different names could actually refer to the same individual. For example, “Queen Elizabeth”, “The Queen” and “Elizabeth Windsor” *might* all refer to the same individual. In OWL, it must be explicitly stated that individuals are the same as each other, or different to each other — otherwise they *might* be the same as each other, or they *might* be different to each other. Figure 3.1 shows a representation of some individuals in some domain – in this tutorial we represent individuals as diamonds in diagrams.



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

3.2.2 Properties

Properties are *binary* relations⁶ on *individuals* - i.e. properties link *two* individuals together⁷. For example, the property **hasSibling** might link the individual **Matthew** to the individual **Gemma**, or the property **hasChild** might link the individual **Peter** to the individual **Matthew**. Properties can have inverses. For example, the inverse of **hasOwner** is **isOwnedBy**. Properties can be limited to having a single value – i.e. to being *functional*. They can also be either *transitive* or *symmetric*. These ‘property characteristics’ are explained in detail section 4.8. Figure 3.2 shows a representation of some properties linking some individuals together.



Properties are roughly equivalent to *slots* in Protégé . They are also known as *roles* in description logics and *relations* in UML and other object oriented notions. In GRAIL and some other formalisms they are called *attributes*.

3.2.3 Classes

OWL classes are interpreted as *sets* that contain individuals. They are *described* using formal (mathematical) descriptions that state precisely the requirements for membership of the class. For example, the class **Cat** would contain all the individuals that are cats in our domain of interest.⁸ Classes may be

⁶A binary relation is a relation between *two* things.

⁷Strictly speaking we should speak of ‘instances of properties’ linking individuals, but for the sake of brevity we will keep it simple.

⁸Individuals may belong to more than one class.

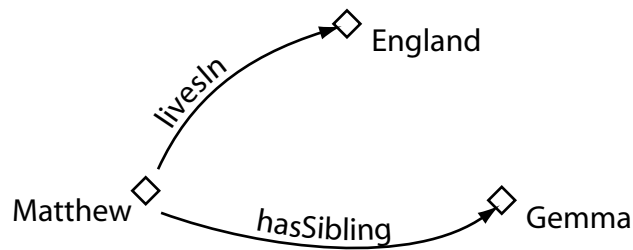


Figure 3.2: Representation Of Properties

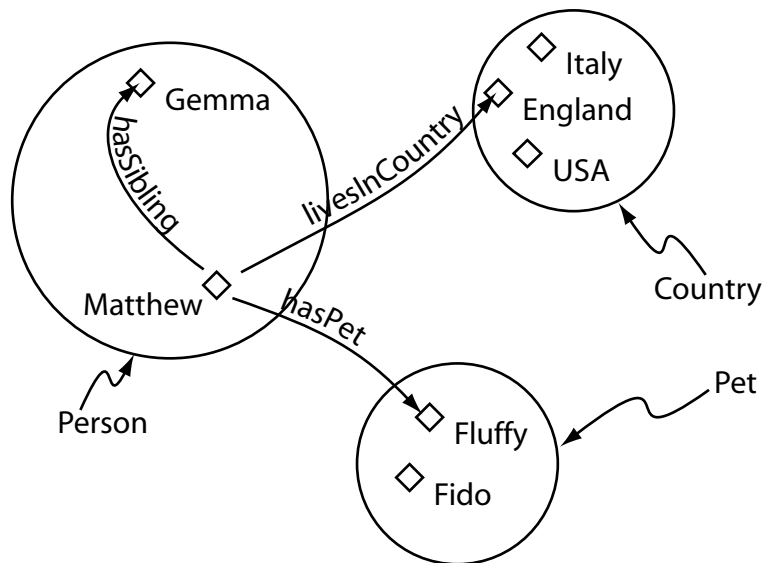


Figure 3.3: Representation Of Classes (Containing Individuals)

organised into a superclass-subclass hierarchy, which is also known as a *taxonomy*. Subclasses specialise (‘are subsumed by’) their superclasses. For example consider the classes **Animal** and **Cat** – **Cat** might be a subclass of **Animal** (so **Animal** is the superclass of **Cat**). This says that, ‘All cats are animals’, ‘All members of the class **Cat** are members of the class **Animal**’, ‘Being a **Cat** implies that you’re an **Animal**’, and ‘**Cat** is *subsumed* by **Animal**’. One of the key features of OWL-DL is that these superclass-subclass relationships (subsumption relationships) can be computed automatically by a *reasoner* – more on this later. Figure 3.3 shows a representation of some classes containing individuals – classes are represented as circles or ovals, rather like sets in Venn diagrams.



The word *concept* is sometimes used in place of class. Classes are a concrete representation of concepts.

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

Chapter 4

Building An OWL Ontology

This chapter describes how to create an ontology of Pizzas. We use Pizzas because we have found them to provide many useful examples.¹

Exercise 2: Create a new OWL Ontology

1. Start Protégé
 2. When the Welcome To Protégé dialog box appears, press the ‘**Create New OWL Ontology**’.
 3. A ‘Create Ontology URI Wizard will appear’. Every ontology is named using a Unique Resource Identifier (URI). Replace the default URI with `http://www.pizza.com/ontologies/pizza.owl` and press ‘**Next**’.
 4. You will also want to save your Ontology to a file on your PC. You can browse your hard disk and save your ontology to a new file, you might want to name your file ‘**pizza.owl**’. Once you choose a file press ‘**Finish**’.
-

After a short amount of time, a new empty Protégé file will have been created and the ‘**Active Ontology Tab**’ shown in Figure 4.1 will be visible. As can be seen from Figure 4.1, the ‘**Active Ontology Tab**’ allows information about the ontology to be specified. For example, the ontology URI can be changed, annotations on the ontology such as comments may be added and edited, and namespaces and imports can be set up via this tab.

¹The Ontology that we will create is based upon a Pizza Ontology that has been used as the basis for a course on editing DAML+OIL ontologies in OilEd (<http://oiled.man.ac.uk>), which was taught at the University Of Manchester.

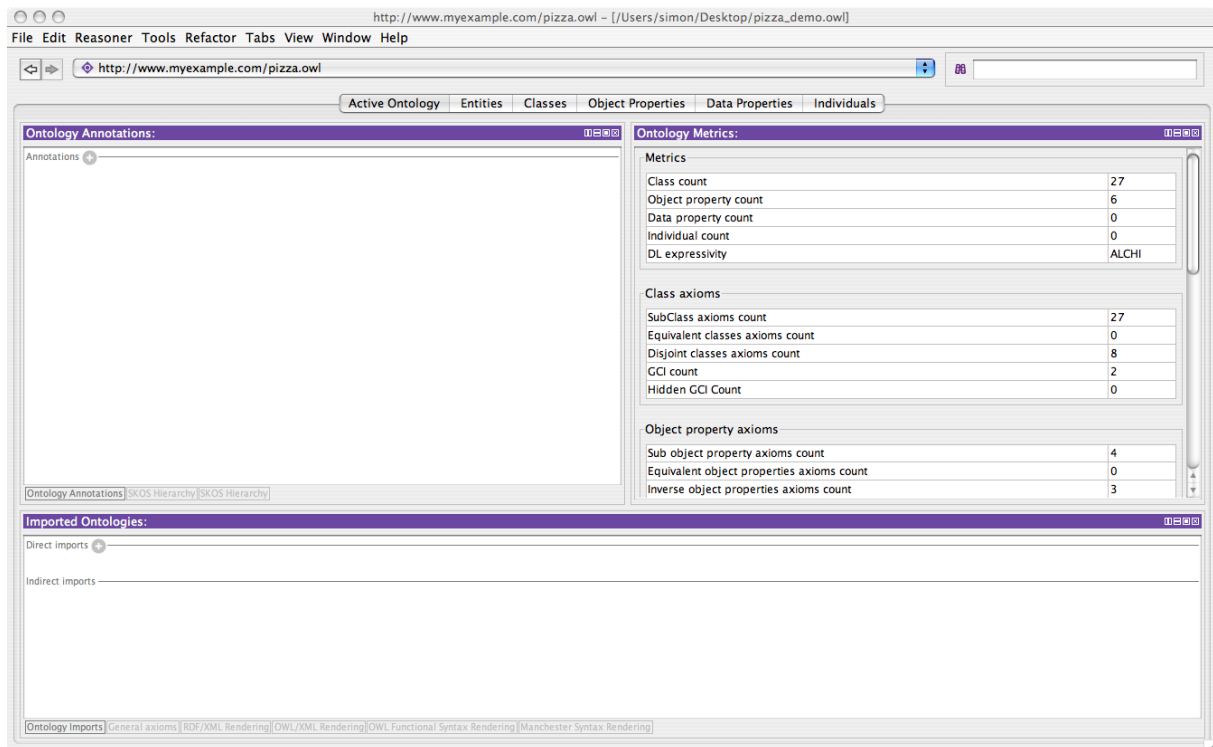


Figure 4.1: The Active Ontology Tab

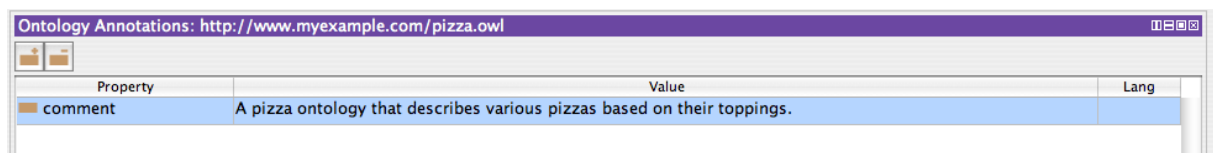


Figure 4.2: The Ontology Annotations View – The ontology has a comment as indicated by the `comment` annotation

Exercise 3: Add a comment to the ontology

1. Ensure that the '**Active Ontology Tab**' is selected.
2. In the '**Ontology Annotations**' view, double click to the right of the comment property name. An editing window will appear in the table.
3. Enter a comment such as `A pizza ontology that describes various pizzas based on their toppings.` and press CTRL+ENTER to assign the comment. The annotations view on the '**Active Ontology Tab**' should look like the picture shown in Figure 4.2

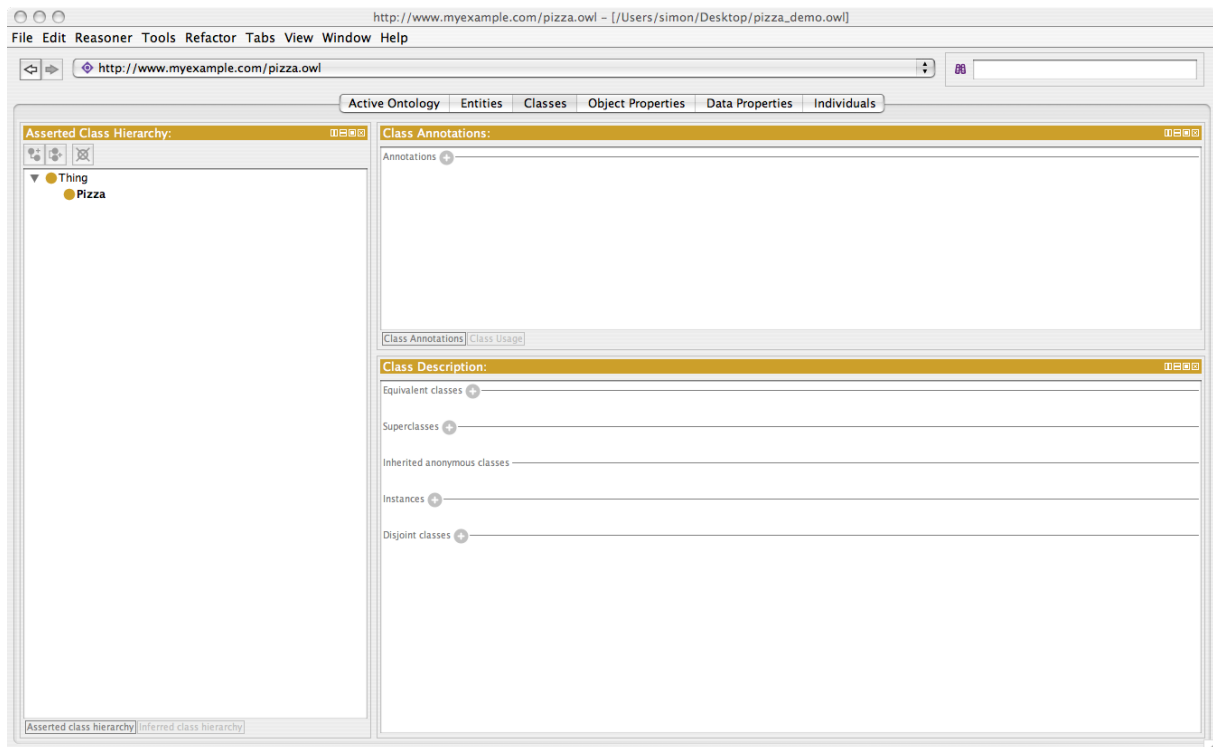


Figure 4.3: The Classes Tab

4.1 Named Classes

As mentioned previously, an ontology contains classes – indeed, the main building blocks of an OWL ontology are classes. In Protégé 4, editing of classes is carried out using the ‘**Classes Tab**’ shown in Figure 4.3. The initial class hierarchy tree view should resemble the picture shown in Figure 4.4. The empty ontology contains one class called **Thing**. As mentioned previously, OWL classes are interpreted as sets of *individuals* (or sets of objects). The class **Thing** is the class that represents the set containing *all* individuals. Because of this all classes are subclasses of **Thing**.²

Let’s add some classes to the ontology in order to define what we believe a pizza to be.

²Thing is part of the OWL Vocabulary, which is defined by the ontology located at <http://www.w3.org/2002/07/owl/#>

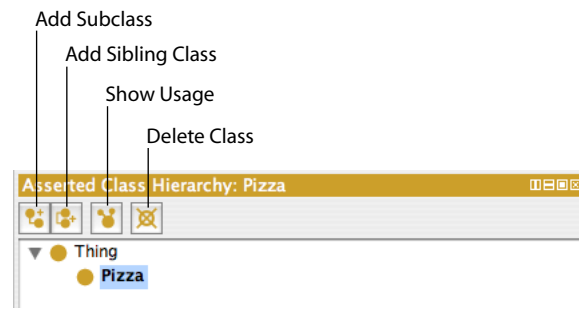


Figure 4.4: The Class Hierarchy Pane

Exercise 4: Create classes Pizza, PizzaTopping and PizzaBase

1. Ensure that the '**Classes Tab**' is selected.
 2. Press the '**Add subclass**' button shown in Figure 4.4. This button creates a new class as a subclass of the selected class (in this case we want to create a subclass of **Thing**).
 3. A dialog will appear for you to name your class, enter **Pizza** (as shown in Figure 4.5) and hit return.
 4. Repeat the previous steps to add the classes **PizzaTopping** and also **PizzaBase**, ensuring that **Thing** is selected before the '**Add subclass**' button is pressed so that the classes are created as subclasses of **Thing**.
-

The class hierarchy should now resemble the hierarchy shown in Figure 4.6.

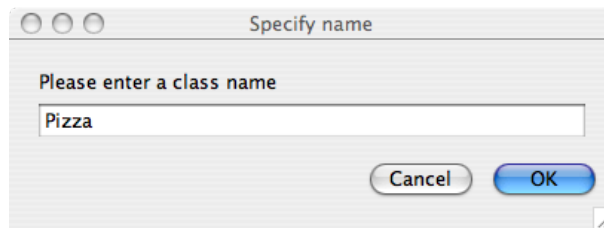


Figure 4.5: Class Name Dialog

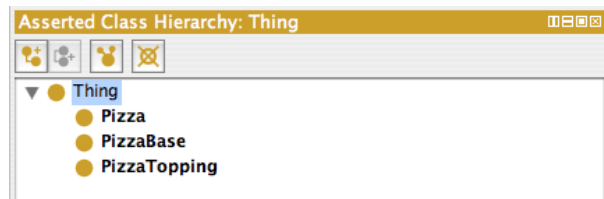


Figure 4.6: The Initial Class Hierarchy

TIP

After creating **Pizza**, instead of re-selecting **Thing** and using the ‘**Create subclass**’ button to create **PizzaTopping** and **PizzaBase** as further subclasses of **Thing**, the ‘**Add sibling class**’ button (shown in Figure 4.4) can be used. While **Pizza** is selected, use the ‘**Create sibling class**’ button to create **PizzaTopping** and then use this button again (while **PizzaTopping** is selected) to create **PizzaBase** as sibling classes of **PizzaTopping** – these classes will of course still be created as subclasses of **Thing**, since **Pizza** is a subclass of **Thing**.

Vocabulary



A class hierarchy may also be called a taxonomy.

TIP

Although there are no mandatory naming conventions for OWL classes, we recommend that all class names should start with a capital letter and should not contain spaces. (This kind of notation is known as CamelBack notation and is the notation used in this tutorial). For example **Pizza**, **PizzaTopping**, **MargheritaPizza**. Alternatively, you can use underscores to join words. For example **Pizza_Topping**. Which ever convention you use, it is important to be consistent.

4.2 Disjoint Classes

Having added the classes **Pizza**, **PizzaTopping** and **PizzaBase** to the ontology, we now need to say these classes are *disjoint*, so that an individual (or object) cannot be an instance of more than one of these three classes. To specify classes that are disjoint from the selected class click the ‘**Disjoints classes**’ button which is located at the bottom of the ‘**Class Description**’ view.

Exercise 5: Make Pizza, PizzaTopping and PizzaBase disjoint from each other

1. Select the class **Pizza** in the class hierarchy.
 2. Press the ‘**Disjoint classes**’ button in the ‘**class description**’ view, this will bring up a dialog where you can select multiple classes to be disjoint. This will make **PizzaBase** and **PizzaTopping** (the sibling classes of **Pizza**) disjoint from **Pizza**.
-

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

MEANING



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

4.3 Using The OWL Tools To Create Classes

³ The OWL Tools are plugins, which is available from the Protégé web site, is an extensible set of Tools that are designed to make carrying out common, repetitive and time consuming tasks easy. In this section we will use the ‘**Create Class Hierarchy**’ tool to add some subclasses of the class **PizzaBase**, this tool is distributed by default with Protégé ⁴. To use the OWL Tools you must ensure that the OWL Tools are installed and configured in Protégé.

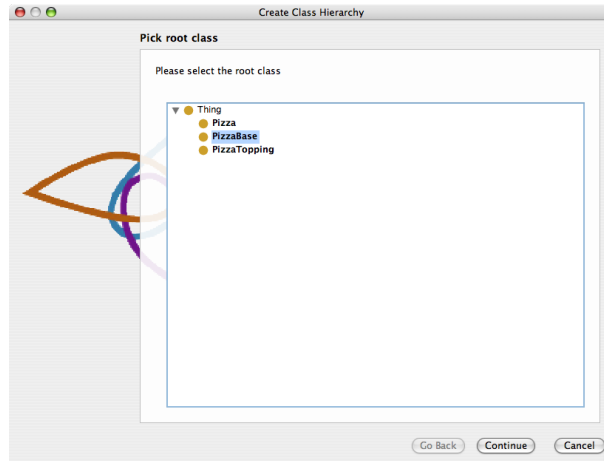


Figure 4.7: Create Class Hierarchy: Select class page

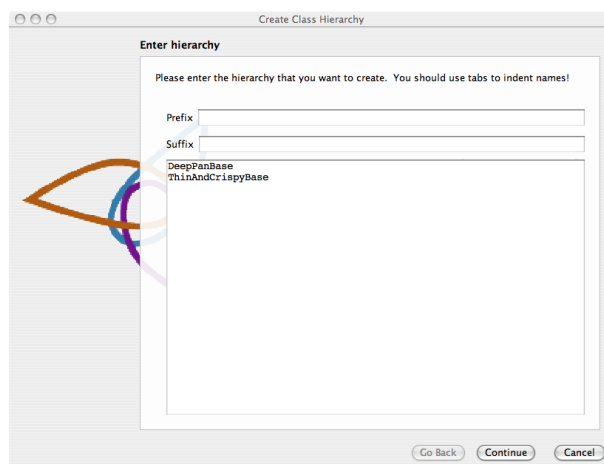


Figure 4.8: Create Class Hierarchy: Enter classes page

Exercise 6: Use the ‘Create Class Hierarchy’ Tool to create ThinAndCrispy and DeepPan as subclasses of PizzaBase

1. Select the class **PizzaBase** in the class hierarchy.
 2. From the Tools menu on the Protégé menu bar select ‘**Create Class Tierarchy...**’.
 3. The tools shown in Figure 4.7 will appear. Since we preselected the **PizzaBase** class, the first radio button at the top of the tool should be prompting us to create the classes under the class **PizzaBase**. If we had not preselected **PizzaBase** before starting the tool, then the tree could be used to select the class.
 4. Press the ‘**Next**’ button on the tool—The page shown in Figure 4.8 will be displayed. We now need to tell the tool the subclasses of **PizzaBase** that we want to create. In the large text area, type in the class name **ThinAndCrispyBase** (for a thin based pizza) and hit return. Also enter the class name **DeepPanBase** so that the page resembles that shown in Figure 4.8 .
 5. Hit the ‘**Next**’ button on the tool. The tool checks that the names entered adhere to the naming styles that have previously been mentioned (No spaces etc.). It also checks for uniqueness – no two class names may be the same. If there are any errors in the class names, they will be presented on this page, along with suggestions for corrections.
 6. Hit the ‘**Next**’ button on the tool. Ensure the tick box ‘**Make all new classes disjoint**’ is *ticked* — instead of having to use the disjoint classes view, the tool will automatically make the new classes disjoint for us.
-

After the ‘**Next**’ button has been pressed, the tool creates the classes, makes them disjoint. Click ‘**Finish**’ to dismiss the tool. The ontology should now have **ThinAndCrispyBase** and also **DeepPanBase** as subclasses of **PizzaBase**. These new classes should be disjoint to each other. Hence, a pizza base cannot be both thin and crispy *and* deep pan. It isn’t difficult to see that if we had a lot of classes to add to the ontology, the tool would dramatically speed up the process of adding them.



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

Creating Some Pizza Toppings

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

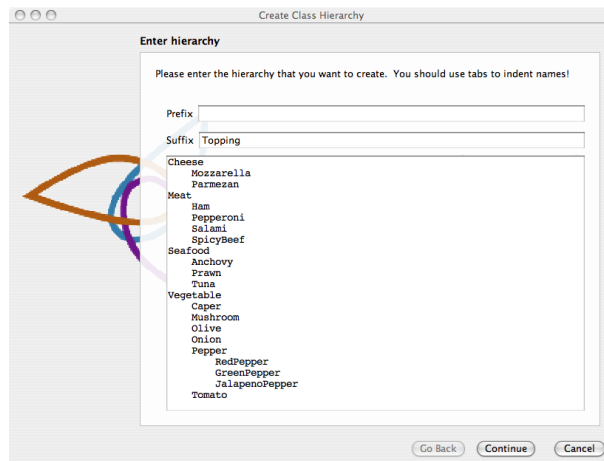


Figure 4.9: Topping Hierarchy

and seafood toppings.

Exercise 7: Create some subclasses of PizzaTopping

1. Select the class **PizzaTopping** in the class hierarchy.
2. Invoke the ‘**Create class hierarchy...**’ tool in the same way as the tool was started in the previous exercise.
3. Ensure **PizzaTopping** is selected and press the ‘**Next**’ button.
4. We want all our topping classes to end in **topping**, so in the ‘**Suffix all in list with**’ field, enter **Topping**. The tool will save us some typing by automatically appending **Topping** to all of our class names.
5. The tool allows a hierarchy of classes to be entered using a tab indented tree. Using the text area in the tool, enter the class names as shown in Figure 4.9. Note that class names must be indented using tabs, so for example **SpicyBeef**, which we want to be a subclass of **Meat** is entered under **Meat** and indented with a tab. Likewise, **Pepperoni** is also entered under **Meat** below **SpicyBeef** and also indented with a tab.
6. Having entered a tab indented list of classes, press the ‘**Next**’ button and then make sure that ‘**Make all primitive siblings disjoint**’ check box is ticked so that new *sibling* classes are made disjoint with each other.
7. Press the ‘**Finish**’ button to create the classes. Press ‘**Finish**’ again to close the tool.

The class hierarchy should now look similar to that shown in Figure 4.10 (the ordering of classes may be slightly different).

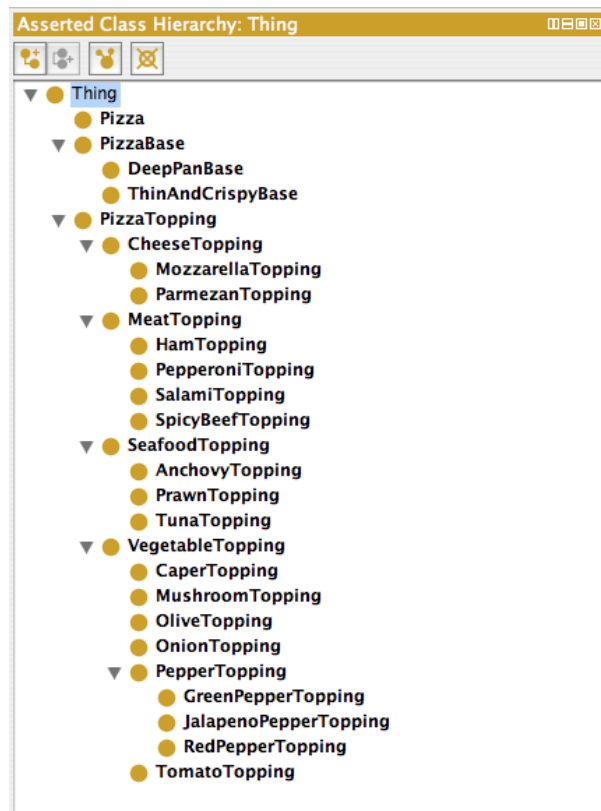


Figure 4.10: Class Hierarchy

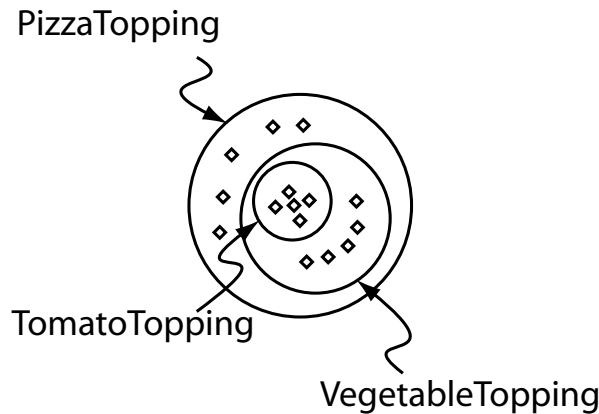


Figure 4.11: The Meaning Of Subclass — *All* individuals that are members of the class **TomatoTopping** are members of the class **VegetableTopping** and members of the class **PizzaTopping** as we have stated that **TomatoTopping** is a subclass of **VegetableTopping** which is a subclass of **PizzaTopping**

MEANING



Up to this point, we have created some simple named classes, some of which are *subclasses* of other classes. The construction of the class hierarchy may have seemed rather intuitive so far. 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**, or for **TomatoTopping** to be a *subclass* of **VegetableTopping**? In OWL *subclass* means *necessary implication*. In other words, if **VegetableTopping** is a *subclass* of **PizzaTopping** then *ALL* instances of **VegetableTopping** are instances of **PizzaTopping**, *without exception* — if something is a **VegetableTopping** then this *implies* that it is also a **PizzaTopping** as shown in Figure 4.11.^a

^aIt is for this reason that we seemingly pedantically named all of our toppings with the suffix of ‘Topping’, for example, **HamTopping**. Despite the fact that class names themselves carry no formal semantics in OWL (and in other ontology languages), if we had named **HamTopping** **Ham**, then this could have implied to human eyes that anything that is a kind of ham is also a kind of **MeatTopping** and also a **PizzaTopping**.

4.4 OWL Properties

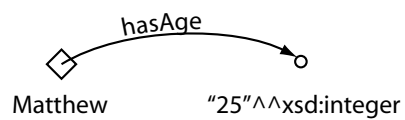
OWL Properties represent relationships. There are two main types of properties, *Object properties* and *Datatype properties*. Object properties are relationships between two individuals. In this chapter we will focus on *Object properties*; *datatype properties* are described in Chapter 5. Object properties link an individual to an individual. OWL also has a third type of property – *Annotation properties*³. Annotation properties can be used to add information (metadata — data about data) to classes, individuals and object/datatype properties. Figure 4.12 depicts an example of each type of property.

Properties may be created using the ‘**Object Properties**’ tab shown in Figure 4.13. Figure 4.14 shows the buttons located in the top left hand corner of the ‘**Object Properties**’ tab that are used for creating OWL properties. As can be seen from Figure 4.14, there are buttons for creating Datatype properties,

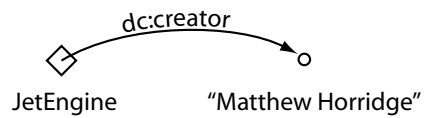
³Object properties and Datatype properties may be marked as Annotation properties



An object property linking the individual Matthew to the individual Gemma



A datatype property linking the individual Matthew to the data literal '25', which has a type of an xsd:integer.



An annotation property, linking the class 'JetEngine' to the data literal (string) "Matthew Horridge".

Figure 4.12: The Different types of OWL Properties

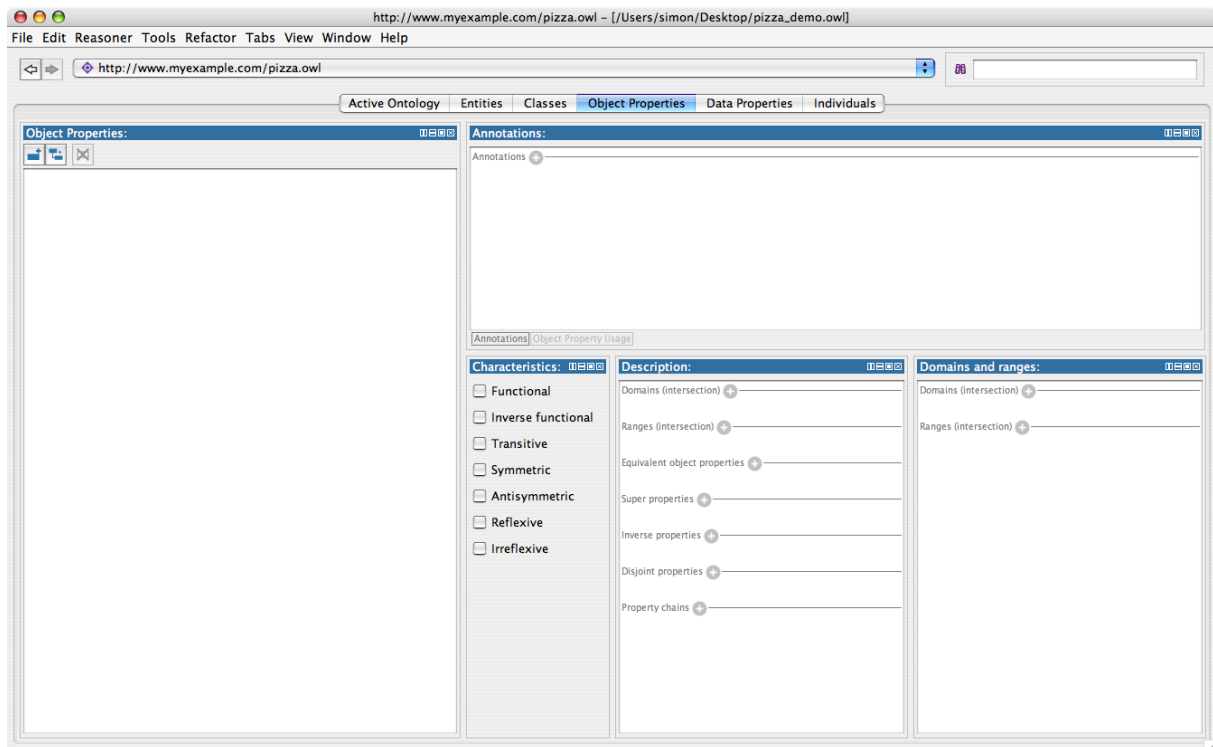


Figure 4.13: The PropertiesTab

Object properties and Annotation properties. Most properties created in this tutorial will be **Object properties**.

Exercise 8: Create an object property called hasIngredient

1. Switch to the '**Object Properties**' tab. Use the '**Add Object Property**' button (see Figure 4.14) to create a new Object property.
2. Name the property to **hasIngredient** using the '**Property Name Dialog**' that pops up, as shown in Figure 4.15 (The '**Property Name Dialog**').

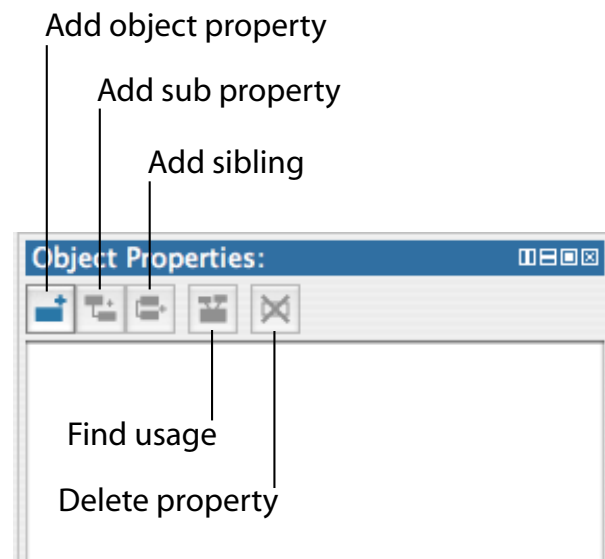


Figure 4.14: Property Creation Buttons — located on the Properties Tab above the property list/tree

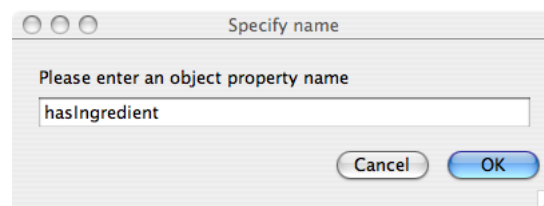


Figure 4.15: Property Name Dialog

TIP

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

^aThe English Prose Tooltip Generator displays the description of classes etc. in a more natural form of English, making it easy to understand a class description. The tooltips pop up when the mouse pointer is made to hover over a class description in the user interface.

Having added the **hasIngredient** property, we will now add two more properties — **hasTopping**, and **hasBase**. In OWL, properties may have sub properties, so that it is possible to form hierarchies of properties. Sub properties specialise their super properties (in the same way that subclasses specialise their superclasses). For example, the property **hasMother** might specialise the more general property of

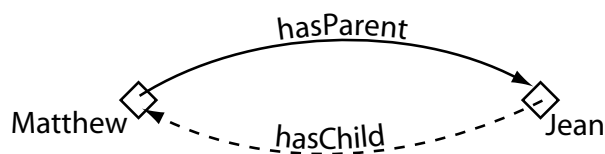


Figure 4.16: An Example Of An Inverse Property: `hasParent` has an inverse property that is `hasChild`

`hasParent`. In the case of our pizza ontology the properties `hasTopping` and `hasBase` should be created as sub properties of `hasIngredient`. If the `hasTopping` property (or the `hasBase` property) links two individuals this implies that the two individuals are related by the `hasIngredient` property.

Exercise 9: Create `hasTopping` and `hasBase` as sub-properties of `hasIngredient`

1. To create the `hasTopping` property as a sub property of the `hasIngredient` property, select the `hasIngredient` property in the property hierarchy on the ‘**Object Properties**’ tab.
2. Press the ‘**Add subproperty**’ button. A new object property will be created as a sub property of the `hasIngredient` property.
3. Name the new property to `hasTopping`.
4. Repeat the above steps but name the property `hasBase`.

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

4.5 Inverse Properties

Each object property may have a corresponding inverse property. If some property links individual **a** to individual **b** then its inverse property will link individual **b** to individual **a**. For example, Figure 4.16 shows the property `hasParent` and its inverse property `hasChild` — if **Matthew** `hasParent` **Jean**, then because of the inverse property we can infer that **Jean** `hasChild` **Matthew**.

Inverse properties can be created/specified using the inverse property view shown in Figure 4.17. For

completeness we will specify inverse properties for our existing properties in the Pizza Ontology.

Exercise 10: Create some inverse properties

1. Use the ‘**Add object property**’ button on the ‘**Object Properties**’ tab to create a new Object property called **isIngredientOf** (this will become the inverse property of **hasIngredient**).
 2. Press the ‘**Add inverse property**’ button on the inverse property view shown in Figure 4.17. This will display a dialog from which properties may be selected. Select the **hasIngredient** property and press ‘**OK**’. The property **hasIngredient** should now be displayed in the ‘**Inverse Property**’ view. The properties hierarchy should also now indicate that **hasIngredient** and **isIngredientOf** are inverse properties of each other.
 3. Select the **hasBase** property.
 4. Press the ‘**Add inverse property**’ button on the ‘**Inverse Property**’ view. This will pop up a dialog that contains information about the newly created property. Use this dialog to rename the property **isBaseOf** and then close the dialog window (using the operating system close window button on the title bar). Notice that the **isBaseOf** property has been created as a sub property of the **isIngredientOf** property. This corresponds to the fact that **hasBase** is a sub property of **hasIngredient**, and **isIngredientOf** is the inverse property of **hasIngredient**.
 5. Select the **hasTopping** property.
 6. Press the ‘**Add inverse property**’ button on the ‘**Inverse Property**’ view. Use the property dialog that pops up to rename the property **isToppingOf**. Close the dialog.
-

The property hierarchy should now look like the picture shown in Figure 4.18.

4.6 OWL Object Property Characteristics

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

4.6.1 Functional Properties

If a property is functional, for a given individual, there can be at most one individual that is related to the individual via the property. Figure 4.19 shows an example of a functional property **hasBirthMother** — something 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**⁴, then because **hasBirthMother** is a functional property, we can infer that **Peggy** and **Margaret** must be the same individual. It should be noted however, that if **Peggy** and **Margaret** were explicitly stated to be two different individuals then the above statements would lead to an inconsistency.

⁴The name Peggy is a diminutive form for the name Margaret

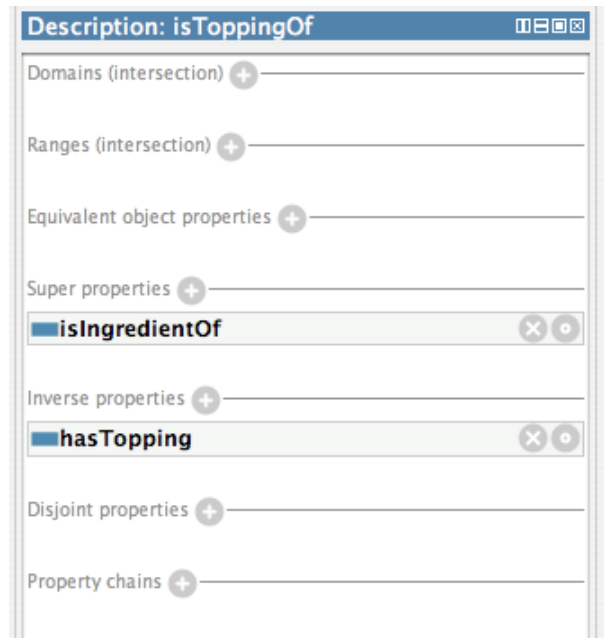


Figure 4.17: The Inverse Property View

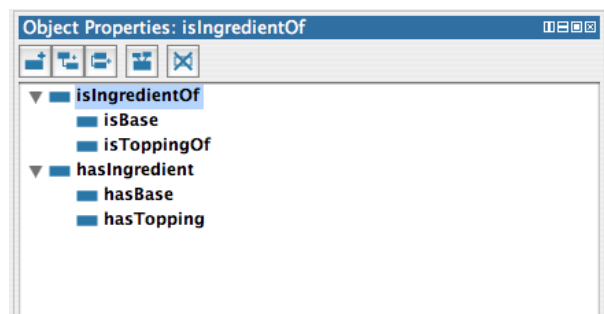


Figure 4.18: The Property Hierarchy

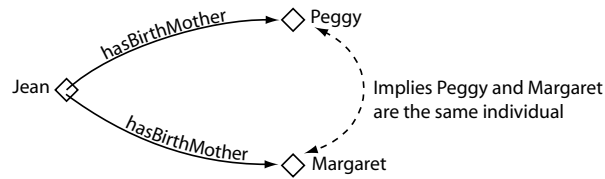


Figure 4.19: An Example Of A Functional Property: `hasBirthMother`



Functional properties are also known as *single valued properties* and also *features*.

4.6.2 Inverse Functional Properties

If a property is inverse functional then it means that the *inverse* property is *functional*. For a given individual, there can be at most one individual related to that individual via the property. Figure 4.20 shows an example of an inverse functional property `isBirthMotherOf`. This is the inverse property of `hasBirthMother` — since `hasBirthMother` is functional, `isBirthMotherOf` is inverse functional. If we state that **Peggy** is the birth mother of **Jean**, and we also state that **Margaret** is the birth mother of **Jean**, then we can infer that **Peggy** and **Margaret** are the same individual.

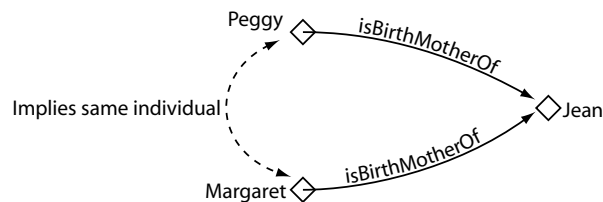


Figure 4.20: An Example Of An Inverse Functional Property: `isBirthMotherOf`

4.6.3 Transitive Properties

If a property is transitive, and the property relates individual **a** to individual **b**, and also individual **b** to individual **c**, then we can infer that individual **a** is related to individual **c** via property **P**. For example, Figure 4.21 shows an example of the transitive property `hasAncestor`. If the individual **Matthew** has an ancestor that is **Peter**, and **Peter** has an ancestor that is **William**, then we can infer that **Matthew** has an ancestor that is **William** — this is indicated by the dashed line in Figure 4.21.

4.6.4 Symmetric Properties

If a property **P** is symmetric, and the property relates individual **a** to individual **b** then individual **b** is also related to individual **a** via property **P**. Figure 4.22 shows an example of a symmetric property. If the individual **Matthew** is related to the individual **Gemma** via the `hasSibling` property, then we can infer

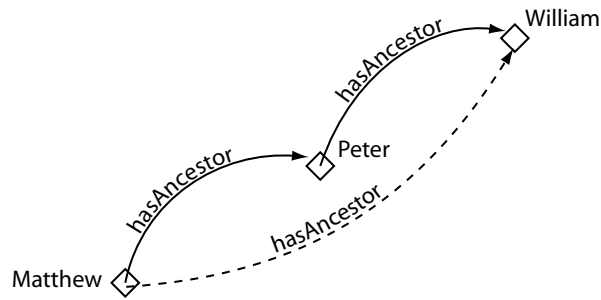


Figure 4.21: An Example Of A Transitive Property: `hasAncestor`

that **Gemma** must also be related to **Matthew** via the `hasSibling` property. In other words, if **Matthew** has a sibling that is **Gemma**, then **Gemma** must have a sibling that is **Matthew**. Put another way, the property is its own inverse property.

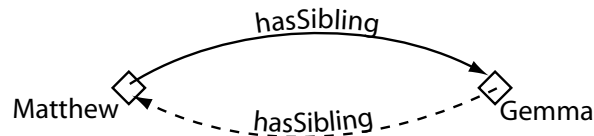


Figure 4.22: An Example Of A Symmetric Property: `hasSibling`

We want to make the `hasIngredient` property transitive, so that for example if a pizza topping has an ingredient, then the pizza itself also has that ingredient. To set the property characteristics of a property the property characteristics view shown in Figure 4.23 which is located in the lower right hand corner of the properties tab is used.

Exercise 11: Make the `hasIngredient` property transitive

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

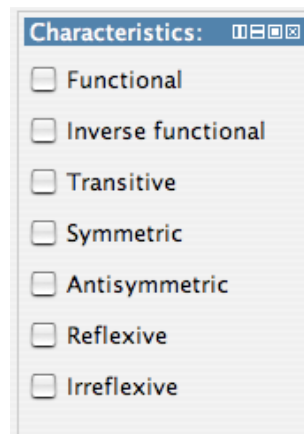


Figure 4.23: Property Characteristics Views



If a property is transitive then its inverse property should also be transitive.^a

^aAt the time of writing this must be done manually in Protégé 4 . However, the reasoner *will* assume that if a property is transitive, its inverse property is also a transitive.



Note that if a property is transitive then it cannot be functional.^a

^aThe reason for this is that transitive properties, by their nature, may form ‘chains’ of individuals. Making a transitive property functional would therefore not make sense.

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

Exercise 12: Make the hasBase property functional

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

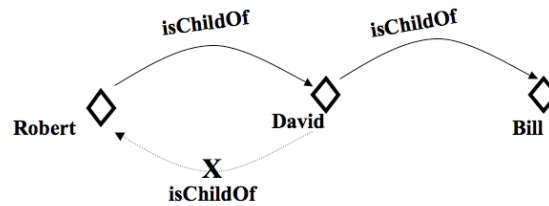


Figure 4.24: An example of the antisymmetric property hasChildOf

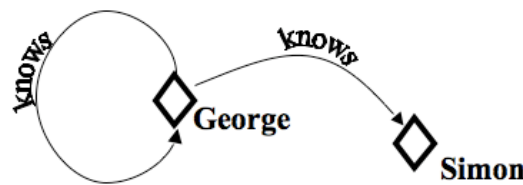


Figure 4.25: An example of a Reflexive Property: knows



NOTE

If a datatype property is selected, the property characteristics view will be reduced so that only options for ‘**Allows multiple values**’ and ‘**Inverse Functional**’ will be displayed. This is because OWL-DL does not allow datatype properties to be transitive, symmetric or have inverse properties.

4.6.5 Antisymmetric properties

If a property **P** is antisymmetric, and the property relates individual **a** to individual **b** then individual **b** cannot be related to individual **a** via property **P**. Figure 4.24 shows an example of a antisymmetric property. If the individual **Robert** is related to the individual **David** via the **isChildOf** property, then it can be inferred that **David** is not related to **Robert** via the **isChildOf** property. It is, however, reasonable to state that **David** could be related to another individual **Bill** via the **isChildOf** property. In other words, if **Robert** is a child of **David**, then **David** cannot be a child of **Robert**, but **David** can be a child of **Bill**.

4.6.6 Reflexive properties

A property **P** is said to be reflexive when the property must relate individual **a** to itself. In Figure 4.25 we can see an example of this: using the property **knows**, an individual **George** must have a relationship to itself using the property **knows**. In other words, **George** must know herself. However, in addition, it is possible for **George** to know other people; therefore the individual **George** can have a relationship with individual **Simon** along the property **knows**.

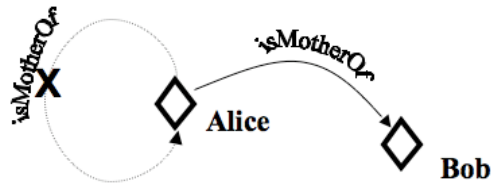


Figure 4.26: An example of a Irreflexive Property: `isMotherOf`

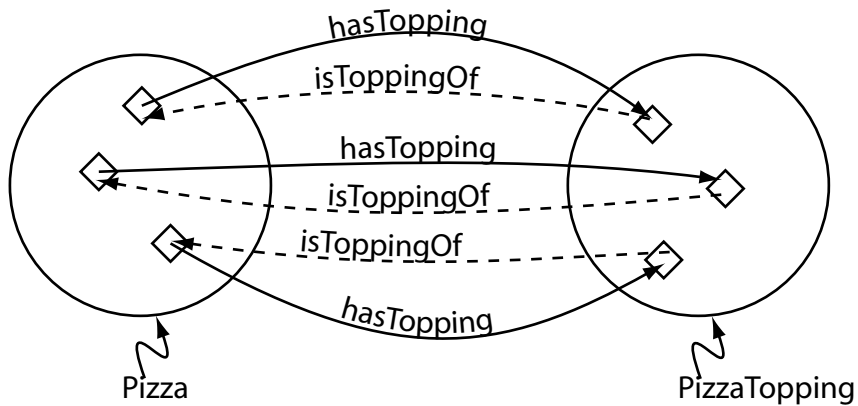


Figure 4.27: The domain and range for the `hasTopping` property and its inverse property `isToppingOf`. The domain of `hasTopping` is `Pizza` the range of `hasTopping` is `PizzaTopping` — the domain and range for `isToppingOf` are the domain and range for `hasTopping` swapped over

4.6.7 Irreflexive properties

If a property **P** is *irreflexive*, it can be described as a property that relates an individual **a** to individual **b**, where individual **a** and individual **b** are not the same. An example of this would be the property `motherOf`: an individual **Alice** can be related to individual **Bob** along the property `motherOf`, but **Alice** cannot be `motherOf` herself (Figure 4.26).

4.7 Property Domains and Ranges

Properties may have a *domain* and a *range* specified. Properties link individuals from the *domain* to individuals from the *range*. For example, in our pizza ontology, the property `hasTopping` would probably link individuals belonging to the class `Pizza` to individuals belonging to the class of `PizzaTopping`. In this case the *domain* of the `hasTopping` property is `Pizza` and the *range* is `PizzaTopping` — this is depicted in Figure 4.27.



Figure 4.28: Property Range View (For Object Properties)



Property Domains And Ranges In OWL — It is important to realise that in OWL domains and ranges should *not* be viewed as constraints to be checked. They are used as ‘axioms’ in reasoning. For example if the property **hasTopping** has the domain set as **Pizza** and we then applied the **hasTopping** property to **IceCream** (individuals that are members of the class **IceCream**), this would generally not result in an error. It would be used to infer that the class **IceCream** must be a subclass of **Pizza**! ^a.

^aAn error will only be generated (by a reasoner) if **Pizza** is disjoint to **IceCream**

We now want to specify that the **hasTopping** property has a *range* of **PizzaTopping**. To do this the range view shown in Figure 4.28 is used.

Exercise 13: Specify the range of hasTopping

1. Make sure that the **hasTopping** property is selected in the property hierarchy on the ‘**Object Properties**’ tab.
2. Press the ‘**add class**’ button on the ‘**Range View**’ (Figure 4.28). A dialog will appear that allows a class to be selected from the ontology class hierarchy.
3. Select **PizzaTopping** and press the ‘**OK**’ button. **PizzaTopping** should now be displayed in the range list.

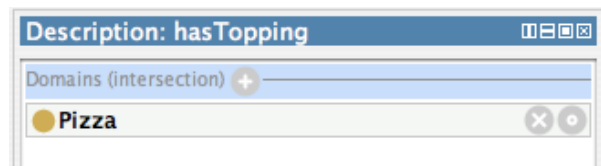


Figure 4.29: Property Domain View



NOTE

It is possible to specify multiple classes as the range for a property. If multiple classes are specified in Protégé 4 the range of the property is interpreted to be the *intersection* of the classes. For example, if the range of a property has the classes **Man** and **Woman** listed in the range view, the range of the property will be interpreted as **Man union Woman**.^a

^aSee section ?? for an explanation of what a union class is.

To specify the domain of a property the domain view shown in Figure 4.29 is used.

Exercise 14: Specify Pizza as the domain of the hasTopping property

1. Make sure that the **hasTopping** property is selected in the property hierarchy on the ‘**Object Properties**’ tab.
2. Press the ‘**Select and add class**’ button on the Domain View. A dialog will appear that allows a class to be selected from the ontology class hierarchy.
3. Select **Pizza** and press the OK button. **Pizza** should now be displayed in the domain list.

MEANING



This means that individuals that are used ‘on the left hand side’ of the **hasTopping** property will be inferred to be members of the class **Pizza**. Any individuals that are used ‘on the right hand side’ of the **hasTopping** property will be inferred to be members of the class **PizzaTopping**. For example, if we have individuals **a** and **b** and an assertion of the form **a hasTopping b** then it will be inferred that **a** is a member of the class **Pizza** and that **b** is a member of the class **PizzaTopping**.^a

^aThis will be the case even if **a** has not been asserted to be a member of the class **Pizza** and/or **b** has not been asserted to be a member of the class **PizzaTopping**.



NOTE

Take a look at the **isToppingOf** property, which is the inverse property of **hasTopping**. Notice that Protégé has automatically filled in domain and range of the **isToppingOf** property because the domain and range of the inverse property were specified. The range of **isToppingOf** is the domain of the inverse property **hasTopping**, and the domain of **isToppingOf** is the range of the inverse property **hasTopping**. This is depicted in Figure 4.27.

Exercise 15: Specify the domain and range for the **hasBase** property and its inverse property **isBaseOf**

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



TIP

In the previous steps we have ensured that the domains and ranges for properties are also set up for inverse properties in a correct manner. In general, domain for a property is the range for its inverse, and the range for a property is the domain for its inverse — Figure 4.27 illustrates this for the **hasTopping** and **isToppingOf**.



Although we have specified the domains and ranges of various properties for the purposes of this tutorial, we generally advise *against* doing this. The fact that domain and range conditions do not behave as constraints and the fact that they can cause ‘unexpected’ classification results can lead problems and unexpected side effects. These problems and side effects can be particularly difficult to track down in a large ontology.

4.8 Describing And Defining Classes

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

4.8.1 Property Restrictions

Recall that in OWL, properties describe binary relationships. Datatype properties describe relationships between individuals and data values. Object properties describe relationships between two individuals. For example, in Figure 3.2 the individual **Matthew** is related to the individual **Gemma** via the **hasSibling** property. Now consider all of the individuals that have a **hasSibling** relationship to some other individual. We can think of these individuals as belonging *the class of individuals* that have some **hasSibling** relationship. The key idea is that a class of individuals is described or defined by the relationships that these individuals participate in. In OWL we can define such classes by using *restrictions*.

Vocabulary



A *restriction* describes a class of individuals based on the relationships that members of the class participate in. In other words a *restriction* is a kind of *class*, in the same way that a named class is a kind of class.

Restriction Examples

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

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

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

- Quantifier Restrictions
- Cardinality Restrictions
- hasValue Restrictions.

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

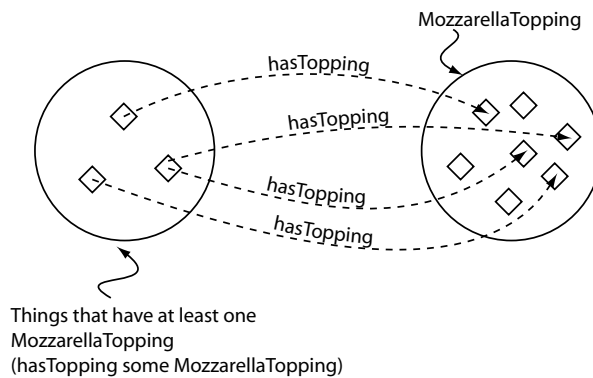


Figure 4.30: The Restriction **hasTopping some Mozzarella**. This restriction describes the class of individuals that have *at least one* topping that is **Mozzarella**

Existential and Universal Restrictions

- Existential restrictions describe classes of individuals that participate in *at least one* relationship along a specified property to individuals that are members of a specified class. For example, “the class of individuals that have *at least one* (some) **hasTopping** relationship to members of **MozzarellaTopping**”. In Protégé 4 the keyword ‘**some**’ is used to denote existential restrictions.⁵
- Universal restrictions describe classes of individuals that for a given property only have relationships along this property to individuals that are members of a specified class. For example, “the class of individuals that *only* have **hasTopping** relationships to members of **VegetableTopping**”. In Protégé 4 the keyword ‘**only**’ is used.⁶

Let’s take a closer look at the example of an existential restriction. The restriction **hasTopping some MozzarellaTopping** is an existential restriction (as indicated by the **some** keyword), which acts along the **hasTopping** property, and has a *filler* **MozzarellaTopping**. This restriction describes *the class* of individuals that have *at least one* **hasTopping** relationship to an individual that is a member of the class **MozzarellaTopping**. This restriction is depicted in Figure 4.30 — The diamonds in the Figure represent individuals. As can be seen from Figure 4.30, the restriction is a class which contains the individuals that satisfy the restriction.

MEANING



A restriction describes an *anonymous class* (an unnamed class). The anonymous 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.

The restrictions for a class are displayed and edited using the ‘**Class Description View**’ shown in Figure 4.31. The ‘**Class Description View**’ is the ‘heart of’ the ‘**Classes**’ tab in protege, and holds virtually all of the information used to describe a class. At first glance, the ‘**Class Description View**’ may seem complicated, however, it will become apparent that it is an incredibly powerful way of describing and defining classes.

⁵Existential restrictions may be denoted by the *existential quantifier* (\exists). They are also known as ‘someValuesFrom’ restrictions in OWL speak.

⁶Universal restrictions may be denoted by the *universal quantifier* (\forall), which can be read as *only*. They are also known as ‘allValuesFrom’ restrictions in OWL speak.

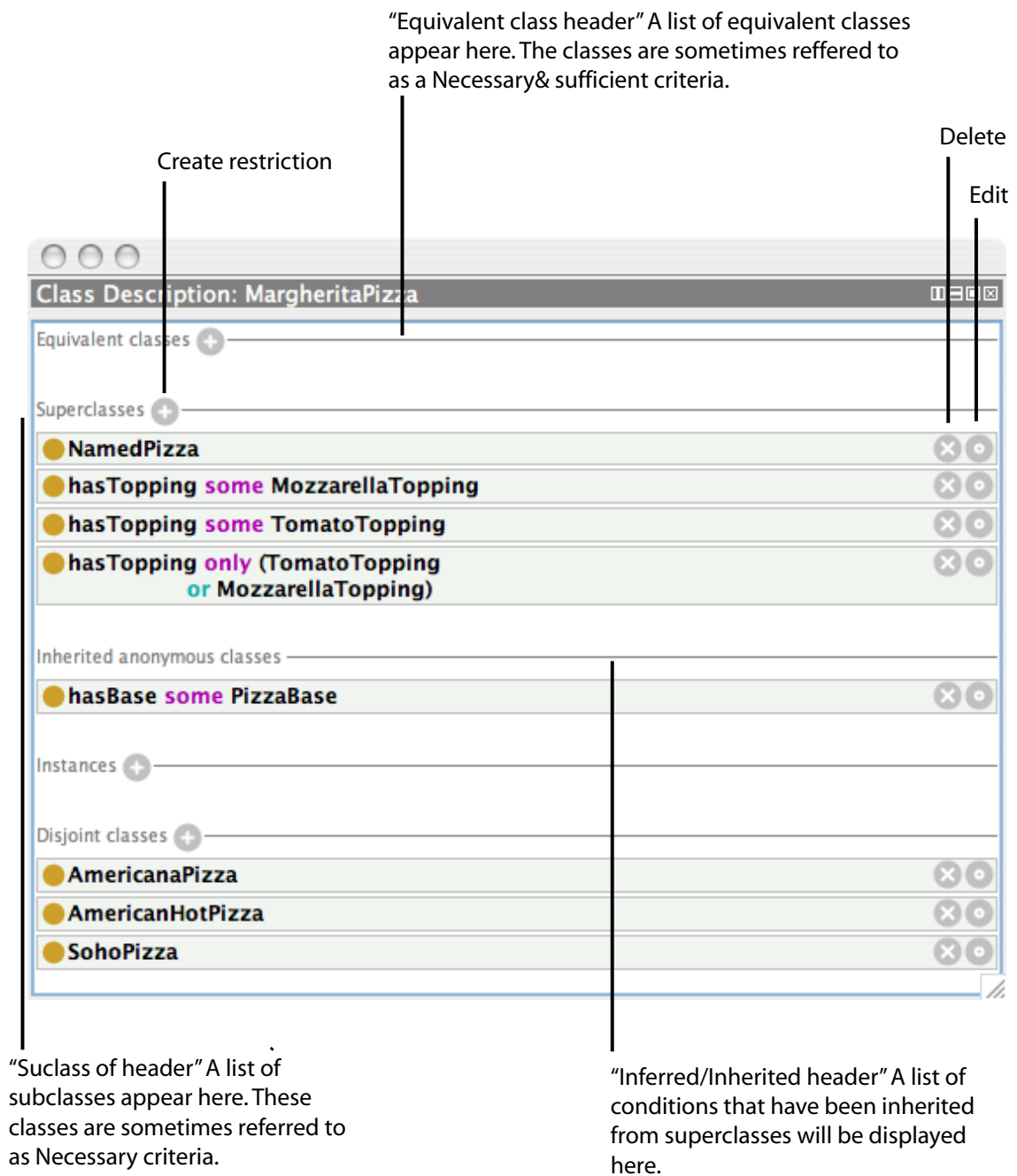


Figure 4.31: The Class Description View

Restrictions are used in OWL class descriptions to specify anonymous superclasses of the class being described.

4.8.2 Existential Restrictions

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

Vocabulary



Existential restrictions are also known as *Some Restrictions*, or as *some values from* restrictions.

TIP



Other tools, papers and presentations might write the restriction **hasBase some PizzaBase** as \exists **hasBase PizzaBase** — this alternative notation is known as DL Syntax (Description Logics Syntax), which is a more formal syntax.

Exercise 16: Add a restriction to Pizza that specifies a Pizza must have a PizzaBase

1. Select **Pizza** from the class hierarchy on the ‘**Classes**’ tab.
 2. Select the “Subclass of” header in the ‘**Class Description View**’ shown in Figure 4.32 in order to create a necessary condition.
 3. Press the ‘**Add Class**’ button shown in Figure 4.32. This will open a text box in the Class Description view where we can enter our restrictions as shown in Figure 4.33
-

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

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

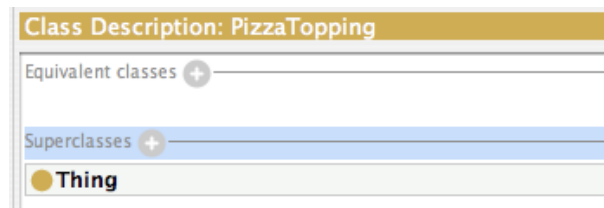


Figure 4.32: Creating a Necessary Restriction

Exercise 17: Add a restriction to Pizza that specifies a Pizza must have a PizzaBase (Continued...)

1. You can either drag and drop **hasBase** from the property list into the create restriction text box, or type it in.
2. Now add the type or restriction, we will use an existential restriction so type '**some**'.
3. Specify that the filler is **PizzaBase** — to do this either: type **PizzaBase** into the filler edit box, or drag and drop **PizzaBase** into the text box as show in Figure 4.33
4. Press '**Enter**' to create the restriction and close the create restriction text box. If all information was entered correctly the dialog will close and the restriction will be displayed in the '**Class Description View**'. If there were errors they will be underlined in red in the text box, o popup will give some hints to the cause of the error — if this is the case, recheck that the type of restriction, the property and filler have been specified correctly.

TIP

A very useful feature of the expression builder is the ability to 'auto complete' class names, property names and individual names. Auto completion is activated by pressing '**alt tab**' on the keyboard. In the above example if we had typed **Pi** into the inline expresion editor and pressed the tab key, the choices to complete the word **Pi** would have popped up in a list as shown in Figure 4.33. The up and down arrow keys could then have been used to select **PizzaBase** and pressing the Enter key would complete the word for us.

The class description view should now look similar to the picture shown in Figure 4.34.

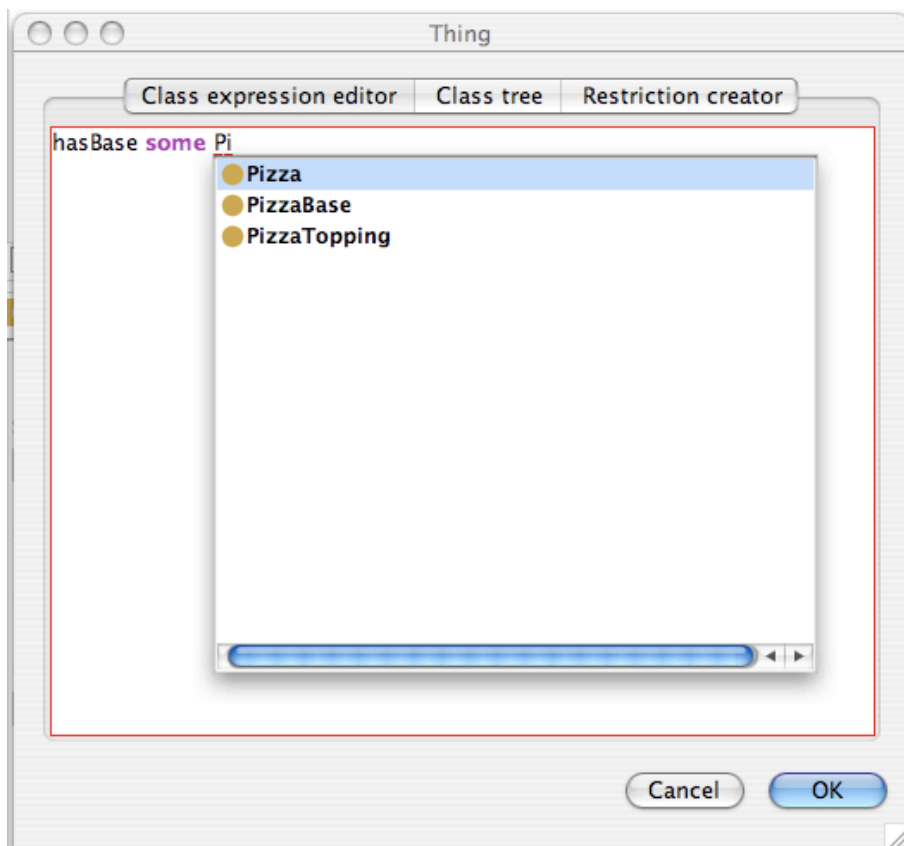


Figure 4.33: Creating a restriction in the text box, with auto-complete



Figure 4.34: class description view: Description of a Pizza

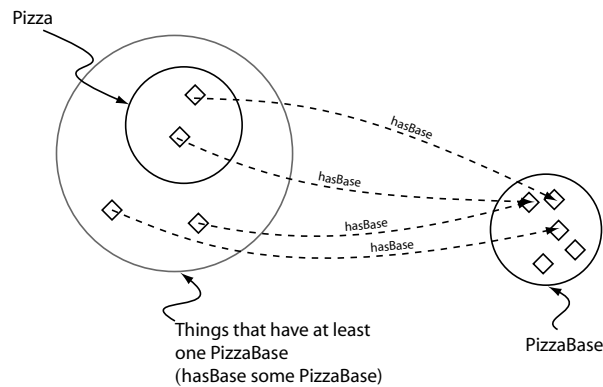


Figure 4.35: A Schematic Description of a **Pizza** — In order for something to be a **Pizza** it is *necessary* for it to have a (*at least one*) **PizzaBase** — A **Pizza** is a *subclass* of the things that have *at least one* **PizzaBase**

MEANING



We have described the class **Pizza** 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** — This is depicted in Figure 4.35.

MEANING



When restrictions are used to describe classes, they actually specify anonymous superclasses of the class being described. For example, we could say that **MargheritaPizza** is a subclass of, amongst other things, **Pizza** and also a subclass of the things that have *at least one* topping that is **MozzarellaTopping**.

Creating Some Different Kinds Of Pizzas

It's now time to add some different kinds of pizzas to our ontology. We will start off by adding a 'MargheritaPizza', which is a pizza that has toppings of mozzarella and tomato. In order to keep our

ontology tidy, we will group our different pizzas under the class ‘NamedPizza’:

Exercise 18: Create a subclass of Pizza called NamedPizza, and a subclass of NamedPizza called MargheritaPizza

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

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

Exercise 19: Create an existential (some) restriction on MargheritaPizza that acts along the property hasTopping with a filler of MozzarellaTopping to specify that a MargheritaPizza has at least one MozzarellaTopping

1. Make sure that **MargheritaPizza** is selected in the class hierarchy.
 2. Select the “Subclass Of” header in the ‘**Class Description View**’, as we want to create and add a necessary condition.
 3. Use the ‘**Add Class**’ button on the ‘**Class Description view**’ (Figure 4.31) to open a text box.
 4. Type **hasTopping** as the property to be restricted in the text box.
 5. Type ‘**some**’ to create the existential restriction.
 6. Type the class **MozzarellaTopping** as the filler for the restriction — remember that this can be achieved by typing the class name **MozzarellaTopping** into the filler edit box, or by using drag and drop from the class hierarchy.
 7. Press ‘**Enter**’ to create the restriction — if there are any errors, the restriction will not be created, and the error will be highlighted in red.
-

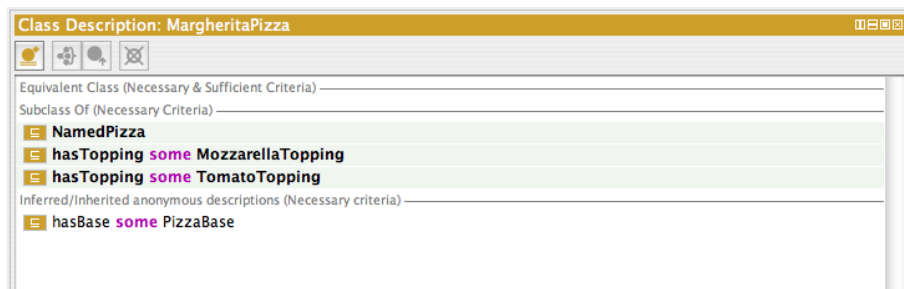


Figure 4.36: The Class Description View Showing A Description Of A MargheritaPizza

Now specify that **MargheritaPizzas** also have **TomatoTopping**.

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

1. Ensure that **MargheritaPizza** is selected in the class hierarchy.
2. Select the “Subclass Of” header in the ‘**Class Description View**’, as we want to create and add a necessary condition.
3. Use the ‘**Add class**’ button on the ‘**Class Description View**’ (Figure 4.31) to display open the text box.
4. Type **hasTopping** as the property to be restricted.
5. Type ‘**some**’ to create the existential restriction.
6. Type the class **TomatoTopping** as the filler for the restriction.
7. Click ‘**Enter**’ to create restriction dialog to create the restriction.

The ‘**Class Description View**’ should now look similar to the picture shown in Figure 4.36.

MEANING



We have added restrictions to **MargheritaPizza** to say that a **MargheritaPizza** is a **NamedPizza** that has at least one kind of **MozzarellaTopping** and at least one kind of **TomatoTopping**.

More formally (reading the class description view line by line), if something is a member of the class **MargheritaPizza** it is *necessary* for it to be a member of the class **NamedPizza** *and* it is *necessary* for it to be a member of the anonymous class of things that are linked to at least one member of the class **MozzarellaTopping** via the property **hasTopping**, *and* it is *necessary* for it to be a member of the anonymous class of things that are linked to at least one member of the class **TomatoTopping** via the property **hasTopping**.

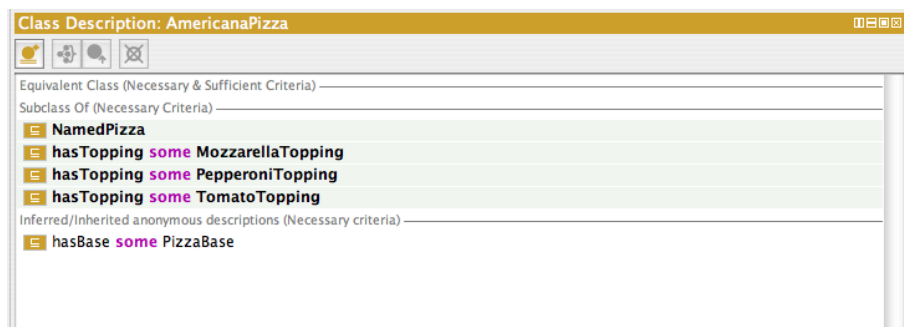


Figure 4.37: The Class Description View displaying the description for **AmericanaPizza**

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

Exercise 21: Create AmericanaPizza by cloning and modifying the description of MargheritaPizza

1. Select the class **MargheritaPizza** in the class hierarchy on the Classes tab.
2. Select 'Duplicate selected class' from the 'Edit' menu. This will create a copy of the class **MargheritaPizza** named **MargheritaPizza.2**, that has exactly the same conditions (restrictions etc.) as **MargheritaPizza**.
3. Rename the **MargheritaPizza.2** to **AmericanaPizza** using the 'Rename...' option 'Refactor' menu.
4. Ensuring that **AmericanaPizza** is still selected, select the "Subclass of" header in the class description view, as we want to add a new restriction to the necessary conditions for **AmericanaPizza**.
5. Press the 'Add class' button on the class description view to open a text box.
6. Type the property **hasTopping** as the property to be restricted.
7. Type 'some' to create the existential restriction.
8. Specify the restriction filler as the class **PepperoniTopping** by either typing **PepperoniTopping** into the text box, or by using drag and drop from the class hierarchy.
9. Press **Enter** to create the restriction.

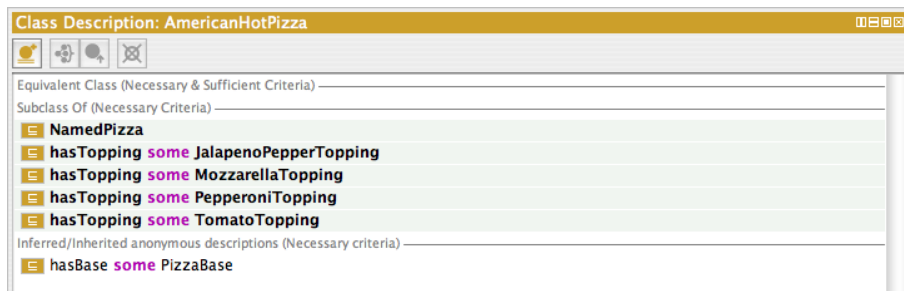


Figure 4.38: The Class Description View displaying the description for **AmericanHotPizza**

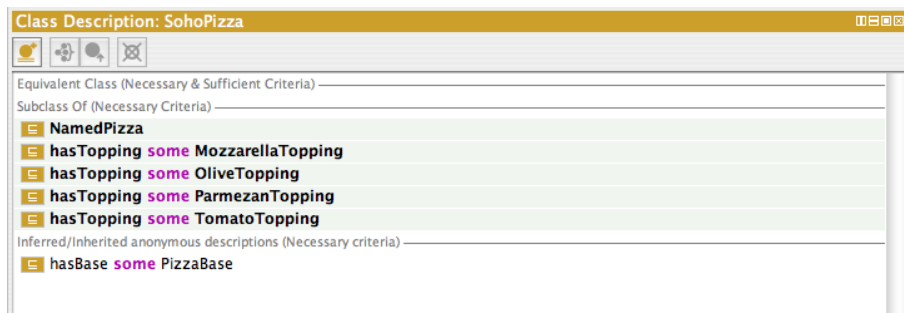


Figure 4.39: The Class Description View displaying the description for **SohoPizza**

The ‘**Class Description View**’ should now look like the picture shown in Figure 4.37.


Exercise 22: Create an **AmericanHotPizza** and a **SohoPizza**

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

For **AmericanHot** pizza the class description view should now look like the picture shown in Figure 4.38.
For **SohoPizza** the class description view should now look like the picture shown in 4.39.

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

Exercise 23: Make subclasses of NamedPizza disjoint from each other

- 
1. Select the class **MargheritaPizza** in the class hierarchy on the ‘**Classes**’ tab.
 2. Press the ‘**Add all siblings**’ button on the ‘**Disjoints view**’ to make the pizzas disjoint from each other.
-

4.9 Using A Reasoner

4.9.1 Determining the OWL Sub-Language

As mentioned in section 3.1, OWL comes in three flavours (or sub-languages): OWL-Lite, OWL-DL (DL stands for Description Logics) and OWL-Full. The exact definitions of these sub-languages can be found in the OWL Overview, which is available on the World Wide Web Consortium website⁷. Protégé 4 features a *species validation facility*, which is able to determine the sub-language of the ontology being edited. To use the species validation facility, use the ‘**Determine OWL Species...**’ option on the ‘**Tools menu**’. This will report the sub-language of the ontology.

One of the key features of ontologies that are described using OWL-DL is that they can be processed by a *reasoner*. One of the main services offered by a reasoner is to test whether or not one class is a subclass of another class⁸. By performing such tests on the classes in an ontology it is possible for a reasoner to compute the *inferred* ontology class hierarchy. Another standard service that is offered by reasoners is *consistency* checking. Based on the description (conditions) of a class the reasoner can check whether or not it is possible for the class to have any instances. A class is deemed to be inconsistent if it cannot possibly have any instances.

Vocabulary



Reasoners are also known as *classifiers*.

4.9.2 Invoking The Reasoner

Protégé 4 allows different OWL reasoners to be plugged in, the reasoner shipped with Protégé is called Fact++. The ontology can be ‘sent to the reasoner’ to automatically compute the classification hierarchy, and also to check the logical consistency of the ontology. In Protégé 4 the ‘manually constructed’ class hierarchy is called the *asserted hierarchy*. The class hierarchy that is automatically computed by the reasoner is called the *inferred hierarchy*. To automatically classify the ontology (and check for inconsistencies) the ‘**Classify...**’ action should be used. This can be invoked via the ‘**Classify...**’ button in the Reasoner drop down menu shown in Figure 4.40. When the inferred hierarchy has been computed,

⁷<http://www.w3.org/TR/owl-features/>

⁸Known as subsumption testing — the descriptions of the classes (conditions) are used to determine if a super-class/subclass relationship exists between them.

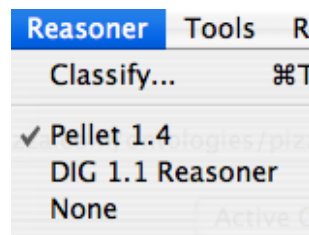


Figure 4.40: Classify the ontology from the reasoner menu

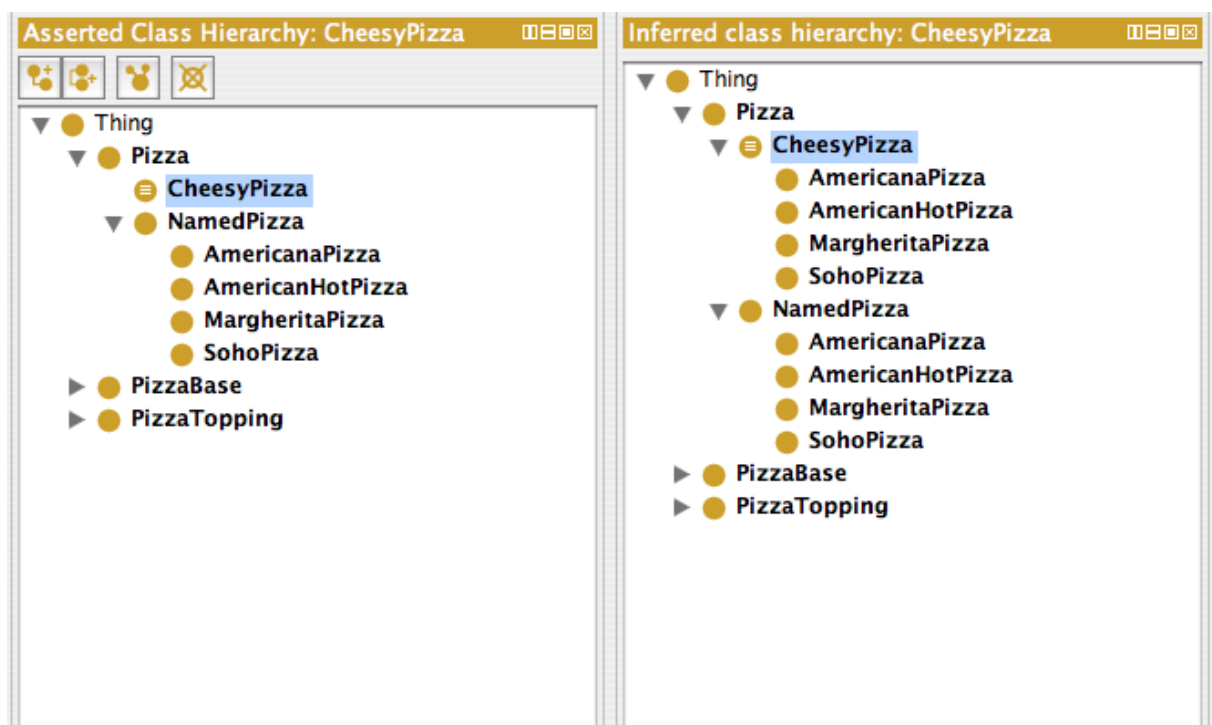


Figure 4.41: The Inferred Hierarchy Pane alongside the Asserted Hierarchy Pane after classification has taken place. Note the inferred subclasses of CheesyPizza

an *inferred hierarchy* window will pop open on top the existing *asserted hierarchy* window as shown in Figure 4.41. If a class has been reclassified (i.e. if its superclasses have changed) then the class name will appear in a blue colour in the *inferred hierarchy*. If a class has been found to be inconsistent it's icon will be highlighted in red.



The task of computing the inferred class hierarchy is also known as *classifying the ontology*.

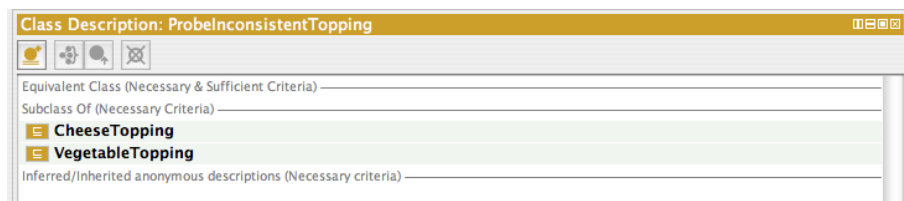


Figure 4.42: The Class Description View Displaying ProbInconsistentTopping

4.9.3 Inconsistent Classes

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

Exercise 24: Add a Probe Class called ProbInconsistentTopping which is a subclass of both CheeseTopping and VegetableTopping

1. Select the class **CheeseTopping** from the class hierarchy on the **Classes** tab.
2. Create a subclass of **CheeseTopping** named **ProbInconsistentTopping**.
3. Add a comment to the **ProbInconsistentTopping** class that is something along the lines of, "This class should be inconsistent when the ontology is classified.". This will enable anyone who looks at our pizza ontology to see that we deliberately meant the class to be inconsistent.
4. Ensure that the **ProbInconsistentTopping** class is selected in the class hierarchy, and then select the "Subclass Of" header in the '**Class Description View**'.
5. Click on the '**superclasses**' button on the '**Class Description View**'. This will display a dialog containing the class hierarchy from which a class may be selected. Select the class **VegetableTopping** and then press the OK button. The class **VegetableTopping** will be added as a superclass, so that the class description view should look like the picture in Figure 4.42.

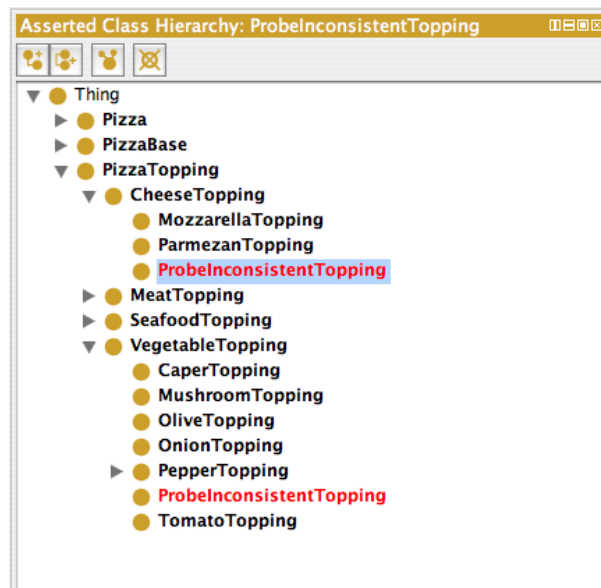


Figure 4.43: The Class `ProbelInconsistentTopping` found to be inconsistent by the reasoner

MEANING



If we study the class hierarchy, `ProbelInconsistentTopping` should appear as a subclass of `CheeseTopping` and as a subclass of `VegetableTopping`. This means that `ProbelInconsistentTopping` is a `CheeseTopping` *and* a `VegetableTopping`. More formally, all individuals that are members of the class `ProbelInconsistentTopping` are also (necessarily) members of the class `CheeseTopping` and (necessarily) members of the class `VegetableTopping`. Intuitively this is incorrect since something can not simultaneously be both cheese and a vegetable!

Exercise 25: Classify the ontology to make sure `ProbelInconsistentTopping` is inconsistent

1. Press the ‘**Classify...**’ button on the Reasoner drop down menu to classify the ontology.

After a few seconds the inferred hierarchy will have been computed and the *inferred hierarchy* window will pop open (if it was previously closed). The hierarchy should resemble that shown in Figure 4.43 — notice that the class `ProbelInconsistentTopping` is highlighted in red, indicating that the reasoner has found this class to be inconsistent (i.e. it cannot possibly have any individuals as members).

MEANING



Why did this happen? Intuitively we know something cannot at the same time be both cheese and a vegetable. Something should not be both an instance of **CheeseTopping** *and* an instance of **VegetableTopping**. However, it must be remembered that we have chosen the names for our classes. As far as the reasoner is concerned names have no meaning. The reasoner cannot determine that something is inconsistent based on names. The actual reason that **ProbeInconsistentTopping** has been detected to be inconsistent is because its superclasses **VegetableTopping** and **CheeseTopping** are *disjoint from each other* — remember that earlier on we specified that the four categories of topping were disjoint from each other. Therefore, individuals that are members of the class **CheeseTopping** cannot be members of the class **VegetableTopping** and vice-versa.

TIP



To close the inferred hierarchy use the small white cross on a red background button on the top right of the inferred hierarchy window.

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

1. Select the class **CheeseTopping** using the class hierarchy.
2. The ‘**Disjoints view**’ should contain **CheeseTopping**’s sibling classes: **VegetableTopping**, **SeafoodTopping** and **MeatTopping**. Select **VegetableTopping** in the Disjoints view.
3. Press the ‘**Delete selected row**’ button on the Disjoints view (shown in Figure ??) to remove the disjoint axiom that states **CheeseTopping** and **VegetableTopping** are disjoint.
4. Press ‘**Classify...**’ on the Reasoner drop down menu to send the ontology to the reasoner. After a few seconds the ontology should have been classified and the results displayed.

MEANING



It should be noticeable that **ProbElInconsistentTopping** is no longer inconsistent! This means that individuals which are members of the class **ProbElInconsistentTopping** are also members of the class **CheeseTopping** and **VegetableTopping** — something can be both cheese and a vegetable!

This clearly illustrates the importance of the careful use of disjoint axioms in OWL. OWL classes ‘overlap’ until they have been stated to be disjoint from each other. If certain classes are not disjoint from each other then unexpected results can arise. Accordingly, if certain classes have been incorrectly made disjoint from each other then this can also give rise to unexpected results.

Exercise 27: Fix the ontology by making **CheeseTopping** and **Vegetable** disjoint from each other

1. Select the class **CheeseTopping** using the class hierarchy.
2. The ‘**Disjoints view**’ should contain **MeatTopping** and **SeafoodTopping**.
3. Press the ‘**Add disjoint class**’ button on the disjoint classes view to display a dialog which classes may be picked from. Select the class **VegetableTopping** and press the OK button. **CheeseTopping** should once again be disjoint from **VegetableTopping**.
4. Test that the disjoint axiom has been added correctly — Press ‘**Classify...**’ on the Reasoner drop down menu to send the ontology to the reasoner. After a few seconds the ontology should have been classified, and **ProbElInconsistentTopping** should be highlighted in red indicating that it is once again inconsistent.

4.10 Necessary And Sufficient Conditions (Primitive and Defined Classes)

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

Vocabulary



A class that only has *necessary* conditions is known as a **Primitive Class**.

Let’s illustrate this with an example. We will create a subclass of **Pizza** called **CheesyPizza**, which will

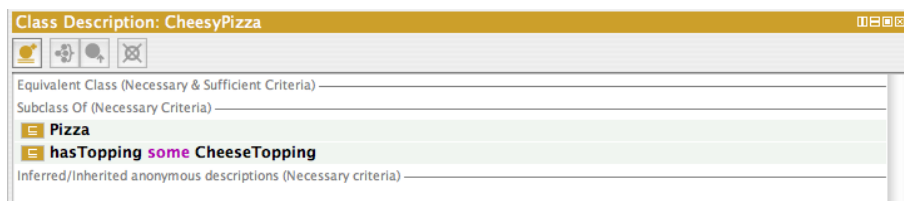


Figure 4.44: The Description of CheesyPizza (Using Necessary Conditions)

be a **Pizza** that has at least one kind of **CheeseTopping**.

Exercise 28: Create a subclass of Pizza called CheesyPizza and specify that it has at least one topping that is a kind of CheeseTopping

1. Select **Pizza** in the class hierarchy on the 'Classes' tab.
2. Press the 'Add subclass' button to create a subclass of **Pizza**. Name it **CheesyPizza**.
3. Make sure that **CheesyPizza** is selected in the class hierarchy. Select the "Subclass of" header in the class description view.
4. Press the 'Add class' button on the class description view to open the restriction text box.
5. Type **hasTopping** as the property to be restricted.
6. Type '**some**' to create the existential restriction.
7. Finally type **CheeseTopping** or drag the class from the class hierarchy. Press '**Enter**' to close the dialog and create the restriction.

The 'Class Description View' should now look like the picture shown in Figure 4.44.

MEANING



Our description of **CheesyPizza** states that if something is a member of the class **CheesyPizza** it is *necessary* for it to be a member of the class **Pizza** and it is *necessary* for it to have *at least one* topping that is a member of the class **CheeseTopping**.

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

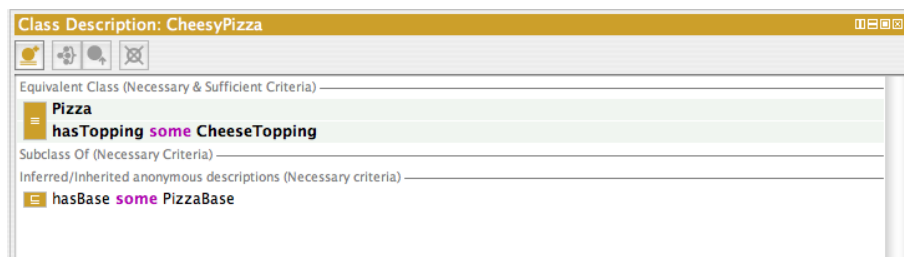


Figure 4.45: The Description of CheesyPizza (Using Necessary AND Sufficient Conditions)

the class CheesyPizza.

Vocabulary



A class that has at least one set of *necessary and sufficient* conditions is known as a **Defined Class**.

Vocabulary



Classes that only have necessary conditions are also known as ‘partial’ classes. Classes that have at least one set of necessary and sufficient conditions are also known as ‘complete’ classes.

In order to convert *necessary* conditions to *necessary and sufficient* conditions, the conditions must be moved from under the “NECESSARY” header in the class description view to be under the “NECESSARY AND SUFFICIENT” header. This can be accomplished by dragging and dropping the conditions one-by-one.

Exercise 29: Convert the necessary conditions for CheesyPizza into necessary & sufficient conditions

1. Ensure that CheesyPizza is selected in the class hierarchy.
2. On the ‘Class Description View’ select the hasTopping some CheeseTopping restriction.
3. Drag the hasTopping some CheeseTopping restriction from under the “Subclass of” header to *on top* of the “Equivalent class” header.
4. Select the class Pizza.
5. Drag the class Pizza from under the “Subclass of” header to *on top* of the hasTopping some CheeseTopping restriction (note *not* on top of the “Equivalent class” header this time).

The ‘Class Description View’ should now look like the picture shown in Figure 4.45.

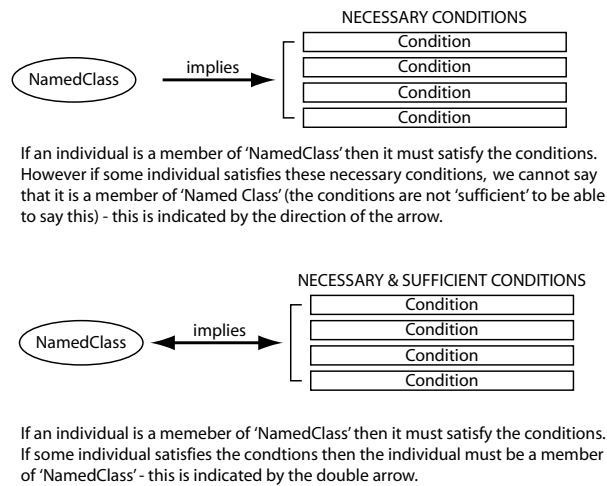


Figure 4.46: Necessary And Sufficient Conditions

MEANING



We have converted our description of **CheesyPizza** into a *definition*. If something is a **CheesyPizza** then it is *necessary* that it is a **Pizza** and it is also *necessary* that *at least one* topping that is a member of the class **CheeseTopping**. *Moreover*, if an individual is a member of the class **Pizza** and it has at least one topping that is a member of the class **CheeseTopping** then these conditions are *sufficient* to determine that the individual *must* be a member of the class **CheesyPizza**. The notion of *necessary* and *sufficient* conditions is illustrated in Figure 4.46.



If you accidentally dropped **Pizza** onto the “Equivalent class” header (rather than onto the **hasTopping some CheeseTopping**) in Exercise 29 the class description view will look like the picture shown in Figure 4.47. In this case, a new necessary and sufficient condition has been created, which is not what we want. To correct this mistake, drag **Pizza** on top of the **hasTopping some CheeseTopping** restriction.

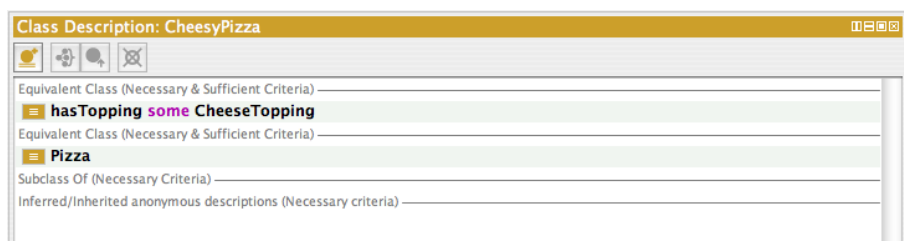


Figure 4.47: An INCORRECT description of CheesyPizza

TIP

Conditions may also be transferred from “Subclass of” to “Equivalent class” and vice versa using Cut and Paste. Right click (ctrl click on a Mac) on a condition and select Cut or Paste from the popup menu.

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

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



In OWL it is possible to have multiple sets of necessary & sufficient conditions. This is discussed later in section 7.5

4.10.1 Primitive And Defined Classes

Classes that have at least one set of necessary and sufficient conditions are known as *defined* classes — they have a definition, and any individual that satisfies the definition will belong to the class. Classes that do not have any sets of necessary and sufficient conditions (only have necessary conditions) are known as *primitive* classes. In Protégé 4 *defined* classes have a class icon with an *orange* background. *Primitive* classes have a class icon that has a *yellow* background. It is also important to understand that the reasoner can only automatically classify classes under *defined* classes - i.e. classes with at least one set of necessary and sufficient conditions.

4.11 Automatic Classification

Being able to use a reasoner to automatically compute the class hierarchy is one of the major benefits of building an ontology using the OWL-DL sub-language. Indeed, when constructing very large ontologies (with upwards of several thousand classes in them) the use of a reasoner to compute subclass-superclass relationships between classes becomes almost vital. Without a reasoner it is very difficult to keep large ontologies in a maintainable and logically correct state. In cases where ontologies can have classes that have many superclasses (multiple inheritance) it is nearly always a good idea to construct the class hierarchy as a simple tree. Classes in the asserted hierarchy (manually constructed hierarchy) therefore have no more than one superclass. Computing and maintaining multiple inheritance is the job of the reasoner. This technique⁹ helps to keep the ontology in a maintainable and modular state. Not only

⁹Sometimes known as ontology normalisation.

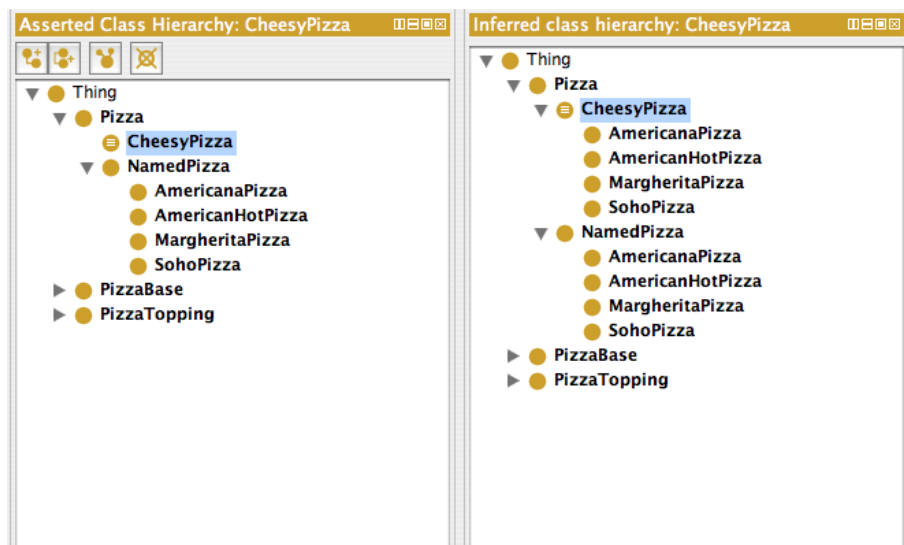


Figure 4.48: The Asserted and Inferred Hierarchies Displaying The Classification Results For **CheesyPizza**

does this promote the reuse of the ontology by other ontologies and applications, it also minimises human errors that are inherent in maintaining a multiple inheritance hierarchy.

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

Exercise 30: Use the reasoner to automatically compute the subclasses of **CheesyPizza**

1. Press the 'Classify...' button on the Reasoner drop down menu (See Figure 4.40).

After a few seconds the inferred hierarchy should have been computed and the inferred hierarchy window will pop open (if it was previously closed). The inferred hierarchy should appear similar to the picture shown in Figure 4.48. Figures 4.49 and 4.50 show the OWLViz display of the asserted and inferred hierarchies respectively. Notice that classes which have had their superclasses changed by the reasoner are shown in blue.

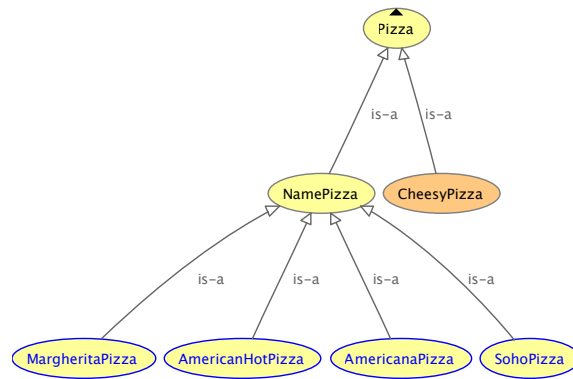


Figure 4.49: OWLViz Displaying the Asserted Hierarchy for CheesyPizza

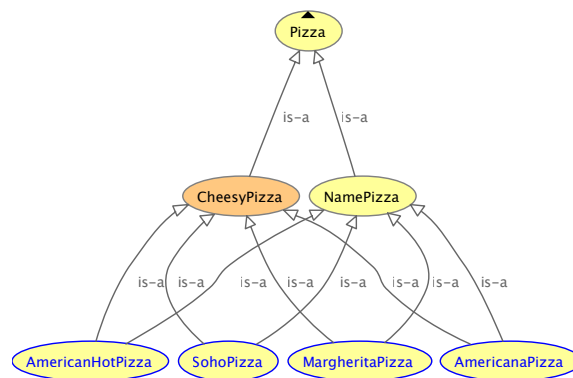


Figure 4.50: OWLViz Displaying the Inferred Hierarchy for CheesyPizza

MEANING



The reasoner has determined that **MargheritaPizza**, **AmericanaPizza**, **AmericanHotPizza** and **SohoPizza** are subclasses of **CheesyPizza**. This is because we *defined* **CheesyPizza** using necessary and sufficient conditions. Any individual that is a **Pizza** and has *at least one* topping that is a **CheeseTopping** is a member of the class **CheesyPizza**. Due to the fact that all of the individuals that are described by the classes **MargheritaPizza**, **AmericanaPizza**, **AmericanHotPizza** and **SohoPizza** are **Pizzas** and they have *at least one* topping that is a **CheeseTopping**^a the reasoner has determined that these classes must be subclasses of **CheeseTopping**.

^aOr toppings that belong to the subclasses of **CheeseTopping**



It is important to realise that, in general, classes will never be placed as subclasses of *primitive* classes (i.e. classes that only have necessary conditions) by the reasoner^a.

^aThe exception to this is when a property has a domain that is a primitive class. This can *coerce* classes to be reclassified under the primitive class that is the domain of the property — the use of property domains to cause such effects is strongly discouraged.

4.11.1 Classification Results

After the reasoner has been invoked, computed superclass-subclass relationships and inconsistent classes are displayed in the ‘**Classification Results**’ pane shown in Figure 4.51. The ‘**Classification Results**’ pane pops open after classification at the bottom of the Protégé application window. The ‘spanner icon’ on the left hand side of the pane is the ‘**Assert Selected Change(s)**’ button. Pressing this button takes the superclass-subclass relationships that have been found by the reasoner and puts them into the asserted (manually constructed) hierarchy. For example, if the ‘**Assert Selected Changes**’ button was pressed with the selection shown in Figure 4.51, **CheesyPizza** would be added as a superclass of **AmericanaPizza**.



Despite that fact that this facility exists, it is generally considered a bad idea to put computed/inferred relationships into the ‘manually constructed’ or asserted model whilst an ontology is being developed — we therefore advise against using this button during the development of an ontology.

4.12 Universal Restrictions

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

Reasoner results:	
	Class
● ProbInconsistentTopping	Inconsistent
● AmericanaPizza	Added CheesyPizza
● AmericanHotPizza	Added CheesyPizza
● MargheritaPizza	Added CheesyPizza
● SohoPizza	Added CheesyPizza

Figure 4.51: The Classification Results Pane

the specified filler class.

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

Universal restrictions are given the symbol \forall . They *constrain* the relationships along a given property to individuals that are members of a specific class. For example the universal restriction \forall **hasTopping MozzarellaTopping** describes the individuals all of whose **hasTopping** relationships are to members of the class **MozzarellaTopping** — the individuals do not have a **hasTopping** relationships to individuals that aren't members of the class **MozzarellaTopping**.

Vocabulary



Universal restrictions are also know as *AllValuesFrom Restrictions*.



The above universal restriction \forall **hasTopping MozzarellaTopping** also describes the individuals that *do not participate in any* **hasTopping** relationships. An individual that does not participate in any **hasTopping** relationships what so ever, by definition does not have any **hasTopping** relationships to individuals that aren't members of the class **MozzarellaTopping** and the restriction is therefore satisfied.



For a given property, universal restrictions do *not* specify the existence of a relationship. They merely state that *if* a relationship exists for the property then it must be to individuals that are members of a specific class.

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

restriction.

Exercise 31: Create a class to describe a VegetarianPizza

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

^aSee section ?? for more information about union classes.

At this point the class description view should look like the picture shown in Figure 4.52.

MEANING



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* (\forall 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.

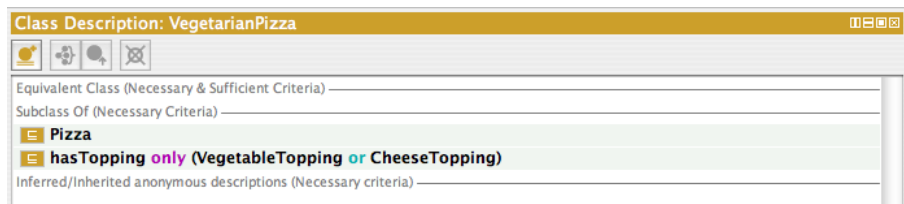


Figure 4.52: The Description of VegetarianPizza (Using Necessary Conditions)



In situations like the above example, a common mistake is to use an *intersection* instead of a *union*. For example, `CheeseTopping \sqcap VegetableTopping`. This reads, `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 obviously incorrect as demonstrated in section 4.9.3. If the classes `CheeseTopping` and `VegetableTopping` were not disjoint, this would have been a logically legitimate thing to say – it would not be inconsistent and therefore would not be ‘spotted’ by the reasoner.



In the above example it might have been tempting to create two universal restrictions — one for `CheeseTopping` (\forall hasTopping `CheeseTopping`) and one for `VegetableTopping` (\forall 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.

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



Figure 4.53: The Class Description View Displaying the Definition of `VegetarianPizza` (Using Necessary and Sufficient Conditions)

the subclasses of `VegetarianPizza`.

Exercise 32: Convert the necessary conditions for `VegetarianPizza` into necessary & sufficient conditions

1. Ensure that `VegetarianPizza` is selected in the class hierarchy.
2. On the ‘Class Description View’ select the (universal) restriction on the `hasTopping` property.
3. Drag the `hasTopping` restriction from under the “Subclass of” header to *on top* of the “Equivalent class” header.
4. Select the class `Pizza`.
5. Drag the class `Pizza` from under the “Subclass of” header to *on top* of the `hasTopping` restriction (note *not* on top of the “Equivalent class” header this time).

The ‘Class Description View’ should now look like the picture shown in Figure 4.53.

MEANING



We have converted our description of `VegetarianPizza` into a *definition*. If something is a `VegetarianPizza`, then it is *necessary* that it is a `Pizza` and it is also *necessary* that *all* toppings belong to the class `CheeseTopping` or `VegetableTopping`. *Moreover*, if something is a member of the class `Pizza` and all of its toppings are members of the class `CheeseTopping` or the class `VegetableTopping` then these conditions are *sufficient* to recognise that it *must* be a member of the class `VegetarianPizza`. The notion of *necessary* and *sufficient* conditions is illustrated in Figure 4.46.

4.13 Automatic Classification and Open World Reasoning

We want to use the reasoner to automatically compute the superclass-subclass relationship (subsumption relationship) between `MargheritaPizza` and `VegetarianPizza` and also, `SohoPizza` and `VegetarianPizza`. Recall that we believe that `MargheritaPizza` and `SohoPizza` should be vegetarian pizzas (they should

be subclasses of **VegetarianPizza**). This is because they have toppings that are essentially vegetarian toppings — by our definition, vegetarian toppings are members of the classes **CheeseTopping** or **VegetableTopping** and their subclasses. Having previously created a definition for **VegetarianPizza** (using a set of *necessary and sufficient* conditions) we can use the reasoner to perform automatic classification and determine the vegetarian pizzas in our ontology.

Exercise 33: Use the reasoner to classify the ontology



1. Press the ‘**Classify...**’ button in the Reasoner Drop Down menu.
-

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

Reasoning in OWL (Description Logics) is based on what is known as the *open world assumption* (OWA). It is frequently referred to as *open world reasoning* (OWR). The *open world assumption* means that we cannot assume something doesn’t exist until it is explicitly stated that it does not exist. In other words, because something hasn’t been stated to be true, it cannot be assumed to be false — it is assumed that ‘the knowledge just hasn’t been added to the knowledge base’. In the case of our pizza ontology, we have stated that **MargheritaPizza** has toppings that are kinds of **MozzarellaTopping** and also kinds of **TomatoTopping**. Because of the *open world assumption*, until we explicitly say that a **MargheritaPizza** *only* has these kinds of toppings, it is assumed (by the reasoner) that a **MargheritaPizza** could have other toppings. To specify explicitly that a **MargheritaPizza** has toppings that are kinds of **MozzarellaTopping** or kinds of **MargheritaTopping** and *only* kinds of **MozzarellaTopping** or **MargheritaTopping**, we must add what is known as a *closure axiom*¹⁰ on the **hasTopping** property.

4.13.1 Closure Axioms

A *closure axiom* on a property consists of a universal restriction that acts along the property to say that it can *only* be filled by the specified fillers. The restriction has a filler that is the *union* of the fillers that occur in the existential restrictions for the property¹¹. For example, the closure axiom on the **hasTopping** property for **MargheritaPizza** is a universal restriction that acts along the **hasTopping** property, with a filler that is the *union* of **MozzarellaTopping** and also **TomatoTopping**. i.e. $\forall \text{ hasTopping } (\text{MozzarellaTopping} \sqcup \text{TomatoTopping})$

¹⁰ Also referred to as a closure restriction.

¹¹ And technically speaking the classes for the values used in any **hasValue** restrictions (see later).

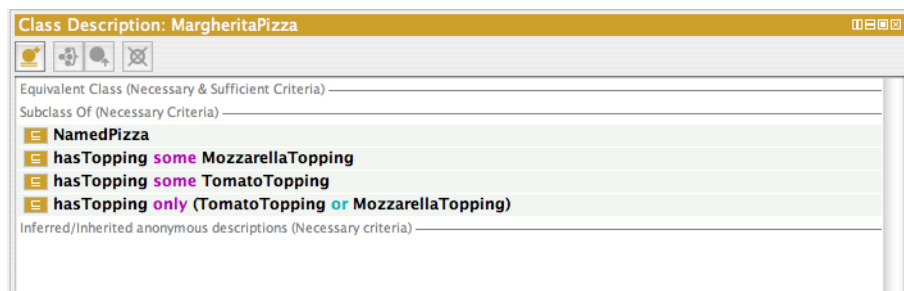


Figure 4.54: Class Description View: Margherita Pizza With a Closure Axiom for the hasTopping property

laTopping \sqsubseteq TomatoTopping).

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

1. Make sure that **MargheritaPizza** is selected in the class hierarchy on the ‘Classes’ tab.
2. Select the “Subclass of” header in the ‘**Class Description View**’.
3. Press the ‘**Add class**’ button on the class description view to display open the edit text box.
4. Type **hasTopping** as the property to be restricted.
5. Type ‘**only**’ to create the universal restriction.
6. Open brackets and type **MozzarellaTopping** or **TomatoTopping** close bracket.
7. Press Enter to create the restriction and add it to the class **MargheritaPizza**.

The class description view should now appear as shown in Figure 4.54.

MEANING



This now says that if an individual is a member of the class **MargeritaPizza** then it must be a member of the class **Pizza**, *and* it must have at least one topping that is a kind of **MozzarellaTopping** *and* it must have at least one topping that is a member of the class **TomatoTopping** *and* the toppings must *only* be kinds of **MozzarellaTopping** *or* **TomatoTopping**.



A common error in situations such as above is to only use universal restrictions in descriptions. For example, describing a **MargheritaPizza** by making it a subclass of **Pizza** and then only using \forall **hasTopping** (**MozzarellaTopping** \sqcup **TomatoTopping**) without any existential restrictions. However, because of the semantics of the universal restriction, this actually means either: things that are **Pizzas** and only have toppings that are **MozzarellaTopping** or **TomatoTopping**, OR, things that are **Pizzas** and *do not have any* toppings at all.

Exercise 35: Add a closure axiom on the **hasTopping** property for **SohoPizza**

1. Make sure that **SohoPizza** is selected in the class hierarchy on the ‘**Classes**’ tab.
2. Select the “Subclass of” header in the ‘**Class Description View**’.
3. Press the ‘**Add class**’ button on the class description view to display open the edit text box.
4. Type **hasTopping** as the property to be restricted.
5. Type ‘**only**’ to create the universal restriction.
6. Open brackets and type **MozzarellaTopping** or **TomatoTopping** or **ParmesanTopping** close bracket.
7. Press Enter to create the restriction and add it to the class **SohoPizza**.

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

Fortunately Protégé 4 has the capability of creating closure axioms for us.

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

1. Select **AmericanaPizza** in the class hierarchy on the Classes tab.
 2. In the ‘**Class Description View**’ select one of the existing hasTopping restrictions so its all highlighted. Now simply click the add closure axiom button from the class description view see figure 4.31. A closure restriction (universal restriction) will be created along the hasTopping property, which contains the union of the existential hasTopping fillers.
-

Exercise 37: Automatically create a closure axiom on the hasTopping property for AmericanHot-Pizza

1. Select **AmericanHotPizza** in the class hierarchy on the Classes tab.
 2. In the ‘**Class Description View**’ right click (Ctrl click on the Mac) on one of the *existential* hasTopping restrictions. Select ‘**Add closure axiom**’ from the pop up menu that appears.
-

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

Exercise 38: Use the reasoner to classify the ontology

1. Press the ‘**Classify...**’ button on the reasoner drop down menu. to invoke the reasoner.
-

After a short amount of time the ontology will have been classified and the ‘**Inferred Hierarchy**’ pane will pop open (if it is not already open). This time, **MargheritaPizza** and also **SohoPizza** will have been classified as subclasses of **VegetarianPizza**. This has happened because we specifically ‘closed’ the hasTopping property on our pizzas to say *exactly* what toppings they have and **VegetarianPizza** was *defined* to be a **Pizza** with only kinds of **CheeseTopping** and only kinds of **VegetableTopping**. Figure 4.55 shows the current asserted and inferred hierarchies. It is clear to see that the asserted hierarchy is simpler and ‘cleaner’ than the ‘tangled’ inferred hierarchy. Although the ontology is only very simple at this stage, it should be becoming clear that the use of a reasoner can help (especially in the case of large ontologies) to maintain a multiple inheritance hierarchy for us.

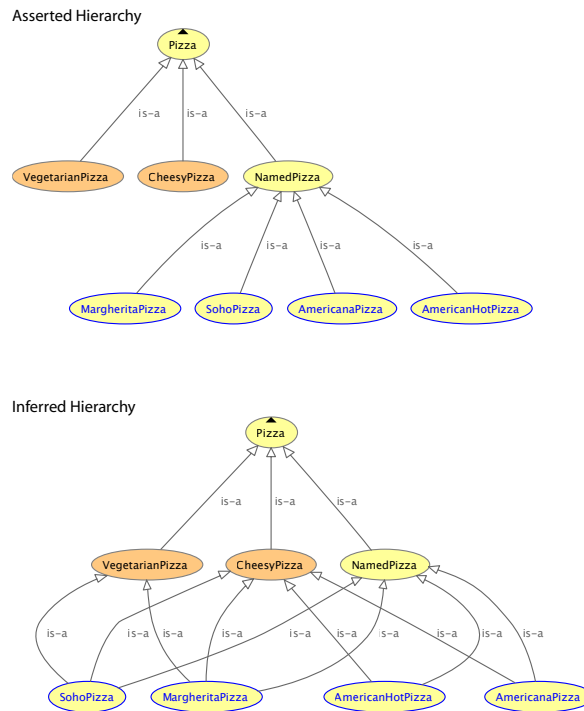


Figure 4.55: The asserted and inferred hierarchies showing the “before and after” classification of **Pizzas** into **CheesyPizzas** and **VegetarianPizzas**.

4.14 Value Partitions

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

1. Create a class to represent the ValuePartition. For example to represent a ‘spiciness’ ValuePartition we might create the class **SpicinessValuePartition**.
2. Create subclasses of the ValuePartition to represent the possible options for the ValuePartition. For example we might create the classes **Mild**, **Medium** and **Hot** as subclasses of the **SpicinessValuePartition** class.
3. Make the subclasses of the ValuePartition class disjoint.
4. Provide a *covering axiom* to make the list of value types *exhaustive* (see below).
5. Create an object property for the ValuePartition. For example, for our spiciness ValuePartition, we might create the property **hasSpiciness**.

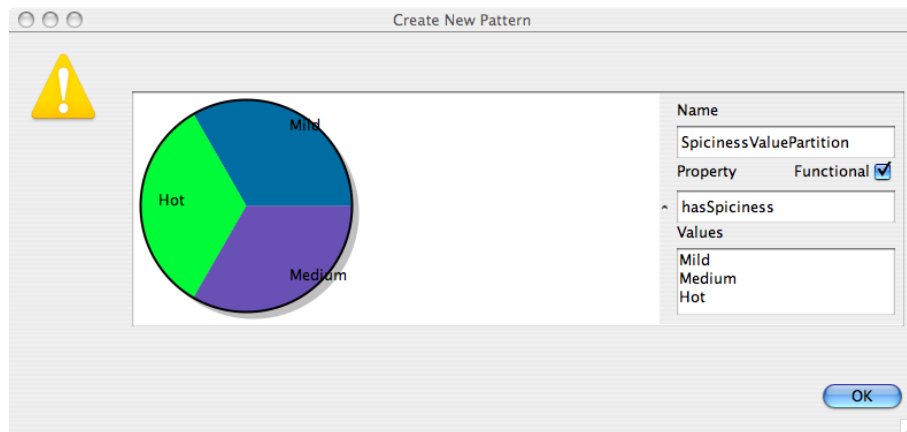


Figure 4.56: Patterns plugin tab

6. Make the property *functional*.
7. Set the range of the property as the ValuePartition class. For example for the **hasSpiciness** property the range would be set to **SpicinessValuePartition**.

It should be relatively clear that due to the number of steps and the complexity of some of the steps, it would be quite easy to make a mistake. It could also take a significant amount of time to create more than a few ValuePartitions. Fortunately, the OWL plugins package contains a plugin for creating ValuePartitions.

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

Exercise 39: Create a ValuePartition to represent the spiciness of pizza toppings

1. You will need the Patterns tab, if the plugin is already installed you can open the Patterns tab in the Protégé menu under Tabs drop down.
2. Select the **'Create new partition'** button a pop up window will appear. Type in **SpicinessValuePartition** for the name.
3. Now enter **hasSpiciness** for the ValuePartition property name, we want to make this property functional so check the tick box.
4. We now need to specify the values for the value type. In the text area type **Mild** and press return, type **Medium** and press return, and type **Hot**. Your window should look like figure 4.56 .
5. Press **'OK'** and you will return to the patterns tab. You will see a pie chart depicting the SpicinessValuePartition you just created.

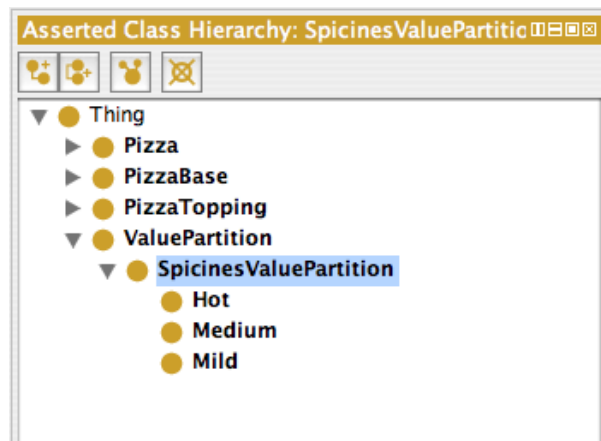


Figure 4.57: Classes Added by the ‘Create ValuePartition’ Wizard

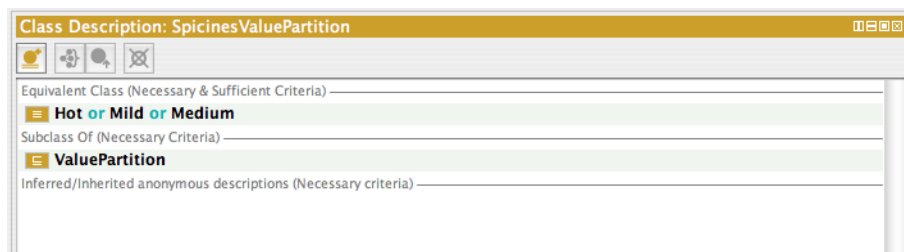


Figure 4.58: The Class Description View Displaying the Description of the `SpicinessValuePartition` Class

Let’s look at what the wizard has done for us (refer to Figure 4.57 and Figure 4.58):

1. A `ValuePartition` class has been created as a subclass of `Thing`. All value partitions are created under a `ValuePartition` Class by default.
2. A `SpicinessValuePartition` class has been created as a subclass of `ValuePartition`.
3. The classes `Mild`, `Medium`, `Hot` have been created as subclasses of `SpicinessValuePartition`.
4. The classes `Mild`, `Medium` and `Hot` have been made disjoint from each other.
5. A class that is the *union* of `Mild`, `Medium` and `Hot` has been created as the subclass of `SpicinessValuePartition` (see Figure 4.58).
6. A `hasSpiciness` object property has been created.
7. The `hasSpiciness` property has been made *functional*
8. `SpicinessValuePartition` has been set as the range of the `hasSpiciness` property.

4.14.1 Covering Axioms

As part of the ValuePartition pattern we use a *covering axiom*. A covering axiom consists of two parts: The class that is being ‘covered’, and the classes that form the covering. For example, suppose we have

three classes **A**, **B** and **C**. Classes **B** and **C** are subclasses of class **A**. Now suppose that we have a covering axiom that specifies class **A** is *covered* by class **B** and also class **C**. This means that a member of class **A** *must* be a member of **B** and/or **C**. If classes **B** and **C** are disjoint then a member of class **A** *must* be a member of *either* class **B** *or* class **C**. Remember that ordinarily, although **B** and **C** are subclasses of **A** an individual may be a member of **A** without being a member of either **B** or **C**.

In Protégé 4 a covering axiom manifests itself as a class that is the *union* of the classes being covered, which forms a superclass of the class that is being covered. In the case of classes **A**, **B** and **C**, class **A** would have a superclass of $B \sqcup C$. The effect of a covering axiom is depicted in Figure 4.59.

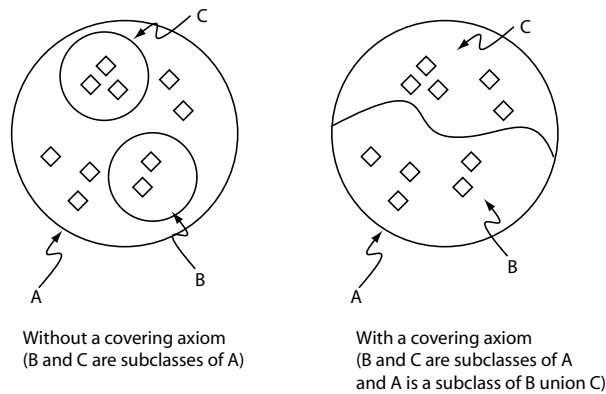


Figure 4.59: A schematic diagram that shows the effect of using a Covering Axiom to cover class **A** with classes **B** and **C**

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

The difference between not using a covering axiom, and using a covering axiom is depicted in Figure 4.60. In both cases the classes **Mild**, **Medium** and **Hot** are disjoint — they do not overlap. It can be seen that in the case without a covering axiom an individual may be a member of the class **SpicinessValuePartition** and still not be a member of **Mild**, **Medium** or **Hot** — **SpicinessValuePartition** is *not* covered by **Mild**, **Medium** and **Hot**. Contrast this with the case when a covering axiom *is* used. It can be seen that if an individual is a member of the class **SpicinessValuePartition**, it *must* be a member of one of the three subclasses **Mild**, **Medium** or **Hot** — **SpicinessValuePartition** is *covered* by **Mild**, **Medium** and **Hot**.

4.15 Using the Matrix Wizard

We can now use the **SpicinessValuePartition** to describe the spiciness of our pizza toppings. To do this we will add an existential restriction to each kind of **PizzaTopping** to state it's spiciness. Restrictions will take the form, **hasSpiciness** some **SpicinessValuePartition**, where **SpicinessValuePartition** will be one of **Mild**, **Medium** or **Hot**. As we have over twenty toppings in our pizza ontology this could take rather a long time. Fortunately, the *Matrix Plugin* can help to speed things up. The matrix plugin can be used

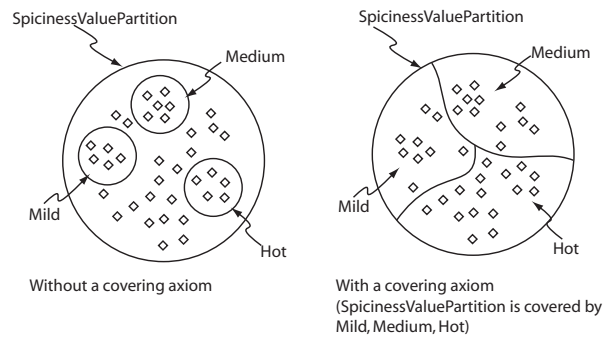


Figure 4.60: The effect of using a covering axiom on the `SpicinessValuePartition`

to add existential restrictions along specified properties to many classes in a quick and efficient manner.

Exercise 40: Use the matrix plugin to specify the spiciness of pizza toppings

1. If you already have the matrix plugin installed you can find it under the ‘**Tabs**’ drop down in the Protégé menu bar. Open the matrix tab.
2. In the Matrix Tab select the ‘**Class Existential Restrictions**’ view as shown in figure 4.61. You should be able to see your class hierarchy as well as the property view.
3. Create a new property called `hasSpiciness` in the ‘**Object Properties**’ view.
4. Drag and drop the new `hasSpiciness` property into the centre of the ‘**Class Existential Restriction**’ view. You should see a new column created with the title `hasSpiciness`.
5. For each pizza topping in the class hierarchy enter a spiciness value as shown in Figure 4.62. Remember you can use the auto completion to help fill in the values for you.
6. Return to the ‘**Classes**’ tab and browse

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

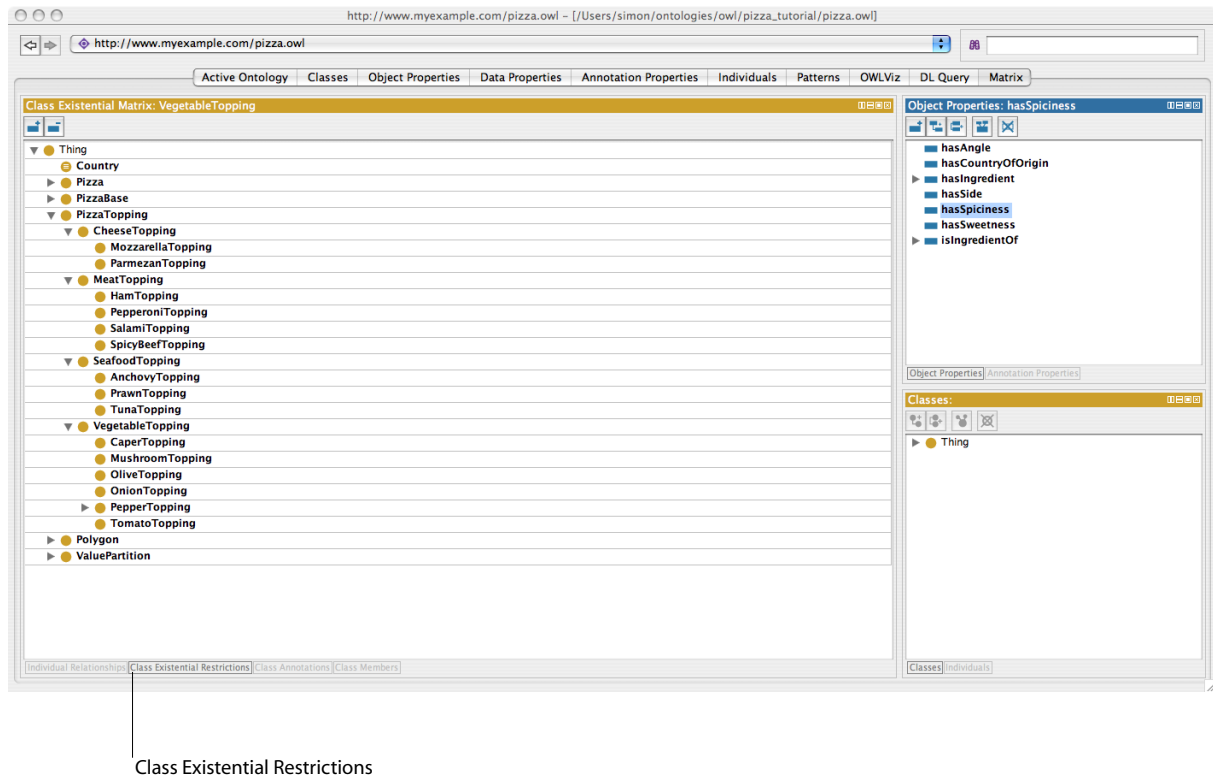


Figure 4.61: Property Matrix Wizard: Class Selection Page

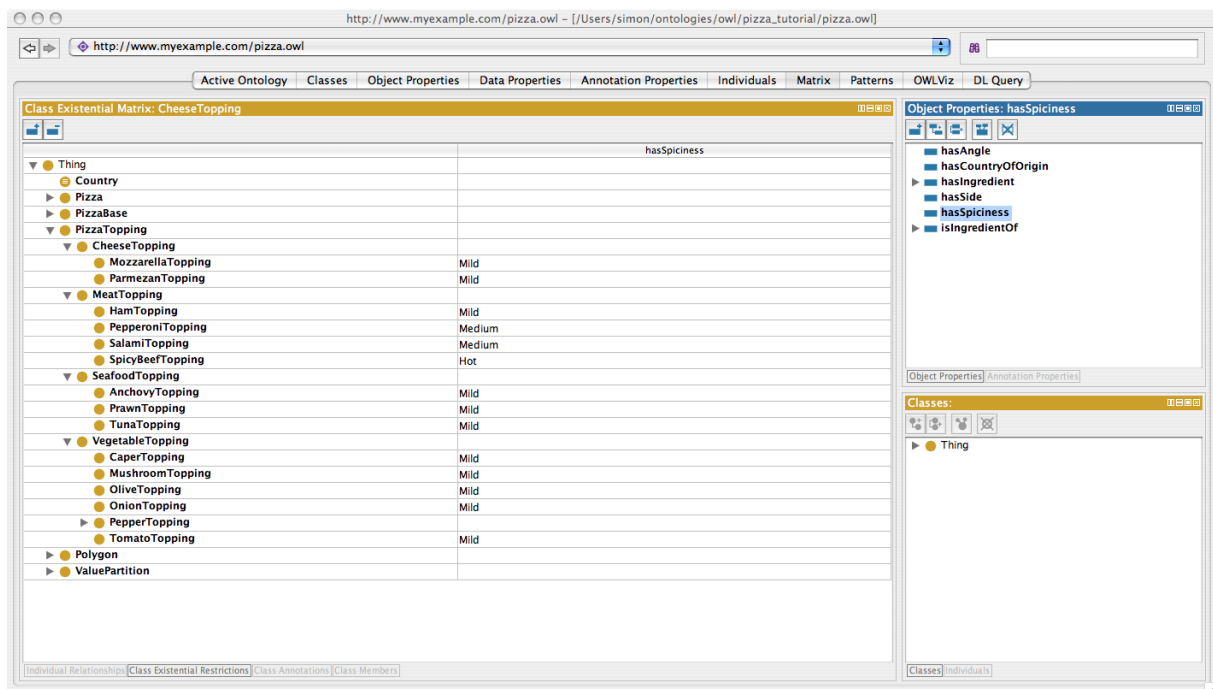


Figure 4.62: Property Matrix Wizard: Property Selection Page

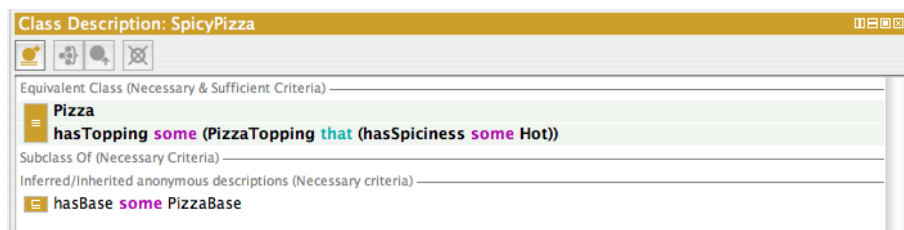


Figure 4.63: The definition of SpicyPizza

has a restriction on the `hasSpiciness` property as its filler.

Exercise 41: Create a SpicyPizza as a subclass of Pizza

1. Create a subclass of **Pizza** called **SpicyPizza**.
2. With **SpicyPizza** selected in the class hierarchy, select the “Equivalent Class” header in the class description view.
3. Press the ‘**Add class**’ button on the class description view to open a text box.
4. Type **hasTopping** as the property to be restricted.
5. Type ‘**some**’ as the type of restriction.
6. The filler should be: **PizzaTopping** and **hasSpiciness** some **Hot**. This filler describes an anonymous class, which contains the individuals that are members of the class **PizzaTopping** and also members of the class of individuals that are related to the members of class **Hot** via the **hasSpiciness** property. In other words, the things that are **PizzaToppings** and have a spiciness that is **Hot**. To create this restriction in the text box type, ‘(PizzaTopping and (hasSpiciness some Hot))’, including the brackets.
7. Finally, drag **Pizza** from under the “Subclass Of” header to on top of the newly created restriction.

The class description view should now look like the picture shown in Figure 4.63

MEANING



Our description of a **SpicyPizza** above says that all members of **SpicyPizza** are **Pizzas** and have at least one topping that has a Spiciness of **Hot**. It also says that *anything* that is a **Pizza** and has *at least* one topping that has a spiciness of **Hot** is a **SpicyPizza**.



NOTE

In the final step of Exercise 41 we created a restriction that had the *class expression* (**PizzaTopping** and **hasSpiciness some Hot**) rather than a named class as its filler. This filler was made up of an intersection between the named class **PizzaTopping** and the restriction **hasSpiciness some Hot**. Another way to do this would have been to create a subclass of **PizzaTopping** called **HotPizzaTopping** and define it to be a hot topping by having a necessary condition of **hasSpiciness some Hot**. We could have then used **hasTopping some HotPizzaTopping** in our definition of **SpicyPizza**. Although this alternative way is simpler, it is more verbose. OWL allows us to essentially shorten class descriptions and definitions by using class expressions in place of named classes as in the above example.

We should now be able to invoke the reasoner and determine the spicy pizzas in our ontology.

Exercise 42: Use the reasoner to classify the ontology

1. Press ‘**Classify...**’ in the Reasoner drop down menu to invoke the reasoner and classify the ontology.

After the reasoner has finished, the ‘**Inferred Hierarchy**’ class pane will pop open, and you should find that **AmericanHotPizza** has been classified as a subclass of **SpicyPizza** — the reasoner has automatically computed that any individual that is a member of **AmericanHotPizza** is also a member of **SpicyPizza**.

4.16 Cardinality Restrictions

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

Relationships (for example between two individuals) are only counted as separate relationships if it can be determined that the individuals that are the *fillers* for the relationships are *different* to each other. For example, Figure 4.64 depicts the individual **Matthew** related to the individuals **Nick** and the individual **Hai** via the **worksWith** property. The individual **Matthew** satisfies a *minimum cardinality* restriction of

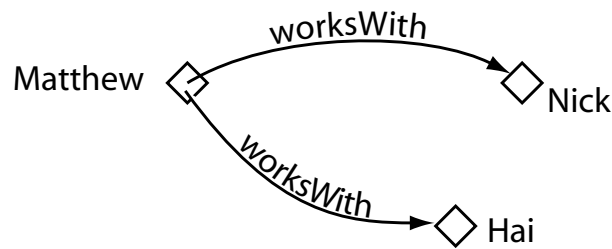


Figure 4.64: Cardinality Restrictions: Counting Relationships

2 along the **worksWith** property if the individuals **Nick** and **Hai** are distinct individuals i.e. they are different individuals.

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

Exercise 43: Create an InterestingPizza that has at least three toppings

1. Switch to the **Classes** tab and make sure that the **Pizza** class is selected.
 2. Create a subclass of **Pizza** called **InterestingPizza**.
 3. Select the "Equivalent class" header in the class description view.
 4. Press the '**Add class**' button to open a text box.
 5. Type **hasTopping** as a property to be restricted.
 6. Type '**min**' to create a minimum cardinality restriction.
 7. Specify a minimum cardinality of three by typing **3** into the text box.
 8. Press the '**Enter**' to close the dialog and create the restriction.
 9. The class description view should now have a "Subclass of" condition of **Pizza**, and a "Equivalent class" condition of **hasTopping min 3**. We need to make **Pizza** part of the necessary and sufficient conditions. Drag **Pizza** and drop it *on top of* the **hasTopping min 3** condition.
-

The class description view should now appear like the picture shown in Figure 4.65.

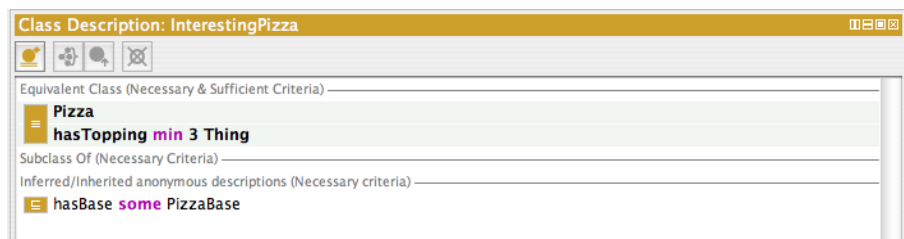


Figure 4.65: The Class Description View Displaying the Description of an InterestingPizza

MEANING



What does this mean? Our definition of an **InterestingPizza** describes the set of individuals that are members of the class **Pizza** *and* that have *at least* three **hasTopping** relationships with other (distinct) individuals.

Exercise 44: Use the reasoner to classify the ontology

1. Press ‘**Classify...**’ in the Reasoner drop down menu.

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

4.17 Qualified Cardinality Restrictions

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

Figure 4.66: Describing a FourCheesePizza using a Qualified Cardinality Restriction

cheese toppings.

Exercise 45: Create a Four Cheese Pizza that has exactly four cheese toppings (Figure 4.66)

1. Switch to the Classes tab and make sure the NamedPizza class is selected
 2. Create a subclass of Pizza called FourCheesePizza
 3. Select the 'Superclasses' header in the class description view
 4. Press the '+' button to open a text box
 5. Type **hasTopping** for the property
 6. Type **exactly** to create an exact cardinality restriction
 7. Specify a QCR of four by typing *4* into the text box
 8. Type **CheeseTopping** to specify the type of topping
 9. Click OK and create the restriction
-

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

Chapter 5

Datatype Properties

In Section 4.4 of Chapter 4 we introduced properties in OWL, but only described object properties—that is, relationships between individuals. In this chapter we will discuss and show examples of Datatype properties. Datatype properties link an individual to an XML Schema Datatype value or an rdf literal. In other words, they describe relationships between an individual and data values. Most of the property characteristics described in Chapter 4 cannot be used with datatype properties. We will describe those characteristics of properties that are applicable to data properties later in this chapter.


Datatype properties can be created using the ‘Datatype Properties’ tab shown in Figure 5.1.

We will use datatype properties to describe different sizes of pizza - small, medium and large. A large pizza will be defined as a pizza that has a diameter greater than or equal to 12 inches; a medium pizza less than 12 inches, but more than or equal to 7 inches; and a small pizza is less than 7 inches in diameter. In order to do this we need to complete the following steps:

- Create a datatype property **hasDiameter**, which will be used to restrict the size of a pizza.
- Create four classes describing different sizes of pizza.

Now let us do this in Protégé .

Exercise 46: Create a datatype property called hasDiameter

- 
1. Switch to the ‘**Datatype Properties**’ tab. Use the ‘**Add Datatype Property**’ button to create a new Datatype property.
 2. Name the property to **hasDiameter** using the ‘**Property Name Dialog**’ that pops up (The ‘**Property Name Dialog**’).
-

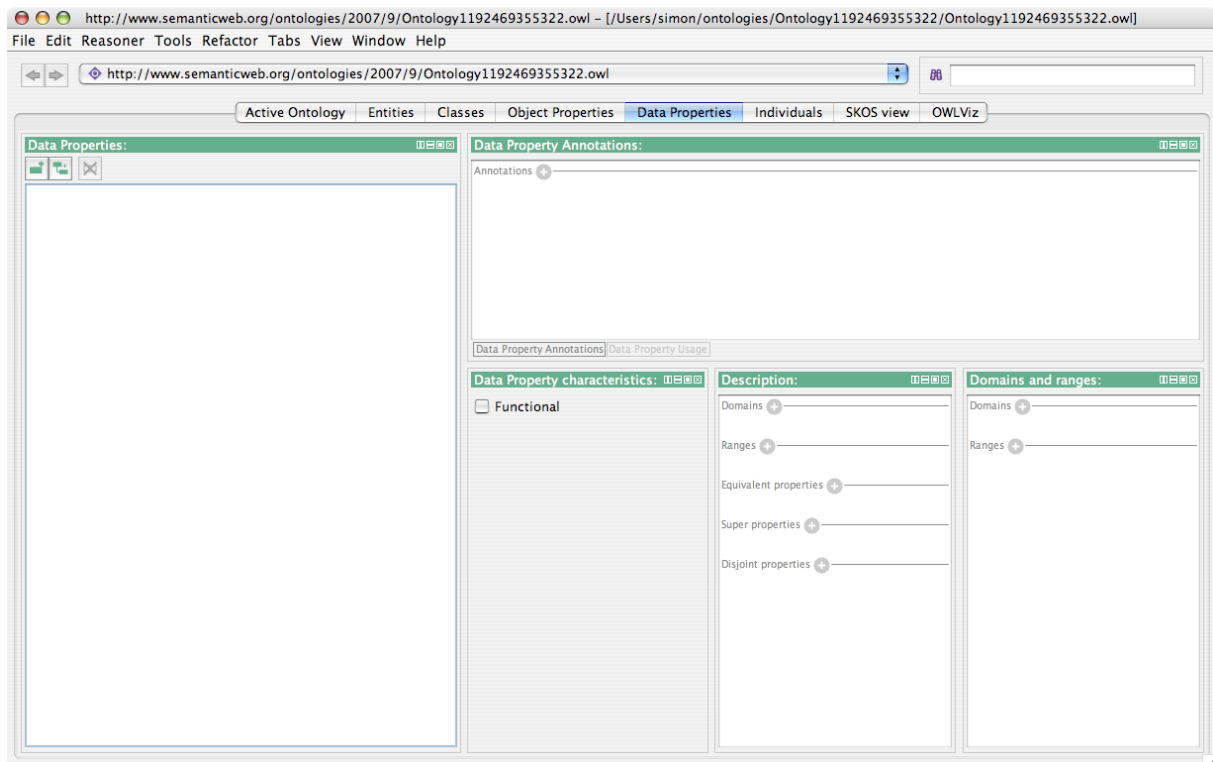


Figure 5.1: A snapshot of the Datatype Properties tab in Protégé

Exercise 47: Create classes `PizzaSize`, `LargePizzaSize`, `MediumPizzaSize` and `SmallPizzaSize`

1. Ensure the 'Classes Tab' is selected
2. Press the 'Add subclass' button and create the `PizzaSize` class
3. Make the `LargePizzaSize`, `MediumPizzaSize` and `SmallPizzaSize` classes, subclasses of the `PizzaSize` class

Using the datatype property, we can add a restriction to the class `LargePizzaSize` that states that all individuals of the `LargePizzaSize` class must have at least one relationship along the `hasDiameter`

property to a integer value greater than or equal to 12.

Exercise 48: Add a restriction to LargePizzaSize that specifies a LargePizzaSize must have a diameter greater than 12 (inches)

1. Ensure the ‘**Classes Tab**’ is selected
 2. Select LargePizzaSize in the class hierarchy
 3. On the Equivalent Classes Header click the ‘+’ button to add a new restriction
 4. In the class expression editor, type ‘**PizzaSize that hasDiameter some int[>= 12]**’ and click ‘**OK**’
-

MEANING

We have described a **LargePizzaSize** to be a subclass of **Food** and it is also necessary for it to have a **Diameter** greater than 12. Moreover, if an individual is a member of the class **LargePizzaSize** and it has at least a diameter that is greater than 12 then this is sufficient to determine that the individual must be a member of the class **LargePizzaSize**.

Finally, think about how many sizes can be held by an individual pizza. More precisely, ask yourself how many diameters are held by any one pizza? Of course, the answer is only one. Remember that there is a property characteristic that states that the property with that characteristic can only be held by an individual once. By describing a object property as *functional* it is being said that an individual can only have that property once. So, if we say that **Pizza hasBase PizzaBase** and make **hasBase** functional, we are saying that any instance of pizza can only ever have one **hasBase** property.

We can also use the functional characteristic on data properties. This is the only characteristic it is possible to use on data properties. By making **hasDiameter** functional we are saying that any one pizza individual can only ever have one **hasDiameter** property and hence only one diameter.

Exercise 49: Making the hasDiameter datatype property functional

1. Go to the ‘**Datatype Properties**’ tab and select **hasDiameter**
 2. In the ‘**Data Type Characteristics**’ pane, click the ‘**functional**’ radio button.
-

Creating some Pizzas of Different Sizes

Now let us create some different sized pizzas in our ontology. We will start by adding a **‘LargePizza’** and describing it using the classes (created above) and creating a new object property **hasPizzaSize**.

Exercise 50: Create a subclass of Pizza called LargePizza and a new Object Property hasPizzaSize

1. Select the class **Pizza** from the class hierarchy on the **‘Classes’** tab
2. Using the **‘Add subclass’** button, create a class **LargePizza**
3. Move to the **‘Object Properties’** tab and create a new property called **hasPizzaSize**

We will now use the new property to add an existential restriction to the **LargePizza** class.

Exercise 51: Add an existential (some) restriction on LargePizza that acts along the property hasPizzaSize with a filler of LargePizzaSize to specify that a LargePizza has at least one PizzaSize that is a kind of LargePizzaSize

1. Select the class **‘Pizza’** from the class hierarchy on the classes tab
2. In the **‘Restriction creator’** tab,
3. On the Equivalent Classes Header click the **‘+’** button to add a new restriction
4. In the class expression editor, type **‘Pizza that’**
5. Type **‘hasPizzaSize’** has the property to be restricted
6. Type **‘some’** to create the existential restriction
7. Finally type **‘LargePizzaSize’** and click **‘OK’**

We have added restrictions to **LargePizza** to say that a **LargePizza** is a **Pizza** that has at least one relationship to **LargePizzaSize**, which is defined as having a diameter greater than 12. Now let us check the consistency of the ontology and the specified properties **hasDiameter** and **hasPizzaSize** by running the classifier.

Now create the classes **MediumPizza** and **SmallPizza** in the same way that we created the **LargePizza**. Having created the classes that represent **LargePizza**, **MediumPizza** and **SmallPizza**, compute classifications using the reasoner (Figure ??).

Chapter 6

More On Open World Reasoning

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

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

Exercise 52: Create `NonVegetarianPizza` as a subclass of `Pizza` and make it disjoint to `Vegetarian-Pizza`

1. Select `Pizza` in the class hierarchy on the ‘Classes’ tab. Press the ‘Add subclass’ button to create a new class as the subclass of `Pizza`.
 2. Name the new class `NonVegetarianPizza`.
 3. Make `NonVegetarianPizza` disjoint with `VegetarianPizza` — while `NonVegetarian-Pizza` is selected, press the ‘Add disjoint class’ button on the disjoint classes view (Figure ??).
-

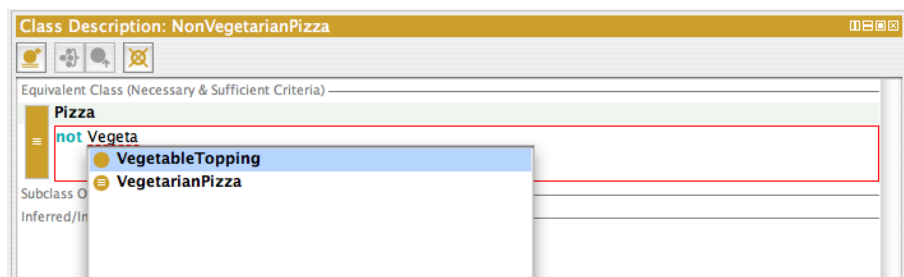


Figure 6.1: Class Description View: Inline Expression Editor Auto Completion

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

Exercise 53: Make `VegetarianPizza` the complement of `VegetarianPizza`

1. Make sure that `NonVegetarianPizza` is selected in the class hierarchy on the ‘**Classes tab**’.
2. Select the “Equivalent class” header in the ‘**Class Description View**’.
3. Press the ‘**Add class**’ button, and in the text box that appears type not `VegetarianPizza`.
4. Press the return key to create and assign the expression. If everything was entered correctly then the inline expression editor will close and the the expression will have been created. (If there are errors, check the spelling of “`VegetarianPizza`”).

TIP

A very useful feature of the expression editor is the ability to ‘auto complete’ class names, property names and individual names. The auto completion for the inline expression editor is activated using the tab key. In the above example if we had typed `Vege` into the inline expression editor and pressed the tab key, the choices to complete the word `Vege` would have popped up in a list as shown in Figure 6.1. The up and down arrow keys could then have been used to select `VegetarianPizza` and pressing the Enter key would complete the word for us.

The class description view should now resemble the picture shown in 6.2. However, we need to add `Pizza` to the *necessary and sufficient* conditions as at the moment our definition of `NonVegetarianPizza` says that an individual that is not a member of the class `VegetarianPizza` (everything else!) is a `NonVegetarianPizza`.

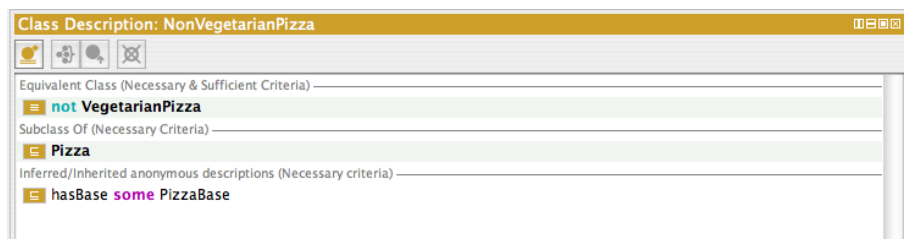


Figure 6.2: The Class Description View Displaying the Intermediate Step of Creating a Definition for NonVegetarianPizza

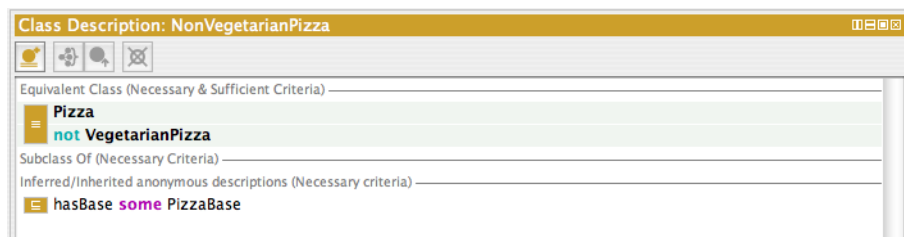


Figure 6.3: The Class Description View Displaying the Definition for NonVegetarianPizza

Exercise 54: Add Pizza to the necessary and sufficient conditions for NonVegetarianPizza

1. Make sure NonVegetarianPizza is selected in the class hierarchy on the 'Classes' tab.
2. Select Pizza in the 'Class Description View'.
3. Drag Pizza from under the "Subclass of" header, and drop it onto the 'not VegetarianPizza' condition to add it to the same set of *necessary and sufficient* conditions as not VegetarianPizza.

The 'Class Description View' should now look like the picture shown in Figure 6.3.

MEANING



The complement of a class includes all of the individuals that are not members of the class. By making **NonVegetarianPizza** a subclass of **Pizza** *and* the complement of **VegetarianPizza** we have stated that individuals that are **Pizzas** and are *not* members of **VegetarianPizza** must be members of **NonVegetarianPizza**. Note that we also made **VegetarianPizza** and **NonVegetarianPizza** disjoint so that if an individual is a member of **VegetarianPizza** it cannot be a member of **NonVegetarianPizza**.

Exercise 55: Use the reasoner to classify the ontology

1. Press the ‘**Classify...**’ button in the Reasoner toolbar. After a short time the reasoner will have computed the inferred class hierarchy, and the inferred class hierarchy pane will pop open.
-

The inferred class hierarchy should resemble the picture shown in Figure 6.4. As can be seen, **MargheritaPizza** and **SohoPizza** have been classified as subclasses of **VegetarianPizza**. **AmericanaPizza** and **AmericanHotPizza** have been classified as **NonVegetarianPizza**. Things *seemed* to have worked. However, let’s add a pizza that does not have a closure axiom on the **hasTopping** property.

Exercise 56: Create a subclass of NamedPizza with a topping of Mozzarella

1. Create a subclass of **NamedPizza** called **UnclosedPizza**.
 2. Making sure that **UnclosedPizza** is selected in the ‘**Class Description View**’ select the “Subclass of” header.
 3. Press the ‘**Add class**’ button to display restriction text box.
 4. Type **hasTopping** as the property to be restricted.
 5. Type ‘**some**’ in order to create an existential restriction.
 6. Type **MozzarellaTopping** into text box to specify that the toppings must be individuals that are members of the class **MozzarellaTopping**.
 7. Press ‘**Enter**’ to close the dialog and create the restriction.
-

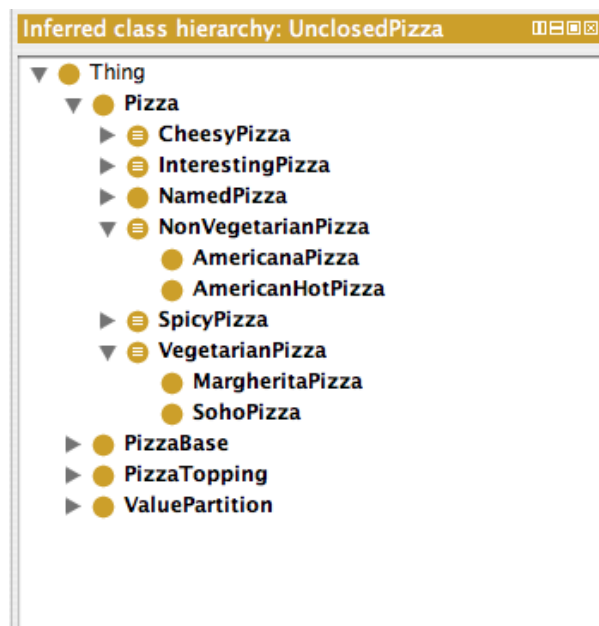


Figure 6.4: The Inferred Class Hierarchy Showing Inferred Subclasses of `VegetarianPizza` and `NonVegetarianPizza`

MEANING



If an individual is a member of `UnclosedPizza` it is necessary for it to be a `NamedPizza` and have *at least one* `hasTopping` relationship to an individual that is a member of the class `MozzarellaTopping`. Remember that because of the Open World Assumption and the fact that we have not added a closure axiom on the `hasTopping` property, an `UnclosedPizza` *might* have additional toppings that are not kinds of `MozzarellaTopping`.

Exercise 57: Use the reasoner to classify the ontology

1. Press '**Classify...**' in the Reasoner drop down menu.

Examine the class hierarchy. Notice that `UnclosedPizza` is neither a `VegetarianPizza` or `NonVegetarianPizza`.

MEANING



As expected (because of Open World Reasoning) **UnclosedPizza** has **not** been classified as a **VegetarianPizza**. The reasoner cannot determine **UnclosedPizza** is a **VegetarianPizza** because there is no closure axiom on the **hasTopping** and the pizza *might* have other toppings. We therefore might have expected **UnclosedPizza** to be classified as a **NonVegetarianPizza** since it has not been classified as a **VegetarianPizza**. However, Open World Reasoning does not dictate that because **UnclosedPizza** cannot be determined to be a **VegetarianPizza** it is *not* a **VegetarianPizza** — it *might* be a **VegetarianPizza** and also it *might not* be a **VegetarianPizza**! Hence, **UnclosedPizza** cannot be classified as a **NonVegetarianPizza**.

Chapter 7

Creating Other OWL Constructs In Protégé 4

This chapter discusses how to create some other owl constructs using Protégé 4 . These constructs are not part of the main tutorial and may be created in a new Protégé 4 project if desired.

7.1 Creating Individuals

OWL allows us to define individuals and to assert properties about them. Individuals can also be used in class descriptions, namely in hasValue restrictions and enumerated classes which will be explained in section 7.2 and section 7.3 respectively. To create individuals in Protégé 4 the '**Individuals Tab**' is used.

Suppose we wanted to describe the country of origin of various pizza toppings. We would first need to add various 'countries' to our ontology. Countries, for example, 'England', 'Italy', 'America', are typically thought of as being individuals (it would be incorrect to have a class **England** for example, as it's members would be deemed to be, 'things that are instances of **England**'). To create this in our Pizza Ontology we

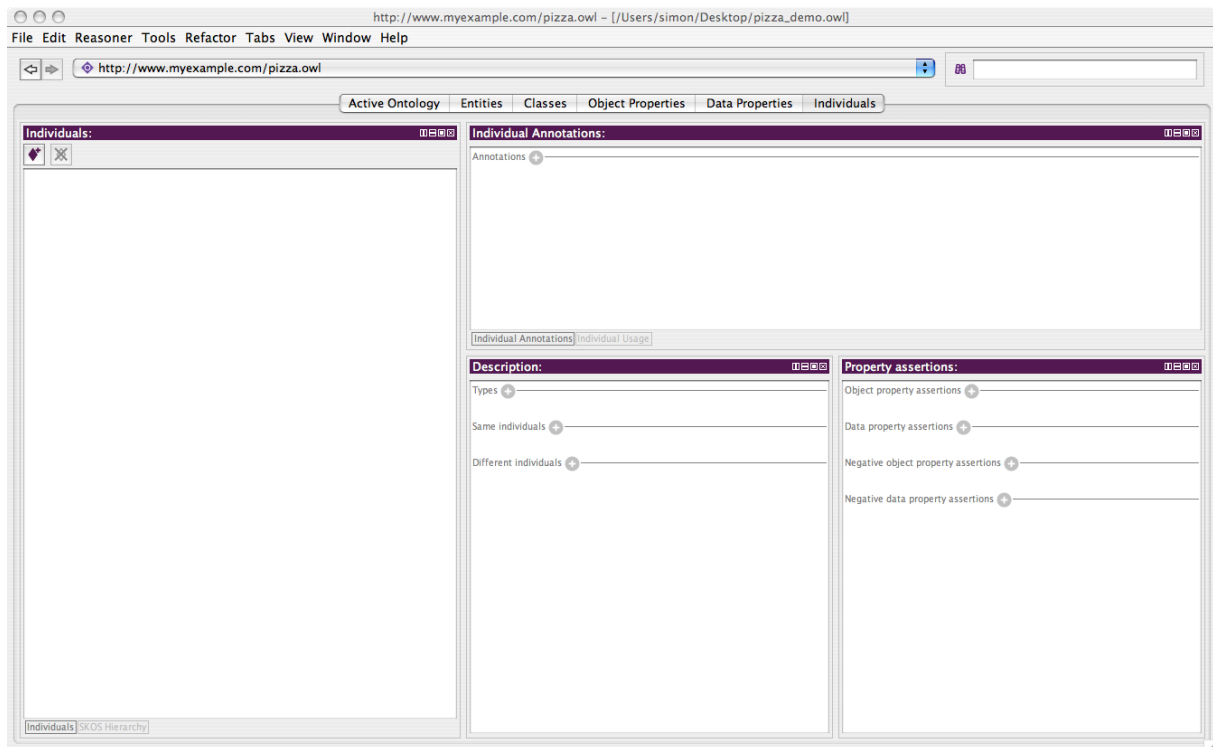


Figure 7.1: The Individuals Tab

will create a class `Country` and then ‘populate’ it with individuals:

Exercise 58: Create a class called `Country` and populate it with some individuals

1. Create `Country` as a subclass of `Thing`.
2. Switch to the ‘**Individuals Tab**’ shown in Figure 7.1.
3. Press the ‘**Add individual**’ button shown in Figure 7.2. (Remember that ‘Individual’ is another name for ‘Instance’ in ontology terminology).
4. Name the new Individual `Italy`.
5. Select the ‘**Select and add class**’ button from the ‘**Individual Types View**’ located in the centre of the Individual tab. Choose `country` from the class hierarchy, this will make `Italy` and individual of the class `Country`.
6. Use the above steps to create some more individuals that are members of the class `Country` called `America`, `England`, `France`, and `Germany`.

Recall from section 3.2.1 that OWL does not use the Unique Name Assumption (UNA). Individuals can therefore be asserted to be the ‘Same As’ or ‘Different From’ other individuals. In Protégé 4 these assertions can be made using the ‘**SameAs**’ and ‘**DifferentFrom**’ tabs shown in Figure 7.3, which are

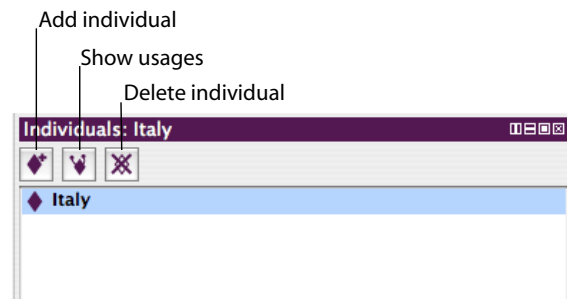


Figure 7.2: Instances Manipulation Buttons

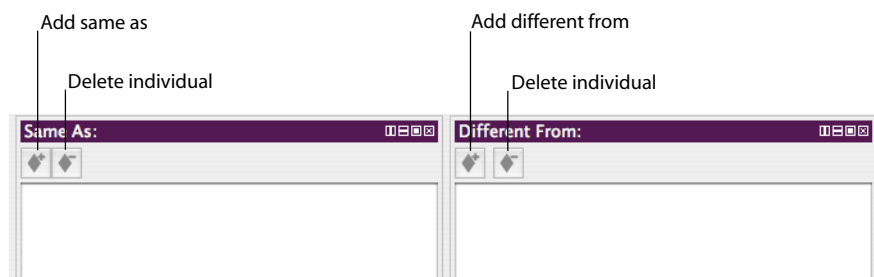


Figure 7.3: The SameAs and DifferentFrom Views

located with the ‘Name’ view on the ‘Individuals’ tab.

Having created some individuals we can now use these individuals in class descriptions as described in section 7.2 and section 7.3.

7.2 hasValue Restrictions

A hasValue restriction, denoted by the symbol \exists , describes the set of individuals that have *at least one* relationship along a specified property to a *specific individual*. For example, the hasValue restriction $\text{hasCountryOfOrigin} \exists \text{Italy}$ (where **Italy** is an individual) describes the set of individuals (the anonymous class of individuals) that have *at least one* relationship along the **hasCountryOfOrigin** property to the *specific* individual **Italy**. For more information about hasValue restrictions please see Appendix ??.

Suppose that we wanted to specify the origin of ingredients in our pizza ontology. For example, we might want to say that mozzarella cheese (**MozzarellaTopping**) is from **Italy**. We already have some countries in our pizza ontology (including **Italy**), which are represented as individuals. We can use a hasValue

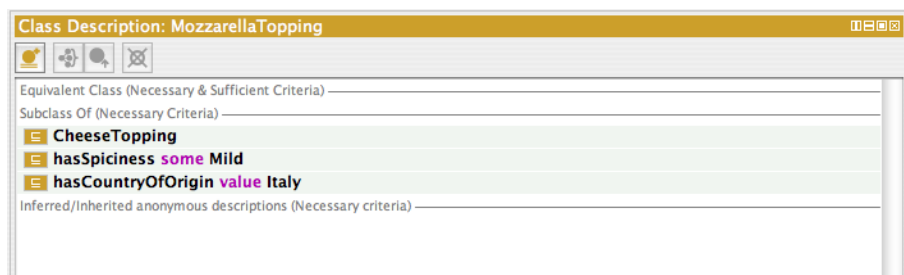


Figure 7.4: The Class Description View Displaying The hasValue Restriction for MozzarellaTopping

restriction along with these individuals to specify the county of origin of MozzarellaTopping as Italy.

Exercise 59: Create a hasValue restriction to specify that MozzarellaTopping has Italy as its country of origin.

1. Switch to the ‘**Object Properties**’ tab. Create a new object property and name it **hasCountryOfOrigin**.
2. Switch to the ‘**Classes**’ tab and select the class **MozzarellaTopping**.
3. Select the “Subclass of” header in the ‘**Class Description View**’.
4. Press the ‘**Create restriction**’ button on the ‘**Class Description View**’ to open a text box.
5. Type **hasCountryOfOrigin** as the property to be restricted.
6. Type **value** as the type of restriction to be created.
7. Enter **Italy** as the individual to complete the restriction. You can either type this in or drag and drop from the individuals window.
8. Press ‘**Enter**’ to close the dialog and create the restriction.

The ‘**Class Description View**’ should now look similar to the picture shown in Figure 7.4.

MEANING



The conditions that we have specified for **MozzarellaTopping** now say that: individuals that are members of the class **MozzarellaTopping** are also members of the class **CheeseTopping** *and* are related to the individual **Italy** via the **hasCountryOfOrigin** property *and* are related to at least one member of the class **Mild** via the **hasSpiciness** property. In more natural English, things that are kinds of mozzarella topping are also kinds of cheese topping and come from Italy and are mildly spicy.



With current reasoners the classification is *not complete* for individuals. Use individuals in class descriptions with care — unexpected results may be caused by the reasoner.

7.3 Enumerated Classes

As well as describing classes through named superclasses and anonymous superclasses such as restrictions, OWL allows classes to be defined by precisely listing the individuals that are the members of the class. For example, we might define a class **DaysOfTheWeek** to contain the individuals (and only the individuals) **Sunday**, **Monday**, **Tuesday**, **Wednesday**, **Thursday**, **Friday** and **Saturday**. Classes such as this are known as *enumerated* classes.

In Protégé 4 enumerated classes are defined using the ‘**Class Description View**’ expression editor – the individuals that make up the enumerated class are listed (separated by spaces) inside curly brackets. For example {**Sunday Monday Tuesday Wednesday Thursday Friday Saturday**}. The individuals must first have been created in the ontology. Enumerated classes described in this way are anonymous classes – they are the class of the individuals (and only the individuals) listed in the enumeration. We can attach these individuals to a named class in Protégé 4 by creating the enumeration as a “Equivalent class” condition.

Exercise 60: Convert the class **Country** into an enumerated class

1. Switch the the ‘**Classes**’ tab and select the class **Country**.
2. Select the “Equivalent class” header in the ‘**Class Description View**’.
3. Press the ‘**Add class**’ button, a text box will appear.
4. Type {**America England France Germany Italy**} into the text box. (Remember to surround the items with curly brackets). Remember that the auto complete function is available — to use it type the first few letters of an individual and press the tab key to get a list of possible choices.
5. Press the enter key to accept the enumeration and close the expression editor.

The ‘**Class Description View**’ should now look similar to the picture shown in Figure 7.5.

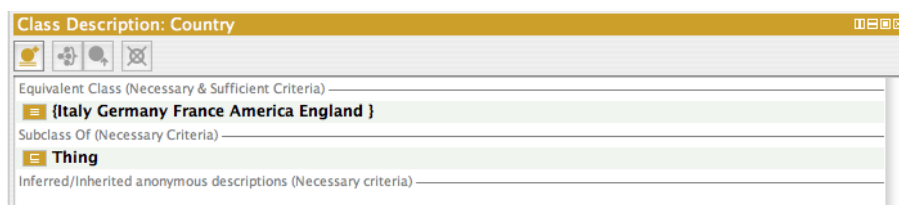


Figure 7.5: The Class Description View Displaying An Enumeration Class

MEANING



This means that an individual that is a member of the **Country** class must be one of the listed individuals (i.e one of **America England France Germany Italy**).^a More formally, the class country is equivalent to (contains the same individuals as) the anonymous class that is defined by the enumeration — this is depicted in Figure 7.6.

^aThis is obviously not a complete list of countries, but for the purposes of this ontology (and this example!) it meets our needs.

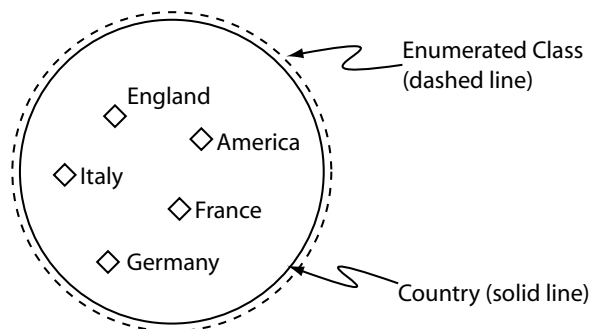


Figure 7.6: A Schematic Diagram Of The **Country** Class Being Equivalent to an Enumerated Class

TIP

The enumerated classes wizard is available for creating enumerated classes in the above fashion.

7.4 Annotation Properties

OWL allows classes, properties, individuals and the ontology itself (technically speaking the ontology header) to be annotated with various pieces of information/meta-data. These pieces of information may take the form of auditing or editorial information. For example, comments, creation date, author, or, references to resources such as web pages etc. OWL-Full does not put any constraints on the usage of annotation properties. However, OWL-DL does put several constraints on the usage of annotation properties — two of the most important constraints are:

- The filler for annotation properties must either be a data literal¹, a URI reference or an individual.
- Annotation properties cannot be used in property axioms — for example they may not be used in the property hierarchy, so they cannot have sub properties, or be the sub property of another property. They also must not have a domain and a range set for them.

OWL has five pre-defined annotation properties that can be used to annotate classes (including anonymous classes such as restrictions), properties and individuals:

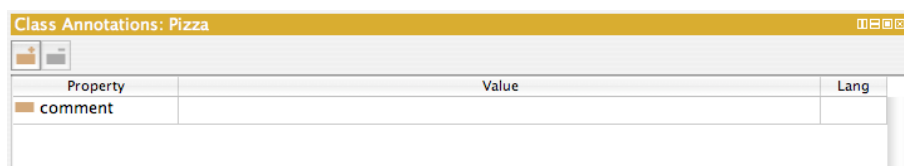
¹A data literal is the character representation of a datatype value, for example, “Matthew”, 25, 3.11.

1. **owl:versionInfo** — in general the range of this property is a string.
2. **rdfs:label** — has a range of a string. This property may be used to add meaningful, human readable names to ontology elements such as classes, properties and individuals. **rdfs:label** can also be used to provide multi-lingual names for ontology elements.
3. **rdfs:comment** — has a range of a string.
4. **rdfs:seeAlso** — has a range of a URI which can be used to identify related resources.
5. **rdfs:isDefinedBy** — has a range of a URI reference which can be used to reference an ontology that defines ontology elements such as classes, properties and individuals.

For example the annotation property **rdfs:comment** is used to store the comment for classes in Protégé 4 . The annotation property **rdfs:label** could be used to provide alternative names for classes, properties etc.

There are also several annotation properties which can be used to annotate an ontology. The ontology annotation properties (listed below) have a range of a URI reference which is used to refer to another ontology. It is also possible to use the **owl:VersionInfo** annotation property to annotate an ontology.

- **owl:priorVersion** — identifies prior versions of the ontology.
- **owl:backwardsCompatibleWith** — identifies a prior version of an ontology that the current ontology is compatible with. This means that all of the identifiers from the prior version have the same intended meaning in the current version. Hence, any ontologies or applications that reference the prior version can safely switch to referencing the new version.
- **owl:incompatibleWith** — identifies a prior version of an ontology that the current ontology is *not* compatible with.



Property	Value	Lang
comment		

Figure 7.7: An annotations view

To create annotation properties use the appropriate annotation property view in each of the ‘**Active Ontology**’, ‘**Classes**’, ‘**Object Property**’ and ‘**Datatype Property**’ Tabs. You can manage your annotation using the ‘**Annotations Properties**’ Tab, new annotation properties can be created by pressing the ‘**create Annotation Property**’ button on the ‘**Annotation Property**’ Tab. To use annotation properties the annotations views shown in Figure 7.7 is used. An annotations view is located on the Classes, Properties, Individuals and Active Ontology tab for annotation classes, properties, individuals and the ontology respectively. Annotations can also be added to restrictions and other anonymous classes by right clicking (ctrl click on a Mac) in the class description view and selecting ‘**Edit annotation properties...**’.

7.5 Multiple Sets Of Necessary & Sufficient Conditions

In OWL it is possible to have multiple sets of necessary and sufficient conditions as depicted in Figure 7.8. In the ‘**Class Description View**’, multiple sets of necessary and sufficient conditions are represented using multiple “Equivalent class” headers with necessary and sufficient conditions listed under each header as shown in Figure 7.8. To create a *new* set of necessary and sufficient conditions, any “Equivalent class” header (any that is visible) should be selected and then the condition created (for example using the ‘**Create Restriction dialog**’). Alternatively, a condition should be dragged and dropped onto a “Equivalent class” header to create a new set of necessary and sufficient conditions and move the condition to that new set. To add to an *existing* set of necessary and sufficient conditions, one of the conditions in the set should be selected and then the condition created (for example using ‘**Create Restrictions dialog**’), or an existing condition may be dragged and dropped onto the existing set (below the “Equivalent class” header) to add the condition to the existing set.

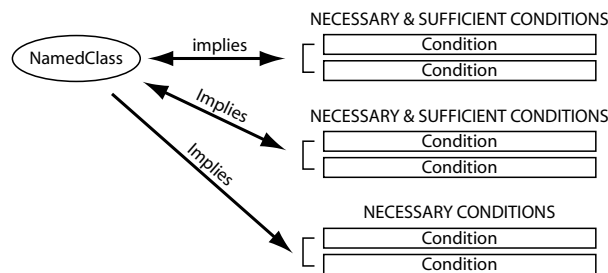


Figure 7.8: Necessary Conditions, and Multiple Sets of Necessary And Sufficient Conditions

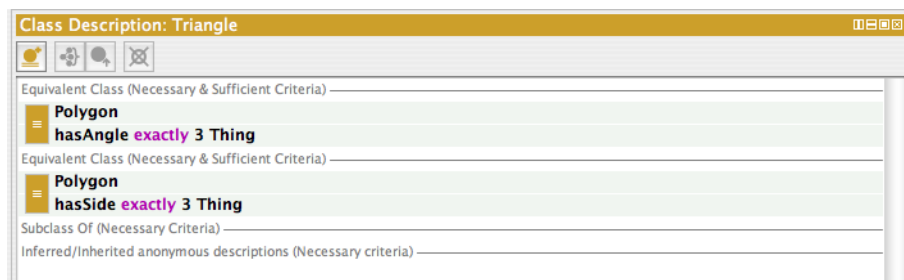


Figure 7.9: The Definition of a Triangle Using Multiple Necessary & Sufficient Conditions

Exercise 61: Create a class to define a Triangle using multiple sets of Necessary & Sufficient conditions

1. Create a subclass of **Thing** named **Polygon**.
2. Create a subclass of **Polygon** named **Triangle**.
3. Create an object property named **hasSide**.
4. Create an object property named **hasAngle**.
5. On the '**Classes**' tab select the **Triangle** class. Select the "Equivalent class" header in the '**Class Description View**'. Press the '**Add class**' button on the '**Class Description View**' to open a text box.
6. Type **hasSide** for the property.
7. Type **exactly** to create the restriction.
8. Type **3** and press Enter to exit the text box and create the restriction.
9. Select the "Equivalent class" header in the '**Class Description View**'. Press the '**Add class**' button on the '**Class Description View**' to open a text box.
10. Type **hasAngle** for the property.
11. Type **exactly** to create the restriction.
12. Type **3** and press Enter to exit the text box and create the restriction.
13. Drag **Polygon** from under the "Subclass of" header and drop it onto the **hasSide exactly 3** restriction.
14. Select the **hasAngle exactly 3** restriction. Click the '**Add class**' button to display a dialog containing the class hierarchy. Select the **Polygon** class and click the '**OK**' button to close the dialog.

The '**Class Description View**' should now look like the picture shown in Figure 7.9.

Figure 7.10: A role-chain using the properties `hasLocation` and `hasPart`

Figure 7.11: Creating a role chain in the Pizza Ontology in Protégé

7.6 Role Chains

We have already used the property characteristic of *transitivity* (see Section 4.6.3). This characteristic, however, only works along properties of the same kind. Sometimes, however, we do wish to have some transfer along properties of different kinds. This effect is achieved by using *Role Chains*. A role chain is best explained using an example. In Figure 7.10, we define a role chain using the object properties `hasLocation` and `hasPart`. We can use such properties, for instance, to describe an injury to a hand. We also describe that a hand is part of an arm. We know that an injured hand is also an injured arm. By creating a role chain of these two properties we can make such an inference. In other words what we are saying is: if any injury `hasLocation` hand, it is also an injury of the arm as there is a role chain between `hasLocation` and `hasPart`.

In the pizza ontology we can define a role chain using the object properties `hasTopping` and `hasSpiciness`.

Exercise 62: Creating a Role Chain using the properties `hasTopping` and `hasSpiciness` (Figure 7.11)

1. Go to the ‘**Object Properties**’ Tab and ensure that `hasSpiciness` is highlighted
 2. Go to the ‘**Property Chains**’ Heading in the ‘**Description**’ pane and click on the ‘**Add**’ button
 3. In the text box, type `hasTopping`, for the first property in the chain
 4. Then type ‘`o`’, which represents the role composition order (*i.e.*, the two properties we wish to link)
 5. Type `hasSpiciness` and then click ‘**OK**’.
-

Exercise 63: Creating a HotPizza class

1. Go to the ‘**Classes**’ Tab and ensure that the `Pizza` class is highlighted
 2. Create a subclass `HotPizza`
 3. Add a restriction in the ‘**Class Description**’ pane as an Equivalent class by typing ‘`Pizza that hasTopping some Hot`’
-

MEANING



Very simply, what we are describing by doing this role chain is that a topping that is spicy makes the pizza that uses it as a topping is spicy. By describing the spiciness of toppings, you should be able to define a class of spicy pizza.