20.453J / 2.771J / HST.958J Biomedical Information Technology
Fall 2008

# A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools
# Edition 1.0

Matthew Horridge[1],
Holger Knublauch[2], Alan Rector[1], Robert Stevens[1], Chris Wroe[1]

[1] THE UNIVERSITY OF MANCHESTER
[2] STANFORD UNIVERSITY

August 27, 2004

# Contents

# Copyright

# Acknowledgements

`http://www.co-ode.org`

# Chapter 1

# Introduction

This guide introduces the Protégé-OWL plugin 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 6 describes some OWL constructs such as hasValue Restrictions and Enumerated classes, which aren't directly used in the main tutorial. Chapter 7 describes Namespaces, Importing ontologies and various features and utilities of the Protégé-OWL application.

## 1.1   Conventions

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

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

Where exercises require information to be typed into Protégé-OWL 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.

**TIP**

> Tips and suggestions related to using Protégé-OWL and building ontologies are presented like this.

**MEANING**

> Explanation as to what things mean are presented like this.

> Potential pitfalls and warnings are presented like this.

**NOTE**

> General notes are presented like this.

Vocabulary

> Vocabulary explanations and alternative names are presented like this.

# Chapter 2

# Requirements

In order to follow this tutorial you must have Protégé 2.1 (or later)[1], the Protégé-OWL plugin (latest beta) and also the OWL Wizards Plugin, which are available via the CO-ODE web site [2]. Since the release of Protégé 2.1, the Protégé-OWL plugin and the OWL Wizards are bundled in one single download. 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 2.1 is installed. For installation steps, please see the documentation for each component. Finally, it is necessary to have a DIG (Description Logic Implementers Group) compliant reasoner installed in order to compute subsumption relationships between classes, and detect inconsistent classes. It is recommended that the latest version of the RACER reasoner be used, which can be obtained from `http://www.sts.tu-harburg.de/~r.f.moeller/racer/`.

---

[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. and, or 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-langauge. 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 and is based on *Description Logics* (hence the suffix DL). Description Logics are a decidable fragment of First Order Logic[2] and are therefore amenable to automated reasoning. It is therefore possible to automatically compute the classification hierarchy[3] 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 Overview[4]. 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).

The Protégé-OWL plugin does not make the distinction between editing OWL-Lite and OWL-DL ontologies. It does however offer the option to constrain the ontology being edited to OWL-DL, or allow the expressiveness of OWL-Full — See section 7.1 for more information on how to constrain the ontology to OWL-DL.

## 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 Instances, Slots and Classes.

---

[2]Logics are *decidable* if computations/algorithms based on the logic will terminate in a *finite* time.
[3]Also known as *subsumption reasoning*.
[4]http://www.w3.org/TR/owl-features
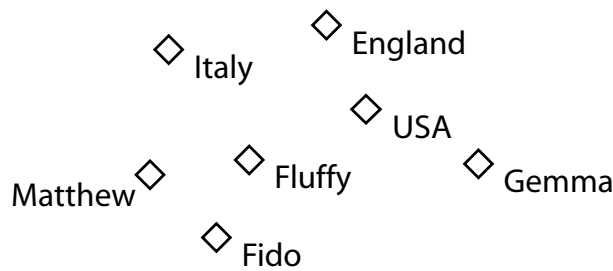
**Figure 3.1:** Representation Of Individuals

### 3.2.1 Individuals

Individuals, represent objects in the domain that we are interested in[5]. An important difference between Protégé and OWL is that OWL does not use the Unique Name Assumption (UNA). This means that two different names could actually refer to the same individual. For example, "Queen Elizabeth", "The Queen" and "Elizabeth Windsor" *might* all refer to the same individual. In OWL, it must be explicitly stated that individuals are the same as each other, or different 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.

> Vocabulary
>
> Individuals are also known as *instances*. Individuals can be referred to as being 'instances of classes'.

### 3.2.2 Properties

Properties are *binary* relations[6] on *individuals* - i.e. properties link *two* individuals together[7]. 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.

> Vocabulary
>
> 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*.

---

[5]Also known as *the domain of discourse.*

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

[7]Strictly speaking we should speak of 'instances of properties' linking individuals, but for the sake of brevity we will keep it simple.
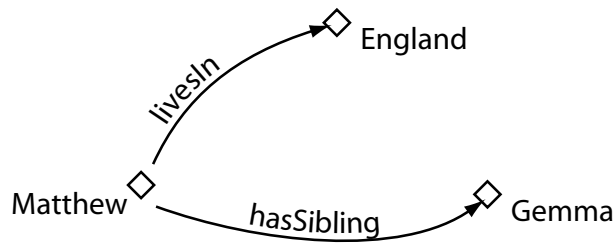
**Figure 3.2:** Representation Of Properties
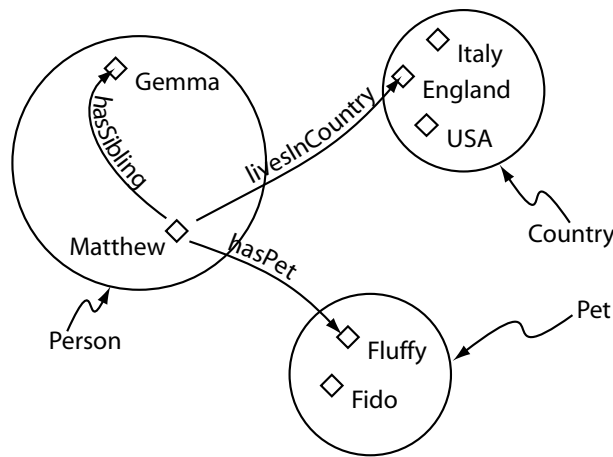


**Figure 3.3:** Representation Of Classes (Containing Individuals)

### 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 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.

Vocabulary

**A**Z
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

---

[8]Individuals may belong to more than one class.

14

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 project

1. Start Protégé

2. When the New Project dialog box appears, select '**OWL Files**' from the 'Project Format' list section on the left hand side of the dialog box, and press '**New**'.

After a short amount of time, a new empty Protégé-OWL project will have been created.

## 4.1 Named Classes

When Protégé-OWL starts the OWLClasses tab shown in Figure 4.1 will be visible. The initial class hierarchy tree view should resemble the picture shown in Figure 4.2. The empty ontology contains one class called owl:Thing. As mentioned previously, OWL classes are interpreted as sets of *individuals* (or sets of objects). The class owl:Thing is the class that represents the set containing *all* individuals. Because of this all classes are subclasses of owl:Thing.[2]

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

---

[1]The Ontology that we will created 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.

[2]owl:Thing is part of the OWL Vocabulary, which is defined by the ontology located at `http://www.w3.org/2002/07/owl/\#`
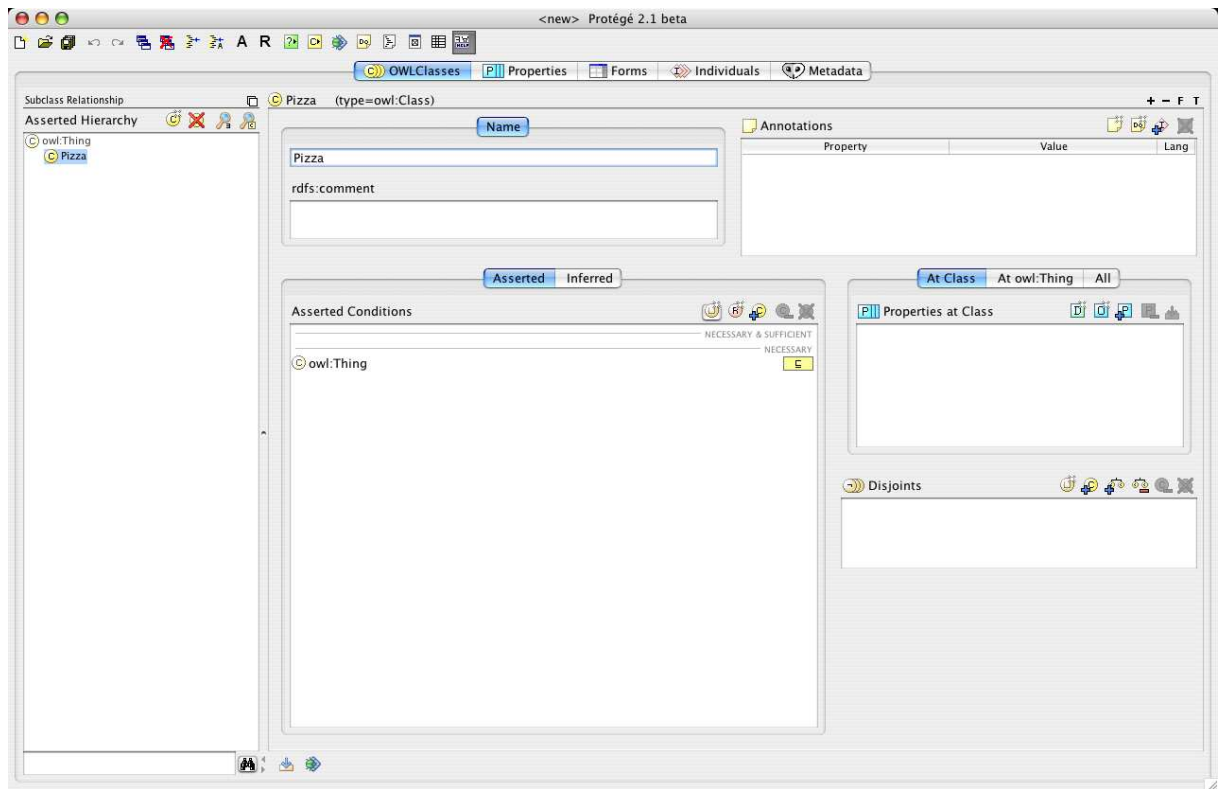
**Figure 4.1:** The Classes Tab
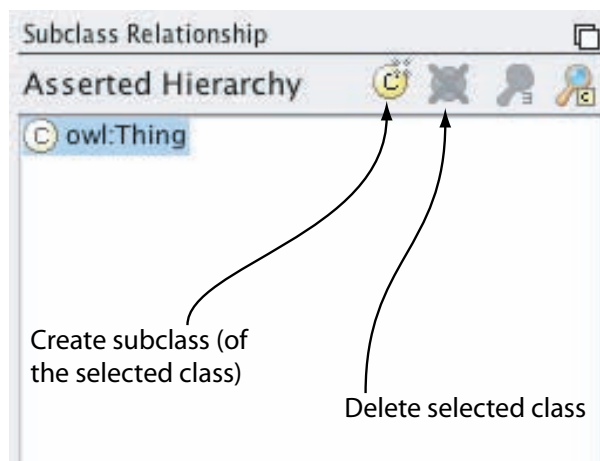


**Figure 4.2:** The Class Hierarchy Pane

17

**Figure 4.3:** Class Name Widget

---

**Exercise 3: Create classes** Pizza**,** PizzaTopping **and** PizzaBase

---

1. Press the '**Create subclass**' button shown in Figure 4.2. This button creates a new class as a subclass of the selected class (in this case we want to create a subclass of owl:Thing).

2. Rename the class using the '**Class name widget**' which is located to the right of the class hierarchy (shown in Figure 4.3) to Pizza and hit return.

3. Repeat the previous steps to add the classes PizzaTopping and also PizzaBase, ensuring that owl:Thing is selected before the '**Create subclass**' button is pressed so that the classes are created as subclasses of owl:Thing.

---

The class hierarchy should now resemble the hierarchy shown in Figure 4.4.



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 the '**Disjoints widget**' which is
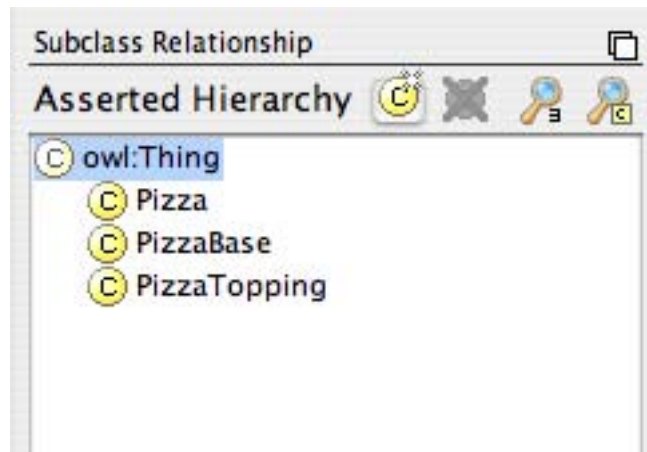
**Figure 4.4:** The Initial Class Hierarchy
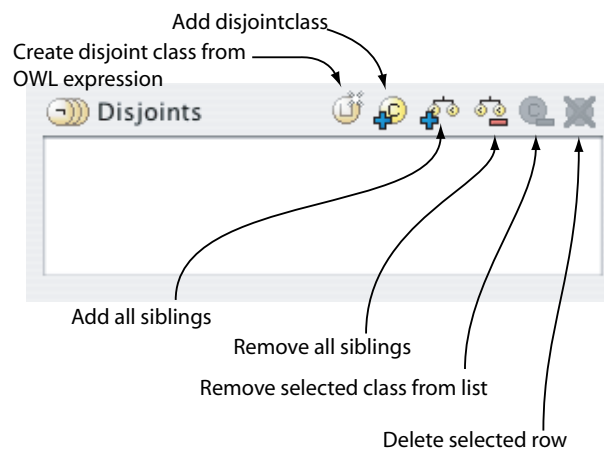


**Figure 4.5:** The Disjoint Classes Widget

located in the lower right hand corner of the '**OWLClasses**' tab is used. (See Figure 4.5).

---

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

---

1. Select the class Pizza in the class hierarchy.

2. Press the '**Add siblings**' button on the disjoint classes widget. This will make Pizz-aBase and PizzaTopping (the sibling classes of Pizza) disjoint from Pizza.

---

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

19
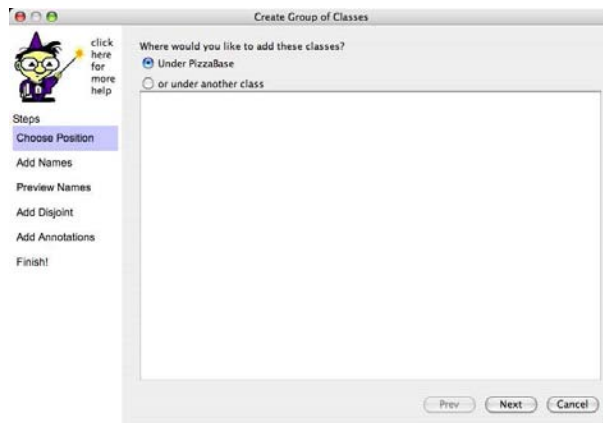
**Figure 4.6:** Add Group Of Classes Wizard: Select class page
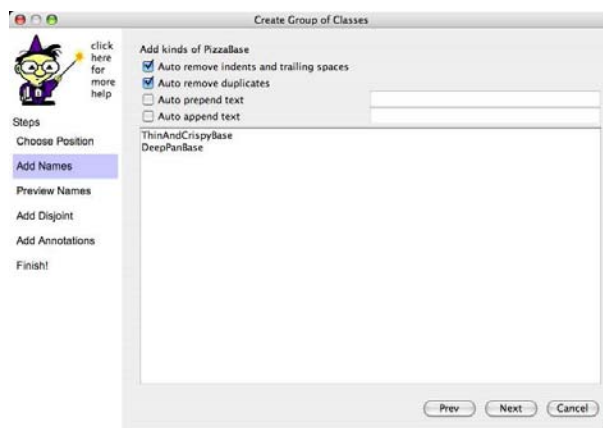


**Figure 4.7:** Add Group Of Classes Wizard: Enter classes page

**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 Wizards To Create Classes

The OWL Wizards plugin, which is available from the Protégé web site, is an extensible set of Wizards that are designed to make carrying out common, repetitive and time consuming tasks easy. In this section we will use the 'Create A Group Of Classes' wizard to add some subclasses of the class PizzaBase. To use the OWL Wizards you must ensure that the OWL Wizards plugin in installed and configured in Protégé .

**Exercise 5: Use the 'Create Group Of Classes' Wizard to create** ThinAndCrispy **and** DeepPan **as subclasses of** PizzaBase

1. Select the class PizzaBase in the class hierachy.

2. From the Wizards menu on the Protégé menu bar select the item '**Create Group Of Classes**'.

3. The Wizard shown in Figure 4.6 will appear. Since we preselected the PizzaBase class, the first radio button at the top of the Wizard should be prompting us to create the classes under the class PizzaBase. If we had not preselected PizzaBase before starting the Wizard, then the tree could be used to select the class.

4. Press the '**Next**' button on the Wizard—The page shown in Figure 4.7 will be displayed. We now need to tell the Wizard 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 4.7 .

5. Hit the '**Next**' button on the Wizard. The Wizard 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 be presented on this page, along with suggestions for corrections.

6. Hit the '**Next**' button on the Wizard. Ensure the tick box '**Make all new classes disjoint**' is *ticked* — instead of having to use the disjoint classes widget, the Wizard will automatically make the new classes disjoint for us.

7. Hit the '**Next**' button to display the annotations page. Here we could add annotations if we wanted to. Most commonly annotations are used to record editorial information about the ontology – who created it, when it was created, when it was revised, etc. The basic OWL annotation properties are selectable by default. For now, we will not add any annotations, so just hit the '**Finish**' button.

After the '**Finish**' button has been pressed, the Wizard creates the classes, makes them disjoint, and selects them in the Protégé OWLClasses tab. The ontology should now have ThinAndCrispyBase and also DeepPanBase as subclasses of PizzaBase. These new classes should also 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 Wizard would dramatically speed up the process of adding them.

TIP

On page two of the '**Create group of classes wizard**' the classes to be created are entered. If we had a lot of classes to create that had the same prefix or suffix we could use the options to auto prepend and auto append text to the class names that we entered.

## Creating Some Pizza Toppings

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

and seafood toppings.

---

**Exercise 6: Create some subclasses of** PizzaTopping

---

1. Select the class PizzaTopping in the class hierarchy.

2. Use the OWL Wizards to add some subclasses of PizzaTopping called MeatTopping, VegetableTopping, CheeseTopping and also SeafoodTopping. Make sure that these classes are disjoint to each other.

3. Next add some different kinds of meat topping. Select the class MeatTopping, and use the '**Create Group Of Classes**' Wizard to add the following subclasses of Meat-Topping: SpicyBeefTopping, PepperoniTopping, SalamiTopping, HamTopping. Once again, ensure that the classes are created as disjoint classes.

4. Add some different kinds of vegetable toppings by creating the following disjoint sub-classes of VegetableTopping: TomatoTopping, OliveTopping, MushroomTopping, PepperTopping, OnionTopping and CaperTopping. Add further subclasses of PepperTopping: RedPepperTopping, GreenPepperTopping and JalapenoPepperTopping making sure that the subclasses of PepperTopping are disjoint

5. Now add some different kinds of cheese toppings. In the same manner as before, add the following subclasses of CheeseTopping, ensuring that the subclasses are disjoint to each other: MozzarellaTopping, and ParmezanTopping

6. Finally, add some subclasses of SeafoodTopping to represent different kinds of sea food: TunaTopping, AnchovyTopping and PrawnTopping.

---

The class hierarchy should now look similar to that shown in Figure 4.8 (the ordering of classes may be slightly different).

**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 4.9.[a]

---
[a]It 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.

**Figure 4.8:** Class Hierarchy
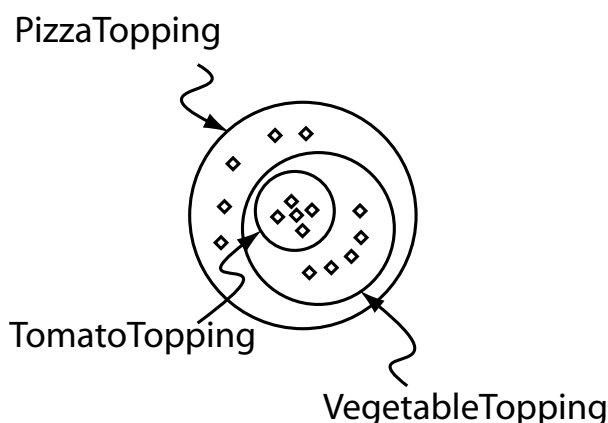
**Figure 4.9:** The Meaning Of Subclass — *All* individuals that are members of the class TomatoTopping are members of the class VegetableTopping and members of the class PizzaTopping as we have stated that TomatoTopping is a subclass of VegetableTopping which is a subclass of PizzaTopping

## 4.4   OWL Properties

OWL Properties represent relationships between two individuals. There are two main types of properties, *Object properties* and *Datatype properties*. Object properties link an individual to an individual. Datatype properties link an individual to an *XML Schema Datatype value*[3] or an *rdf literal*[4]. OWL also has a third type of property – *Annotation* properties[5]. Annotation properties can be used to add information (metadata — data about data) to classes, individuals and object/datatype properties. Figure 4.10 depicts an example of each type of property.

Properties may be created using the 'Properties' tab shown in Figure 4.11. It is also possible to create properties using the 'Properties Widget' shown in Figure 4.12 which is located on the 'OWLClasses' tab. Figure 4.13 shows the buttons located in the top left hand corner of the 'Properties' tab that are used for creating OWL properties. As can be seen from Figure 4.13, there are buttons for creating Datatype properties, Object properties and Annotation properties. Most properties created in this tutorial will be **Object properties**.

---

**Exercise 7:  Create an object property called** hasIngredient

---

1. Switch to the 'Propterties' tab. Use the 'Create Object Property' button (see Figure 4.13 – second button on the left) to create a new Object property. An Object property with a generic name will be created.

2. Rename the property to hasIngredient as shown in Figure 4.14 (The 'Property Name Widget'.

---

[3]See `http://www.w3.org/TR/xmlschema-2/` for more information on XML Schema Datatypes
[4]RDF = Resource Description Framework. See `http://www.w3.org/TR/rdf-primer/` for an excellent introduction to RDF.
[5]Object properties and Datatype properties may be marked as Annotation properties

hasSister

Matthew                     Gemma

An object property linking the individual
Matthew to the individual Gemma

hasAge

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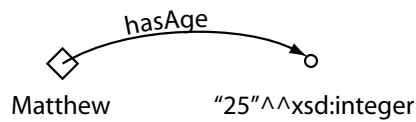
A datatype property linking the individual
Matthew to the data literal '25', which has a type
of an xml:integer.

dc:creator

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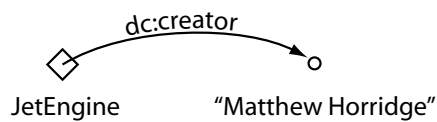
An annotation property, linking the class 'JetEngine'
to the data literal (string) ''Matthew Horridge''.

**Figure 4.10:** The Different types of OWL Properties
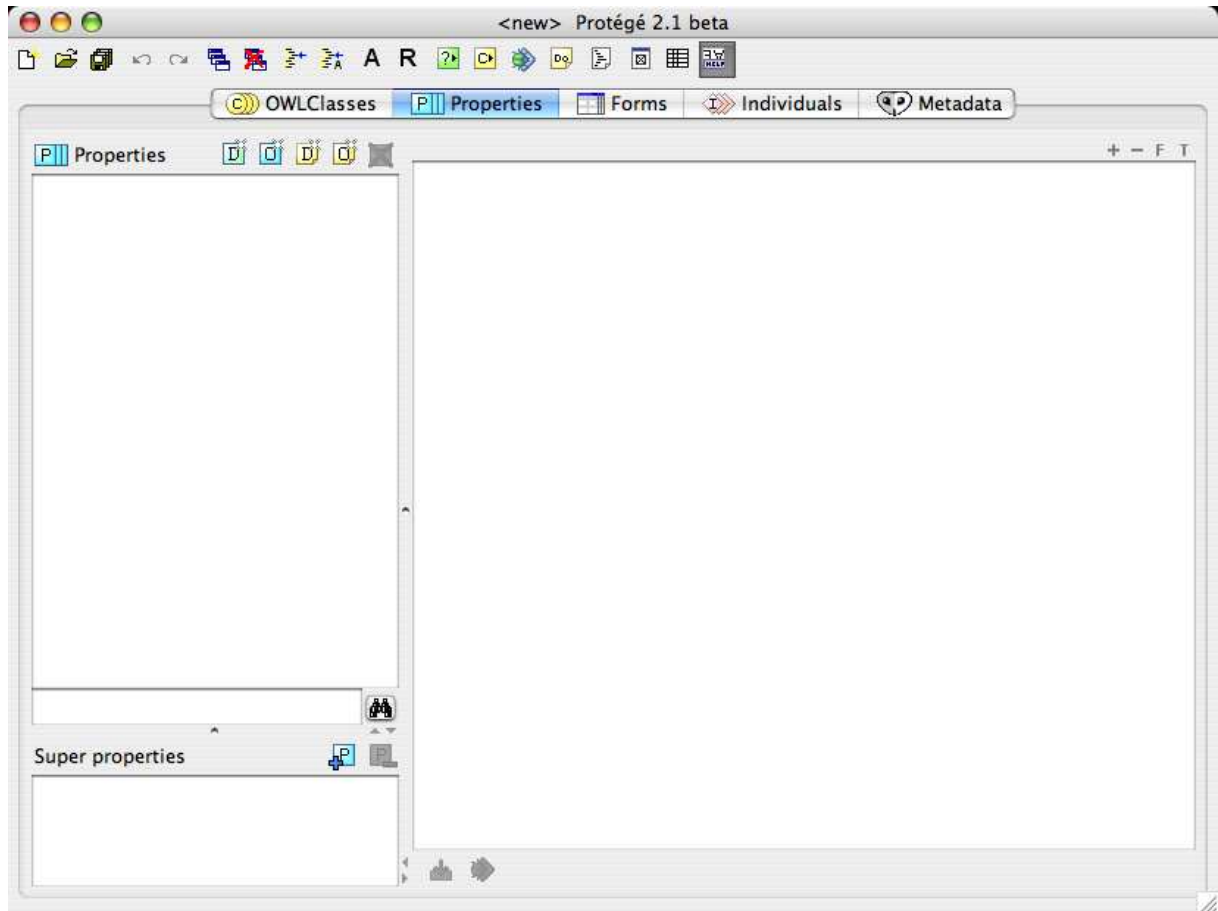
**Figure 4.11:** The PropertiesTab
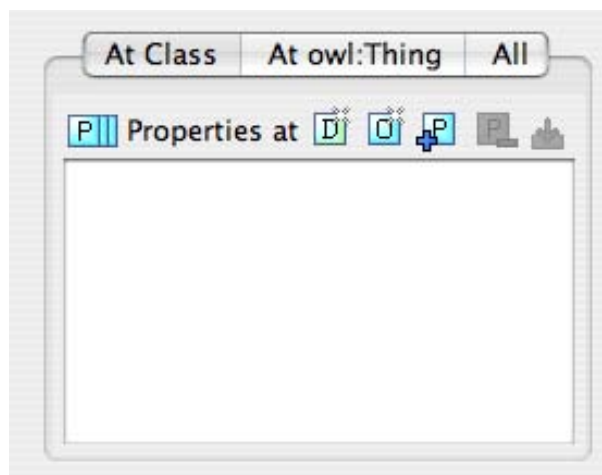


**Figure 4.12:** The Properties Widget
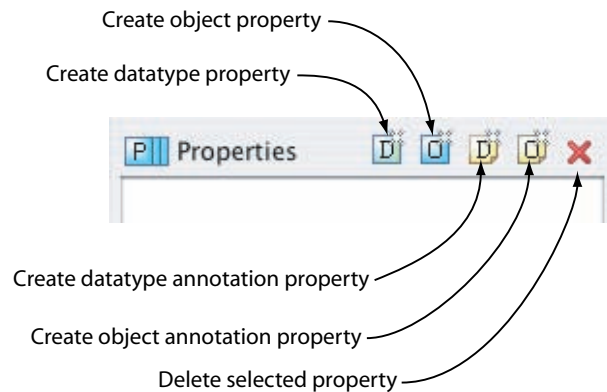
**Figure 4.13:** Property Creation Buttons — located on the Properties Tab above the property list/tree
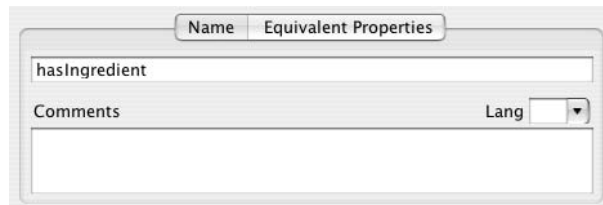


**Figure 4.14:** Property Name Widget

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.

[a]The English Prose Tooltip Generator displays the description of classes etc. in a more natural form of English, making is 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 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

28

individuals this implies that the two individuals are related by the **hasIngredient** property.

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

1. To create the **hasTopping** property as a sub property of the **hasIngredient** property, right click (or ctrl click on the Mac) on the **hasIngredient** property in the property hierarchy on the '**Properties**' tab. The menu shown in Figure 4.15 will pop up.

2. Select the '**Create subproperty**' item from the popup menu. A new object property will be created as a sub property of the **hasIngredient** property.

3. Rename the new property to **hasTopping**.

4. Repeat the above steps but name the property **hasBase**.
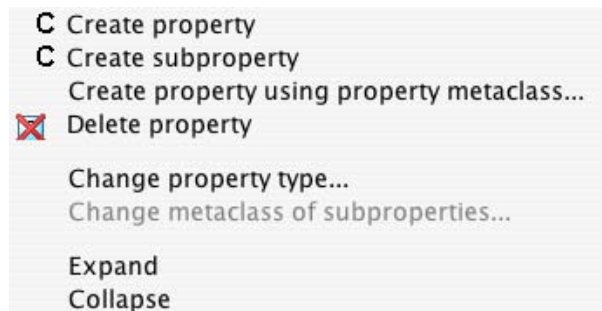


**Figure 4.15:** PropertyHierarchyMenu

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 widget shown in Figure 4.17. For
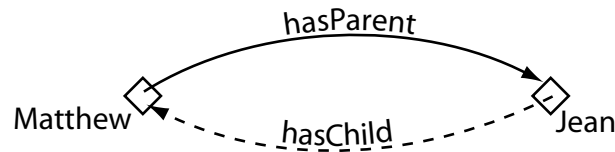
**Figure 4.16:** An Example Of An Inverse Property: hasParent has an inverse property that is hasChild

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

---

**Exercise 9: Create some inverse properties**

1. Use the '**Create object property**' button on the '**Properties**' tab to create a new Object property called isIngredientOf (this will become the inverse property of hasIngredient).

2. Press the '**Assign existing property**' button on the inverse property widget 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**' widget. 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 '**Create new inverse property**' button on the '**Inverse Property**' widget. This will pop up a dialog that contains information about the newly created property. Use this dialog to rename the property isBaseOf and the 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 '**Create new inverse property**' button on the '**Inverse Property**' widget. Use the property dialog that pops up to rename the property isToppingOf. Close the dialog. Notice that isToppingOf has been created as a sub property of isIngredientOf.

---

The property hierarchy should now look like the picture shown in Figure 4.18. Notice the 'bidirectional' arrows that indicate inverse properties.

## 4.6 OWL 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:
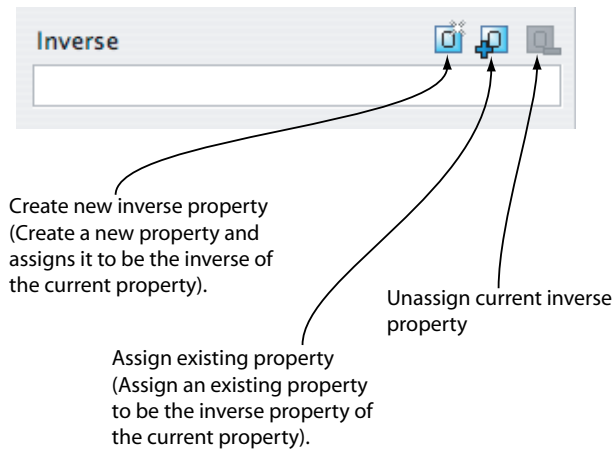
**Figure 4.17:** The Inverse Property Widget
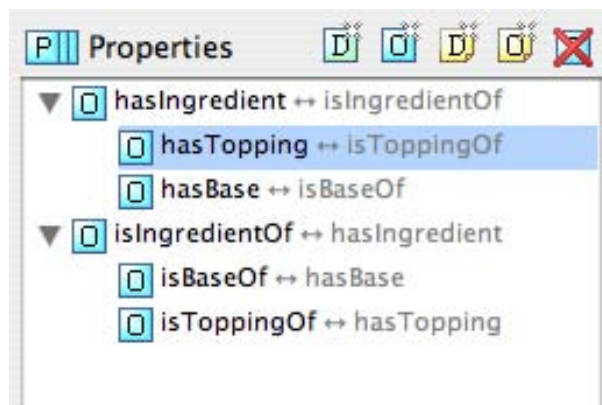


**Figure 4.18:** The Property Hierarchy With Inverse Properties

### 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[6], 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.
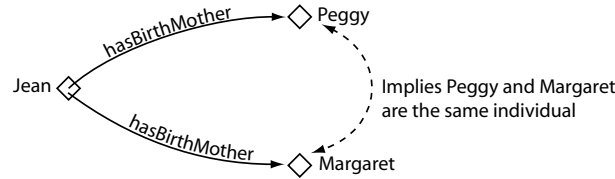


**Figure 4.19:** An Example Of A Functional Property: hasBirthMother



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.
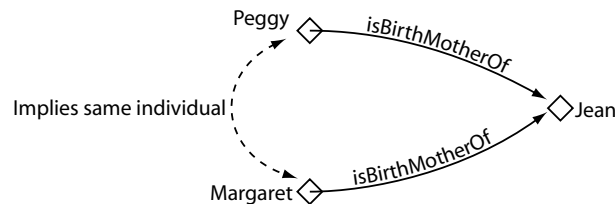


**Figure 4.20:** An Example Of An Inverse Functional Property: isBirthMotherOf

### 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

---

[6]The name Peggy is a diminutive form for the name Margaret

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.
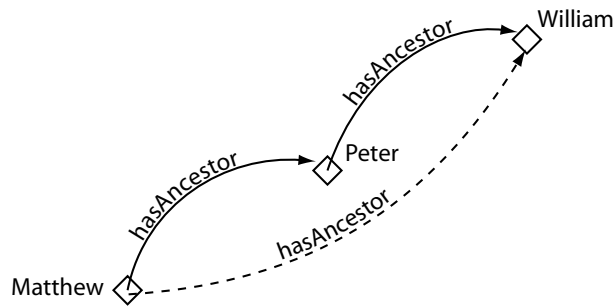


**Figure 4.21:** An Example Of A Transitive Property: hasAncestor

### 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 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.
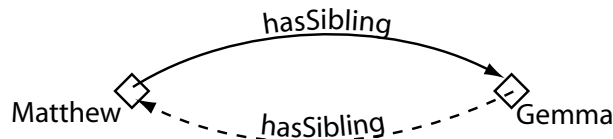


**Figure 4.22:** An Example Of A Symmetric Property: hasSibling

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 widget shown in Figure 4.23 which is located in the lower right hand corner of the properties tab is used.

**Exercise 10: Make the** hasIngredient **property transitive**

1. Select the **hasIngredient** property in the property hierarchy on the '**Properties**' tab.

2. Tick the '**Transitive**' tick box on the '**Property Characteristics Widget**'.

3. Select the **isIngredientOf** property, which is the inverse of **hasIngredient**. Ensure that the transitive tick box is ticked.

Unticking this box makes a property functional

Ticking this box makes a property Invers Functional

An inverse property may be specified here. Use the 'O' button to create a new property and simultaneously make the new property the inverse. Use the 'O+' button to add an existing property as the inverse.

Checking this box makes the property transitive

Checking this box makes the property symmetric

**Figure 4.23:** Property Characteristics Widgets

NOTE

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

[a]At the time of writing this must be done manually in Protégé-OWL . 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]

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

**Exercise 11: Make the** hasBase **property functional**

1. Select the hasBase property.

2. Click the '**Allows multiple values**' tick box on the '**Property Charachteristics Widget**' so that it is unticked.
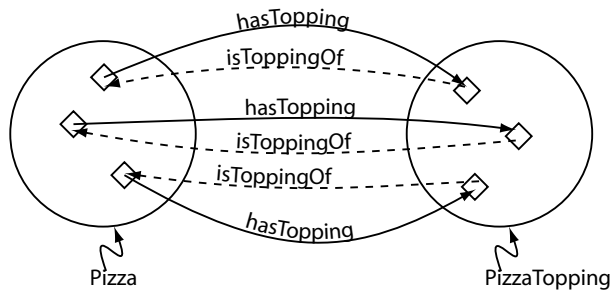
**Figure 4.24:** The domain and range for the hasTopping property and its inverse property isToppingOf. The domain of hasTopping is Pizza the range of hasTopping is PizzaTopping — the domain and range for isToppingOf are the domain and range for hasTopping swapped over

**NOTE**

If a datatype property is selected, the property characteristics widget 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.

## 4.7 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 the 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 4.24.

**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].

---

[a]An 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 widget shown in Figure 4.25 is used. By default the drop down box displays 'Instance', meaning that the
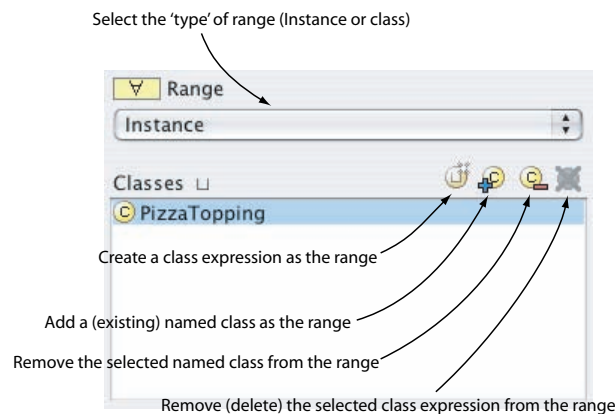
**Figure 4.25:** Property Range Widget (For Object Properties)

property links instances of classes to instances of classes.

### Exercise 12: Specify the range of hasTopping

1. Make sure that the hasTopping property is selected in the property hierarchy on the '**Properties**' tab.

2. Press the '**Add named class**' button on the '**Range Widget**' (Figure 4.25). 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.

It is also possible, but usually incorrect, to specify that a *class* rather than its individuals is the range of a property. This is done by selecting 'class' in the drop down box on the range widget. It is a common mistake to believe that the range of a property is a class when the range is really the individuals that are members of the class. Specifying the range of a property as a class treats the class itself as an individual[a]. This is a kind of 'meta-statement' and causes the ontology to be in OWL-Full.

_____
[a]Remember that instances of object properties link individuals to individuals

It is possible to specify multiple classes as the range for a property. If multiple classes are specified in Protégé-OWL the range of the property is interpreted to be the *union* of the classes. For example, if the range of a property has the classes Man and Woman listed in the range widget, the range of the property will be interpreted as Man *union* Woman. [a]

---
[a]See section B.2 for an explanation of what a union class is.

To specify the domain of a property the domain widget shown in Figure 4.26 is used.

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

1. Make sure that the hasTopping property is selected in the property hierarchy on the 'Properties' tab.

2. Press the 'Add named class' button on the Domain Widget. 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.

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].

---
[a]This 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.

When multiple classes are specified as the domain for a property, Protégé-OWL interprets the domain of the property to be the union of the classes .

Although OWL allows arbitrary *class expressions* to be used for the domain of a property this is not allowed when editing ontologies in Protégé-OWL .
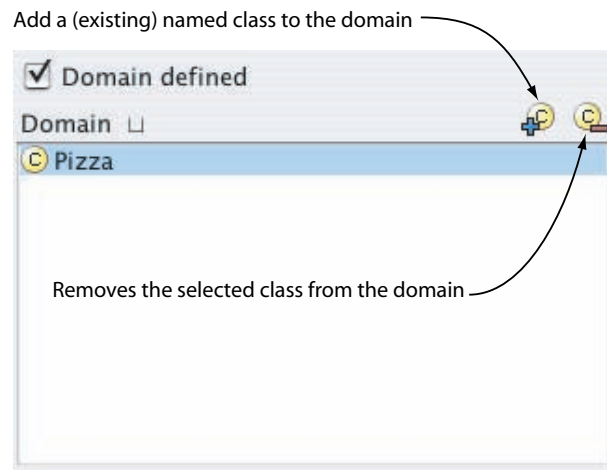
**Figure 4.26:** Property Domain Widget

We now need to fill in the domain and range for the inverse of the hasTopping property isToppingOf:

### Exercise 14: Specify the domain and range for the isToppingOf **property**

1. Select the isToppingOf property.

2. Use the same steps as above to set the domain of the isToppingOf property to PizzaTopping.

3. Set the range of the isToppingOf property to Pizza.

Notice that the domain of the isToppingOf property is the range of the inverse property hasTopping, and that the range of the isToppingOf property is the domain of the hasTopping property. This is depicted in 4.24.

### 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.

5. Make the domain of the isBaseOf property PizzaBase.

6. Make the range of the isBaseOf property Pizza.

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 4.24 illustrates this for the hasTopping and isToppingOf. If these steps are not taken the ontology tests (see section 7.3) can be used to spot any discrepancies.

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

In OWL properties are used to create *restrictions*. As the name may suggest, restrictions are used to restrict the individuals that belong to a class. Restrictions in OWL fall into three main categories:

- Quantifier Restrictions
- Cardinality Restrictions
- hasValue Restrictions.

We will initially use quantifier restrictions. These types of restrictions are composed of a *quantifier*, a property, and a *filler*. The two quantifiers that may be used are:

- The *existential quantifier* ($\exists$), which can be read as *at least one*, or *some*[7].
- The *universal quantifier* ($\forall$), which can be read as *only*[8].

For example the restriction $\exists$ hasTopping MozzarellaTopping is made up of the existential quantifier ($\exists$), the property hasTopping, and the filler MozzarellaTopping. This restriction describes the set, or the

---

[7]It can also be read as 'someValuesFrom' in OWL speak.
[8]It can also be read as 'allValuesFrom' in OWL speak.
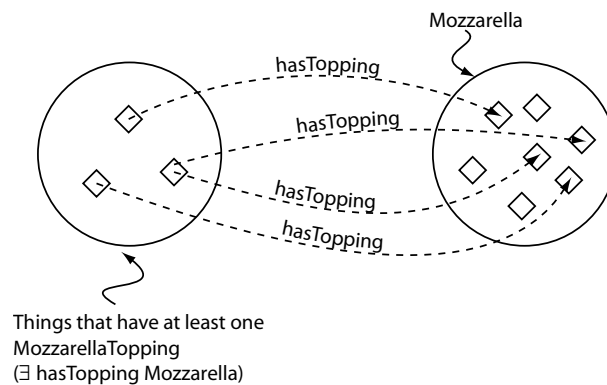
**Figure 4.27:** The Restriction ∃ hasTopping Mozzarella. This restriction describes the class of individuals that have *at least one* topping that is Mozzarella

class, of individuals that have *at least one* topping that is an individual from the class MozzarellaTopping. This restriction is depicted in Figure 4.27 — The diamonds in the Figure represent individuals. As can be seen from Figure 4.27, the restriction describes an *anonymous* (unnamed) class of individuals that satisfy the restriction.

**MEANING**

> A restriction actually describes an *anonymous class* (an unnamed class). The anonymous class contains all of the individuals that satisfy the restriction (see **Appendix A** for further information about what a restriction actually represents and an explanation of existential and universal quantification). 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.

The restrictions for a class are displayed and edited using the '**Conditions Widget**' shown in Figure 4.28. The '**Conditions Widget**' is the 'heart of' the '**OWLClasses**' tab in protege, and holds virtually all of the information used to describe a class. At first glance, the '**Conditions Widget**' may seem complicated, however, it will become apparent that it is an incredibly powerful way of describing and defining classes.

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

### 4.8.2 Existential Restrictions

Existential restrictions (∃) are by far the most common type of restrictions in OWL ontologies. For a set of individuals, an existential restriction specifies the *existence of a* (i.e. at least one) relationship along a given property to an individual that is a member of a specific class. For example, ∃ hasBase 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.
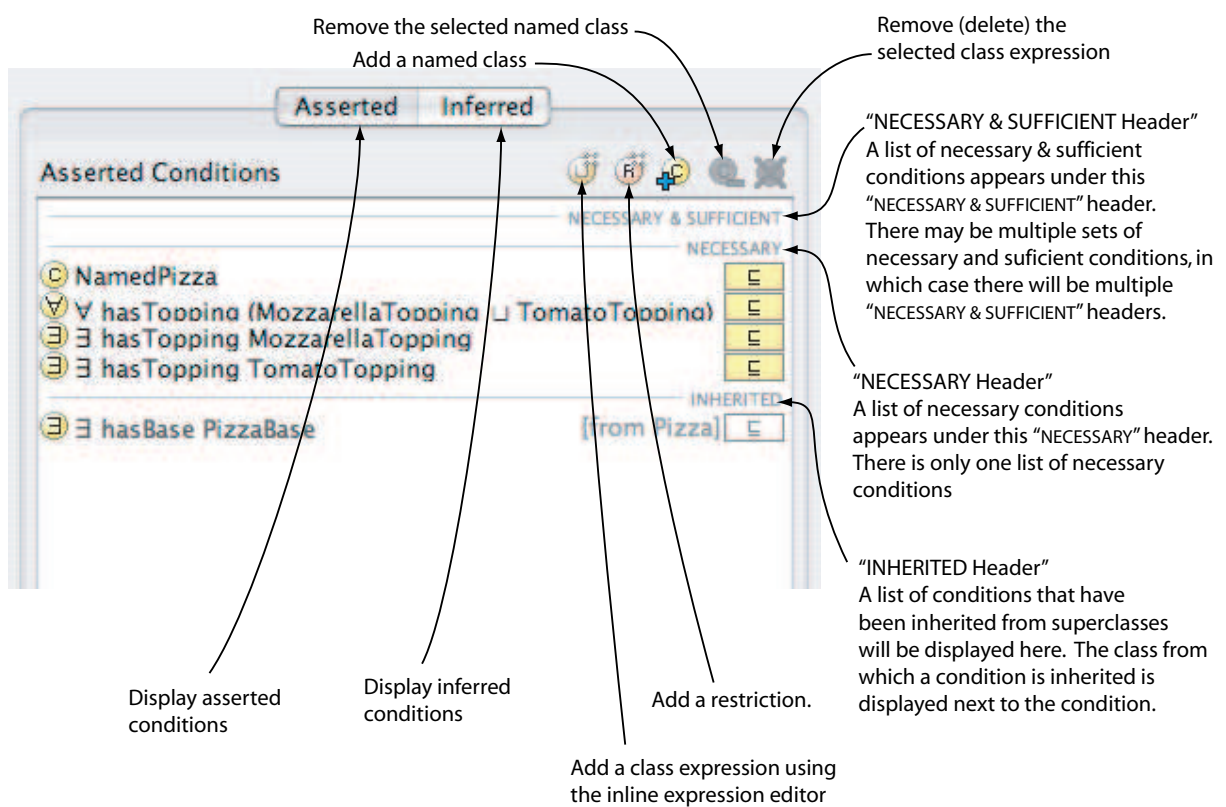
**Figure 4.28:** The Conditions Widget
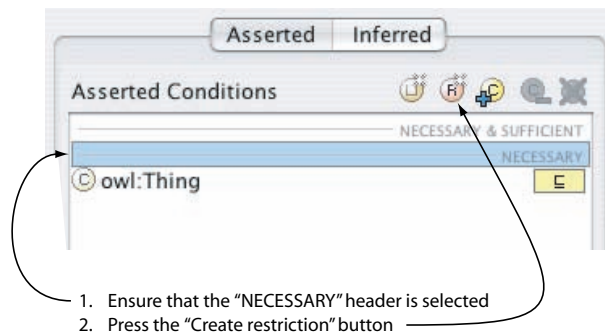
**Figure 4.29:** Creating a Necessary Restriction

Vocabulary

**A-Z**

Existential restrictions are also known as *Some Restrictions*.

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

1. Select Pizza from the class hierarchy on the '**OWLClasses**' tab.

2. Select the "NECESSARY" header in the '**Conditions Widget**' shown in Figure 4.29 in order to create a necessary condition.

3. Press the '**Create restriction**' button shown in Figure 4.29. This will display the '**Create Restriction**' dialog shown in Figure 4.30, which we will use to create a restriction.

The create restriction dialog has four main parts: The restriction type list; The property list; the filler edit box; and the expression builder panel. To create a restriction we have to do three things:

- Select the type of restriction from the restriction type list - the default is an existential ($\exists$) restriction.

- Select the property to be restricted from the property list.

- Specify a filler for the restriction in the filler edit box (possibly using the expression builder panel).

Property list

Restriction type list

Expression builder panel

Filler edit box
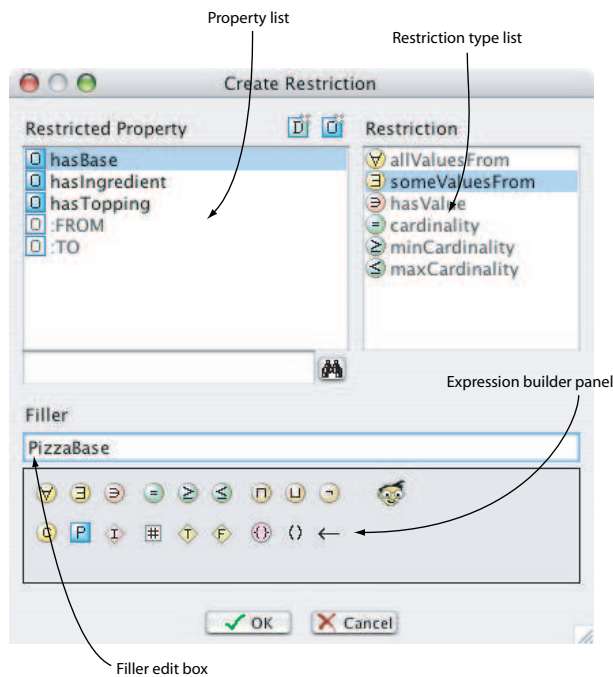
**Figure 4.30:** The Create Restriction Dialog

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

1. Select '∃ **someValuesFrom**' from the restriction type list — 'someValuesFrom' is another name for an existential restriction.

2. Select the property hasBase from the property list.

3. Specify that the filler is PizzaBase — to do this either: type `PizzaBase` into the filler edit box, or press the '**Insert class**' button on the expression builder panel shown in Figure 4.31 to display a class hierarchy tree from which PizzaBase may be selected.

4. Press the '**OK**' button to create the restriction and close the create restriction dialog. If all information was entered correctly the dialog will close and the restriction will be displayed in the '**Conditions Widget**'. If there were errors the dialog will not close. An error message will be displayed at the bottom of the expression builder panel — if this is the case, recheck that the type of restriction, the property and filler have been specified correctly.
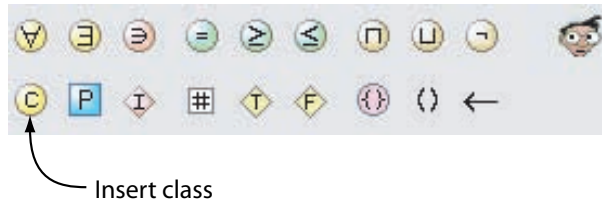
Insert class

**Figure 4.31:** Expression Builder Panel Insert Class Button
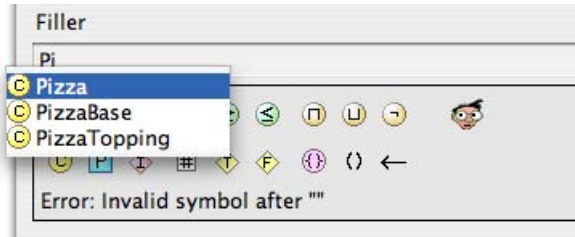


**Figure 4.32:** The Expression Builder Auto-Completion Function

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 poped up in a list as shown in Figure 4.32. 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 conditions widget should now look similar to the picture shown in Figure 4.33.
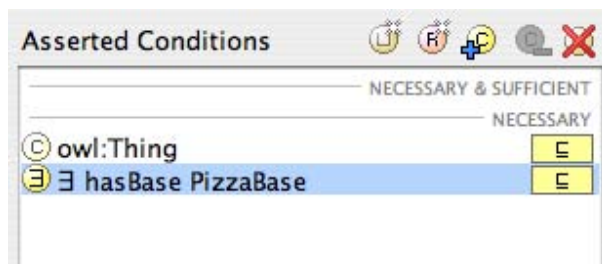


**Figure 4.33:** Conditions Widget: Description of a Pizza

44
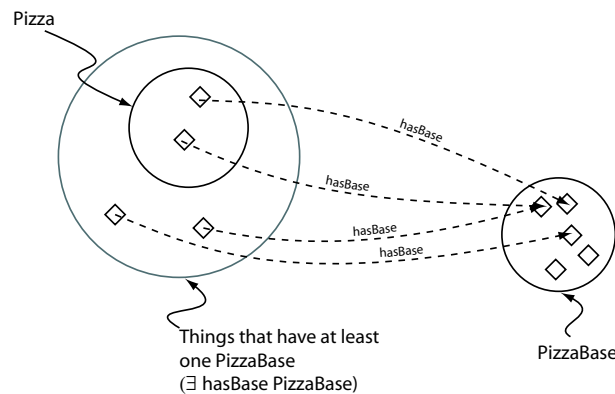
**Figure 4.34:** A Schematic Description of a Pizza — In order for something to be a Pizza it is *necessary* for it to have a (*at least one*) PizzaBase — A Pizza is a *subclass* of the things that have *at least one* PizzaBase

We have described the class Pizza to be a subclass of owl: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 owl:Thing (in OWL, everything is a member of the class owl: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 4.34.

## 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 '**OWLClasses**' tab.

2. Press the '**Create 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 comment box that is located under the class name widget: `A pizza that only has Mozarella 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 (∃) 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 "NECESSARY" header in the '**Conditions Widget**', as we want to create and add a necessary condition.

3. Use the '**Create restriction**' button on the '**Conditions widget**' (Figure 4.28) to display the '**Create Restriction**' dialog.

4. On the '**Create restrictions**' dialog make the restriction an existentially quantified restriction by selecting the restriction type as '∃ **someValuesFrom**'.

5. Select **hasTopping** as the property to be restricted.

6. Enter 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 the '**Insert class**' button (Figure 4.31) to display a dialog containing the ontology class hierarchy which may be used to choose a class.

7. Press the '**OK**' button on the create restriction dialog to create the restriction — if there are any errors, the restriction will not be created, and an error message will be displayed at the bottom of the expression builder panel.

Now specify that MargheritaPizzas also have TomatoTopping.

---

**Exercise 20: Create a existential restriction (∃) 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 "NECESSARY" header in the '**Conditions Widget**', as we want to create and add a necessary condition.

3. Use the '**Create restriction**' button on the '**Conditions Widget**' (Figure 4.28) to display the '**Create Restriction**' dialog'.

4. On the restrictions dialog make the restriction an existentially quantified restriction by selecting the restriction type as '∃ **someValuesFrom**'.

5. Select hasTopping as the property to be restricted.

6. Enter the class TomatoTopping as the filler for the restriction.

7. Click the '**OK**' button on the create restriction dialog to create the restriction.

---

The '**Conditions Widget**' should now look similar to the picture shown in Figure 4.35.

**MEANING**

**?**

> We have added restrictions to MargeritaPizza 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 conditions widget 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.

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
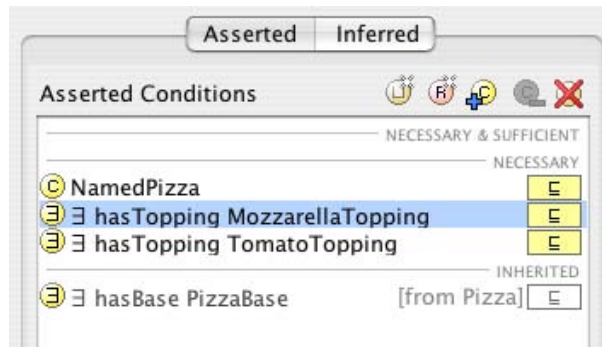
**Figure 4.35:** The Conditions Widget Showing A Description Of A MargheritaPizza

topping of pepperoni.

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

1. Right click (ctrl click on the Mac) on the class MargheritaPizza in the class hierarchy on the OWLClasses tab to display the class hierarchy popup menu.

2. From the popup menu select the menu item '**Create clone**'. 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 class name widget.

4. Ensuring that AmericanaPizza is still selected, select the "NECESSARY" header in the conditions widget, as we want to add a new restriction to the necessary conditions for AmericanaPizza.

5. Press the '**Create restriction**' button on the conditions widget to display the '**Create restriction dialog**'.

6. Select '∃ **someValuesFrom**' as the type of restriction to create an existentially quantified restriction.

7. Select the property hasTopping as the property to be restricted.

8. Specify the restriction filler as the class PepperoniTopping by either typing `PepperoniTopping` into the filler edit box, or by using the '**Insert class**' button to display the class dialog, from which PepperoniTopping may be selected.

9. Press the OK button to create the restriction.
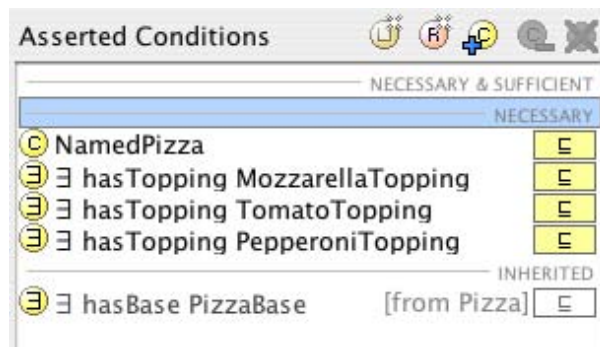
**Figure 4.36:** The Conditions Widget displaying the description for AmericanaPizza
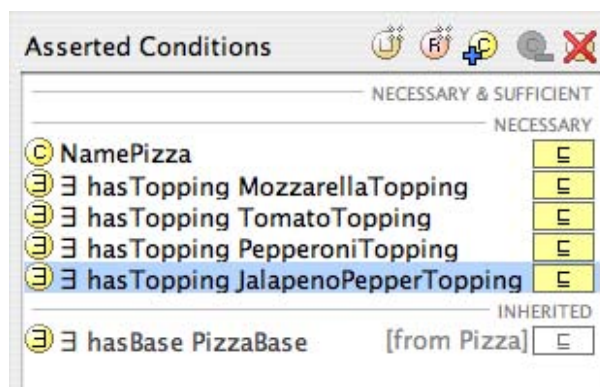


**Figure 4.37:** The Conditions Widget displaying the description for AmericanHotPizza

The '**Conditions Widget**' should now look like the picture shown in Figure 4.36.

---

**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 conditions widget should now look like the picture shown in Figure 4.37. For SohoPizza the conditions widget should now look like the picture shown in 4.38.
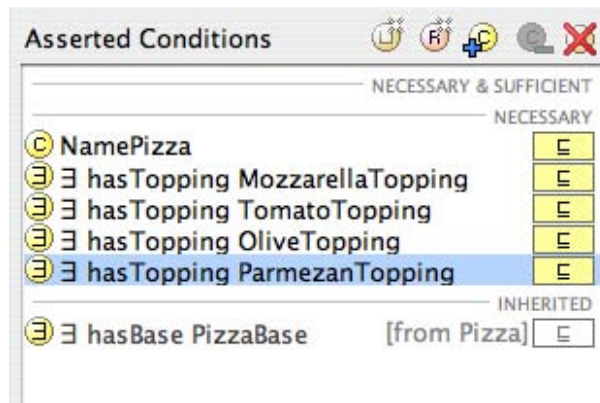
**Figure 4.38:** The Conditions Widget displaying the description for SohoPizza

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 '**OWLClasses**' tab.

2. Press the '**Add all siblings**' button on the '**Disjoints widget**' 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 website[9]. Protégé-OWL 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/Convert OWL Sub-language...**' option on the '**OWL menu**' shown in Figure 4.39. 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 class[10]. By performing such tests on all of 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.

---

[9]http://www.w3.org/TR/owl-features/

[10]Known as subsumption testing — the descriptions of the classes (conditions) are used to determine if a super-class/subclass relationship exists between them.
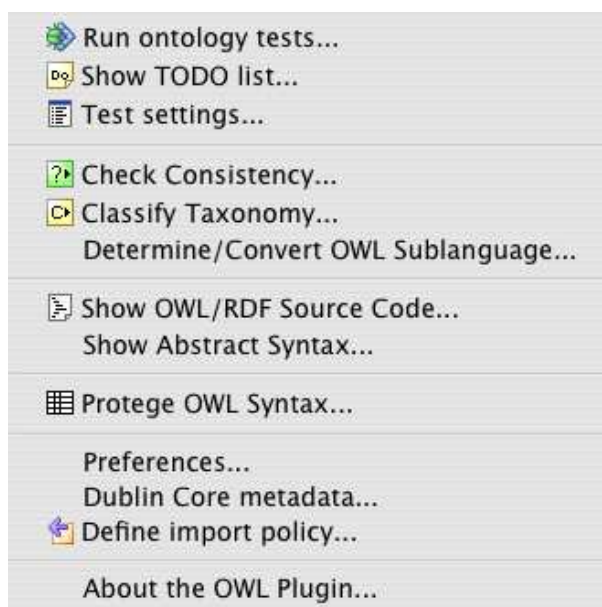
**Figure 4.39:** The OWL Menu

Vocabulary

Reasoners are also known as *classifiers*.

### 4.9.2 Using RACER

In order to reason over the ontologies in Protégé-OWL a DIG[11] compliant reasoner must be installed/configured and started. In this tutorial we use a reasoner called RACER, which is available for a variety of platforms from `http://www.sts.tu-harburg.de/~r.f.moeller/racer/`. RACER comes with a detailed manual that contains installation and setup instructions. When you have installed RACER on your system, it should be started with the default settings — RACER is typically started by double clicking on the RACER application icon, which opens a terminal/console window and starts the reasoner running with HTTP communication enabled.[12] — Figure 4.40 shows a pruned version of the information that is displayed when RACER starts; The second from the bottom line indicates that HTTP communication is running, and specifies the I.P. address and port number. If for any reason RACER needs to be started on a different port (or computer), Protégé-OWL can be configured via the OWL preferences dialog shown in Figure 4.41, which is accessible via the 'Preferences...' item on the OWL menu.

### 4.9.3 Invoking The Reasoner

Having started RACER, or another reasoner, 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-OWL the 'manually constructed' class hierarchy is called the *asserted hierarchy*. The class hierarchy

---

[11]DIG = Description Logic Implementers Group — A DIG compliant reasoner provides the means to communicate via the DIG interface, which is a standard interface/protocol for talking to description logic reasoners.

[12]By default racer runs with the HTTP service enabled on port 8080.

**Figure 4.40:** RACER Startup Screen
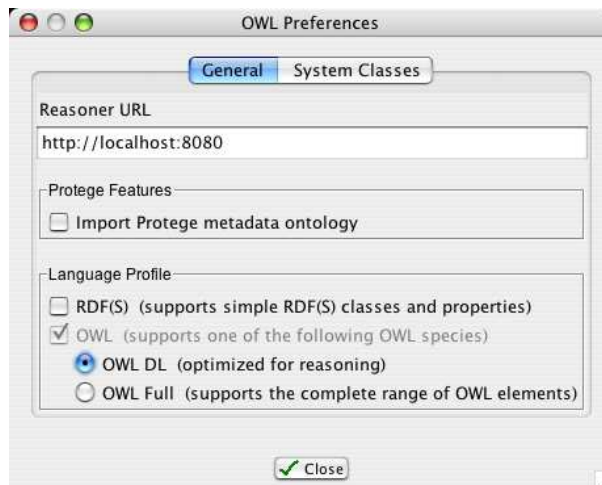


**Figure 4.41:** The OWL Preferences Dialog
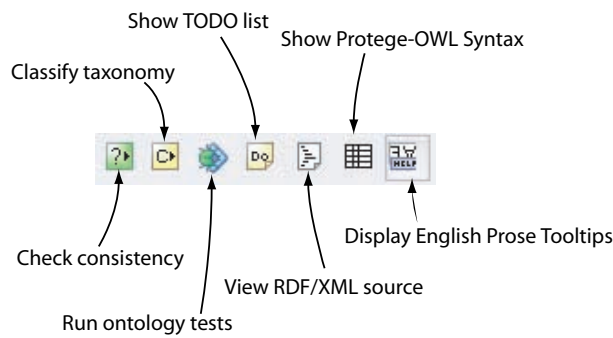
**Figure 4.42:** The OWL Toolbar
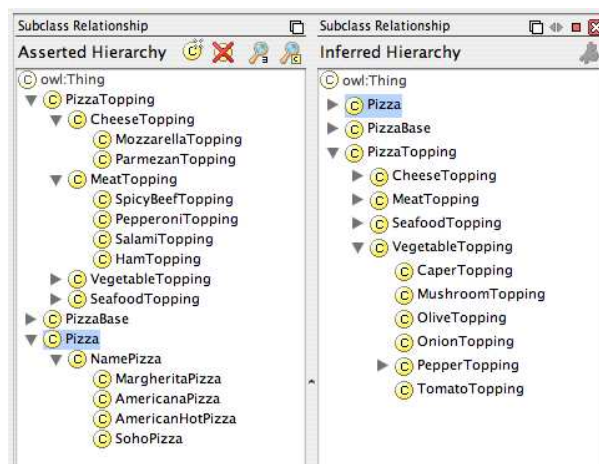


**Figure 4.43:** The Inferred Hierarchy Pane, which pops open next to the Asserted Hierarchy Pane when classification has taken place

that is automatically computed by the reasoner is called the *inferred hierarchy*. To automatically classify the ontology (and check for inconsistencies) the '**Classify Taxonomy...**' action should be used. This can be invoked via the OWL menu (Figure 4.39), or by using the '**Classify taxonomy**' button on the Protégé-OWL toolbar shown in Figure 4.42. To check the consistency of the ontology, the '**Check consistency...**' action should be used, which can be invoked from the OWL menu, or by using the '**Check consistency**' button on the Protégé-OWL toolbar. When the inferred hierarchy has been computer, an *inferred hierarchy* window will pop open next to the existing *asserted hierarchy* window as shown in Figure 4.43. If a class has been reclassified (i.e. if it's 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 circled in red.

Vocabulary

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

53
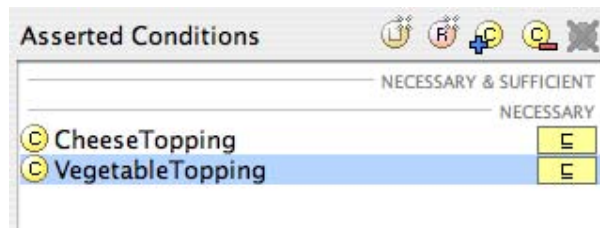
**Figure 4.44:** The Conditions Widget Displaying ProbeInconsistentTopping

### 4.9.4 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 MeatTopping. 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** ProbeInconsistentTopping **which is a subclass of both** CheeseTopping **and** Vegetable

1. Select the class CheeseTopping from the class hierarchy on the OWLClasses tab.

2. Create a subclass of CheeseTopping named ProbeInconsistentTopping.

3. Add a comment to the ProbeInconsistentTopping class that is something along the lines of, "This class should be inconsistent when the ontology is classified.". This will enable anyone who looks at our pizza ontology to see that we deliberately meant the class to be inconsistent.

4. Ensure that the ProbeInconsistentTopping class is selected in the class hierarchy, and then select the "NECESSARY" header in the '**Conditions widget**'.

5. Click on the '**Add named class**' button on the '**Conditions Widget**'. 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 necessary condition (as a superclass), so that the conditions widget should look like the picture in Figure 4.44.
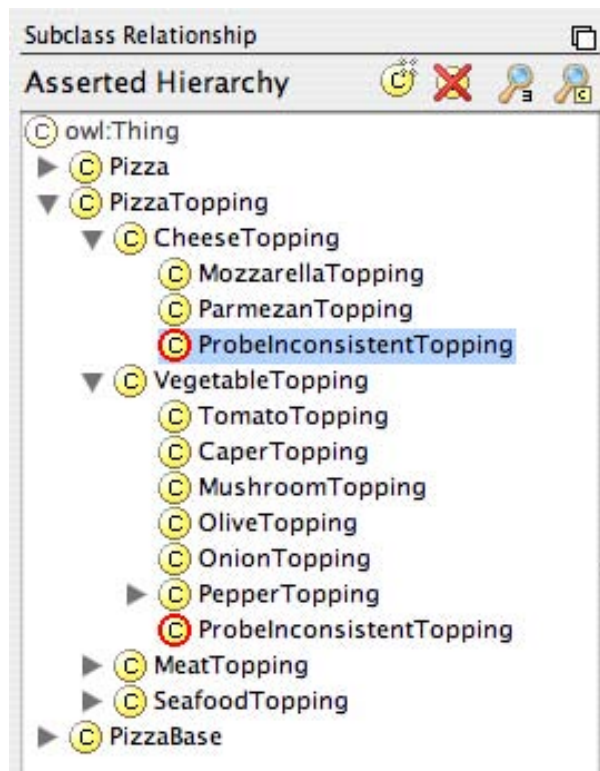
**Figure 4.45:** The Class ProbeInconsistentTopping found to be inconsistent by the reasoner

**MEANING**

**?⁈?**

If we study the class hierarchy, ProbeInconsistentTopping should appear as a subclass of CheeseTopping and as a subclass of VegetableTopping. This means that ProbeInconsistentTopping is a CheeseTopping *and* a VegetableTopping. More formally, all individuals that are members of the class ProbeInconsistentTopping 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** ProbeInconsistentTopping **is inconsistent**

1. Press the '**Classify Taxonomy**' button on the OWL toolbar 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 4.45 — notice that the class ProbeInconsistentTopping is circled in red, indicating that the reasoner has found this class to be inconsistent (i.e. it cannot possibly have any individuals as memebers).

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 using the Wizard. 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 widget**' should contain CheeseTopping's sibling classes: VegetableTopping, SeafoodTopping and MeatTopping. Select VegetableTopping in the Disjoints widget.

3. Press the '**Remove selected class from list**' button on the Disjoints widget (shown in Figure 4.5) to remove the disjoint axiom that states CheeseTopping and MeatTopping are disjoint.

4. Press the '**Classify Taxonomy**' button on the OWL toolbar 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 ProbeInconsistentTopping is no longer inconsistent! This means that individuals which are members of the class ProbeInconsistent-Topping 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 widget**' should contain MeatTopping and SeafoodTopping.

3. Press the '**Add disjoint class**' button on the disjoint classes widget 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 the '**Classify Taxonomy**' button on the OWL toolbar to send the ontology to the reasoner. After a few seconds the ontology should have been classified, and ProbeInconsistentTopping 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
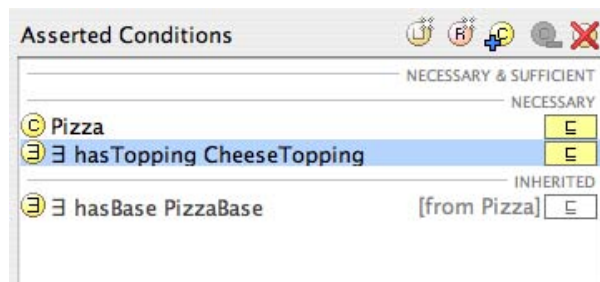
Figure 4.46: The Description of CheesyPizza (Using Necessary Conditions)

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 '**OWLClasses**' tab.

2. Press the '**Create subclass**' button to create a subclass of Pizza. Rename it to CheesyPizza.

3. Make sure that CheesyPizza is selected in the class hierarchy. Select the "NECES-SARY" header in the conditions widget. (You may have to select the '**Asserted**' tab on the '**Conditions Widget**' — the automatically shows the '**Inferred**' tab after classification).

4. Press the '**Create restriction**' button on the conditions widget to display the '**Create restriction dialog**'.

5. Select '∃ **someValuesFrom**' as the type of restriction to be created.

6. Select hasTopping as the property to be restricted.

7. In the filler edit box type `CheeseTopping` (or use the '**Insert class**' button to display a dialog from which CheeseTopping can be selected). Press '**OK**' to close the dialog and create the restriction.

The '**Conditions Widget**' should now look like the picture shown in Figure 4.46.

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 Cheese-Topping.

Our current description of CheesyPizza says that if something is a CheesyPizza it is necessarily a Pizza and it is *necessary* for it to have *at least one* topping that is a kind of CheeseTopping. We have used *necessary* conditions to say this. Now consider some (random) individual. Suppose that we know that

this individual is a member of the class Pizza. We also know that this individual has at least one kind of CheeseTopping. However, given our current description of CheesyPizza this knowledge is not *sufficient* to determine that the individual is a member of the class CheesyPizza. To make this possible we need to change the conditions for CheesyPizza from *necessary* conditions to *necessary AND sufficient* conditions. This means that not only are the conditions *necessary* for membership of the class CheesyPizza, they are also *sufficient* to determine that any (random) individual that satisfies them must be a member of the class CheesyPizza.

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 conditions widget 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 '**Conditions Widget**' select the ∃ hasTopping CheeseTopping restriction.

3. Drag the ∃ hasTopping CheeseTopping restriction from under the "NECESSARY" header to *on top* of the "NECESSARY & SUFFICIENT" header.

4. Select the class Pizza.

5. Drag the class Pizza from under the "NECESSARY" header to *on top* of the ∃ hasTopping CheeseTopping restriction (note *not* on top of the "NECESSARY & SUFFICIENT" header this time).

The '**Conditions Widget**' should now look like the picture shown in Figure 4.47.
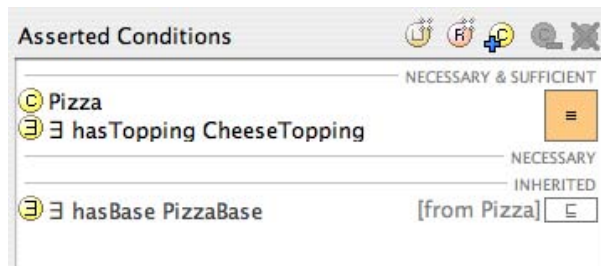
**Figure 4.47:** The Description of CheesyPizza (Using Necessary AND Sufficient Conditions)



If an individual is a member of 'NamedClass' then it must satisfy the conditions. However if some individual satisfies these necessary conditions, we cannot say that it is a member of 'Named Class' (the conditions are not 'sufficient' to be able to say this) - this is indicated by the direction of the arrow.



If an individual is a memeber of 'NamedClass' then it must satisfy the conditions. If some individual satisfies the condtions then the individual must be a member of 'NamedClass' - this is indicated by the double arrow.

**Figure 4.48:** 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 4.48.

If you accidentally dropped Pizza onto the "NECESSARY & SUFFICIENT" header (rather that onto the ∃ hasTopping CheeseTopping) in Exercise 29 the conditions widget will look like the picture shown in Figure 4.49. 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 CheeseTopping restriction.
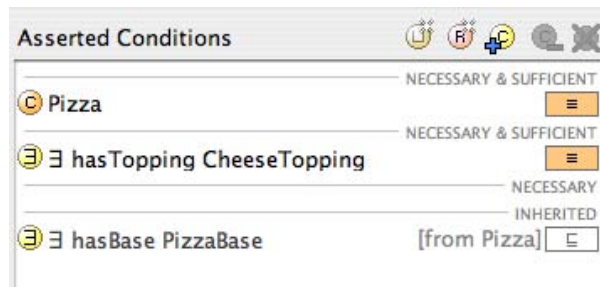
**Figure 4.49:** An INCORRECT description of CheesyPizza

---

TIP

Conditions may also be transferred from "NECESSARY" to "NECESSARY & SUFFICIENT" 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.

---

NOTE

In OWL it is possible to have multiple sets of necessary & sufficient conditions. This is discussed later in section 6.5

---

### 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 know as *primitive* classes. In Protégé-OWL *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.
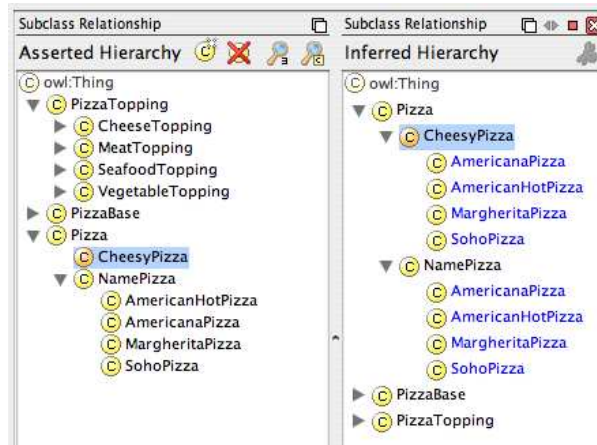
**Figure 4.50:** The Asserted and Inferred Hierarchies Displaying The Classification Results For CheesyPizza

## 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 technique[13] helps to keep the ontology in a maintainable and modular state. Not only does this promote the reuse of the ontology by other ontologies and applications, it also 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. Ensure that a reasoner (RACER) is running.

2. Press the '**Classify Taxonomy...**' button on the toolbar (See Figure 4.42).

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 4.50. Figures 4.51 and 4.52 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.

---

[13]Sometimes know as ontology normalisation.
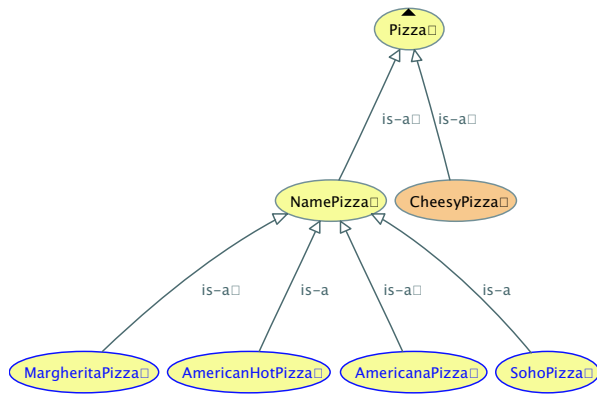
**Figure 4.51:** OWLViz Displaying the Asserted Hierarchy for CheesyPizza
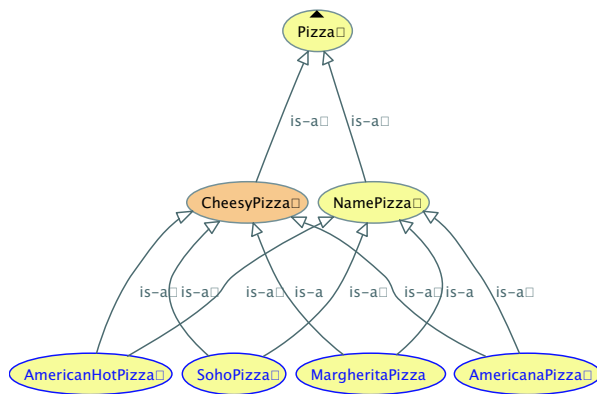


**Figure 4.52:** OWLViz Displaying the Inferred Hierarchy for CheesyPizza

**MEANING**

**??**

The reasoner has determined that MargheritaPizza, AmericanaPizza, American-HotPizza 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 Cheese-Topping[a] the reasoner has determined that these classes must be subclasses of CheeseTopping.

---

[a]Or toppings that belong to the subclasses of CheeseTopping

---

⚠

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

---

[a]The 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 4.53. The '**Classification Results**' pane pops open after classification at the bottom of the Protégé-OWL 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 4.53, 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 ($\exists$). Existential restrictions specify the existence of *at least one* relationship along a given property to an individual that is a member of a specific class (specified by the filler). However, existential restrictions do not mandate that the *only* relationships for the given property that can exist must be to individuals that are members of the specified filler class.
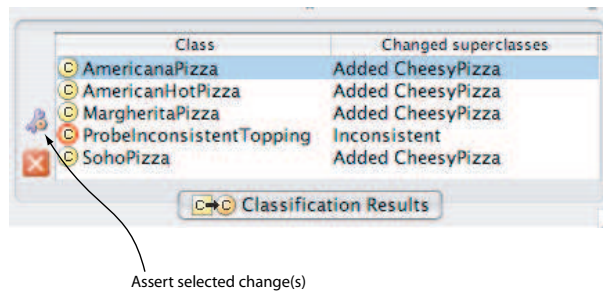
**Figure 4.53:** The Classification Results Pane

For example, we could use an existential restriction ∃ hasTopping 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 ∀. They *constrain* the relationships along a given property to individuals that are members of a specific class. For example the universal restriction ∀ 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 *All Restrictions.*

The above universal restriction ∀ 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*

65

*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 "NECESSRY" header in the '**Conditions Widget**'.

3. Press the '**Create restriction**' button on the '**Conditions Widget**' to display the '**Create restriction dialog**'.

4. Select the type of restriction as '∀ **allValuesFrom**' in order to create a universally quantified restriction.

5. Select hasTopping as the property to be restricted.

6. For the filler we want to say CheeseTopping *or* VegetableTopping. First insert the class CheeseTopping either by typing `CheeseTopping` into the filler box, or by using the '**Insert class**' button. We now need to use the *unionOf* operator between the class names. The unionOf operator may be inserted using the button shown in Figure 4.54[a]. Insert the unionOf symbol by pressing the '**Insert unionOf**' button on the expression builder panel. Next insert the class VegetableTopping either by typing it or by using the '**Insert class button**'. You should now have CheeseTopping ⊔ VegetableTopping in the filler edit box.

7. Press the '**OK**' button on the dialog to close the dialog and create the restriction — if there are any errors (due to typing errors etc.) the dialog will not close and an error message will be displayed at the bottom of the expression builder panel.

---

[a]See section B.2 for more information about union classes.

---

At this point the conditions widget should look like the picture shown in Figure 4.55.
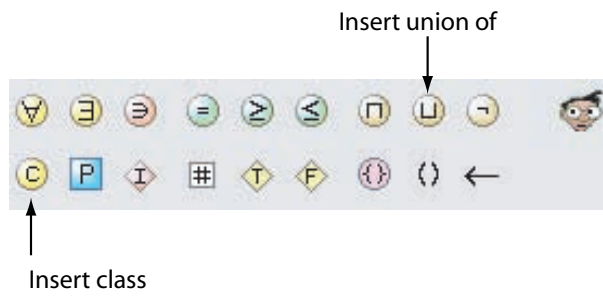
**Figure 4.54:** Using the Expression Builder Panel to insert Union Of
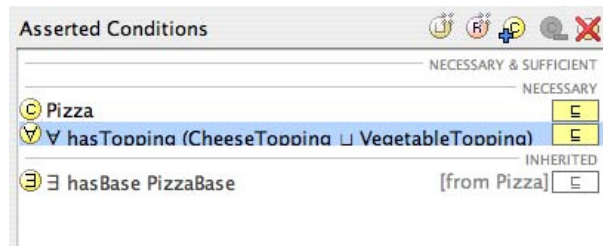


**Figure 4.55:** The Description of VegetarianPizza (Using Necessary Conditions)

**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* (∀ 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.

**TIP**

Instead of using the '**Insert unionOf**' button in  Exercise 31 , we could have simply typed **or** into the filler edit box and it would have automatically been converted to the union of symbol (⊔).

In situations like the above example, a common mistake is to use an *intersection* instead of a *union*. For example, CheeseTopping ⊓ 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 4.9.4. 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 (∀ hasTopping CheeseTopping) and one for VegetableTopping (∀ hasTopping VegetableTopping). However, when multiple restrictions are used (for any type of restriction) the total description is taken to be the intersection of the individual restrictions. This would have therefore been equivalent to one restriction with a filler that is the *intersection* of MozzarellaTopping and TomatoTopping — as explained above this would have been logically incorrect.

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

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

1. Ensure that VegetarianPizza is selected in the class hierarchy.

2. On the '**Conditions Widget**' select the (∀ universal) restriction on the hasTopping property.

3. Drag the hasTopping restriction from under the "NECESSARY" header to *on top* of the "NECESSARY & SUFFICIENT" header.

4. Select the class Pizza.

5. Drag the class Pizza from under the "NECESSARY" header to *on top* of the hasTopping restriction (note *not* on top of the "NECESSARY & SUFFICIENT" header this time).

The '**Conditions Widget**' should now look like the picture shown in Figure 4.56.

**Figure 4.56:** The Conditions Widget Displaying the Definition of VegetarianPizza (Using Necessary and Sufficient Conditions)

> **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 it's 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 4.48.

## 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. Ensure that a resoner (RACER) is running. Press the '**Classify taxonomy**' button.

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*[14] on the hasTopping property.

### 4.13.1   Closure Axioms

A *closure axiom* on a property consists of a universal restriction that acts along the property to say that it can *only* be filled by the specified fillers. The restriction has a filler that is the *union* of the fillers that occur in the existential restrictions for the property[15]. For example, the closure axiom on the hasTopping property for MargheritaPizza is a universal restriction that acts along the hasTopping property, with a filler that is the *union of* MozzarellaTopping and also TomatoTopping. i.e. ∀ hasTopping (MozzarellaTopping ⊔ 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 '**OWLClasses**' tab.

2. Select the "NECESSARY" header in the '**Conditions Widget**'.

3. Press the '**Create restriction**' button on the conditions widget to display the '**Create Restriction dialog**'.

4. Select the restriction type as '∀ **allValuesFrom**' (universal restriction).

5. Select hasTopping as the property to be restricted.

6. In the filler edit box enter MozzarellaTopping ⊔ TomatoTopping. This can be done by typing `MozzarellaTopping or TomatoTopping` into the filler edit box ("or" will be automatically converted to ⊔ as the filler is typed). This can also be accomplished by using the '**Insert class**' button and the '**Insert unionOf**' button to insert the class MozzarellaTopping, then insert the unionOf symbol and then insert the class TomatoTopping.

7. Press the OK button to create the restriction and add it to the class MargheritaPizza.

The conditions widget should now appear as shown in Figure 4.57.

---

[14]Also referred to as a closure restriction.

[15]And technically speaking the classes for the values used in any hasValue restrictions (see later).
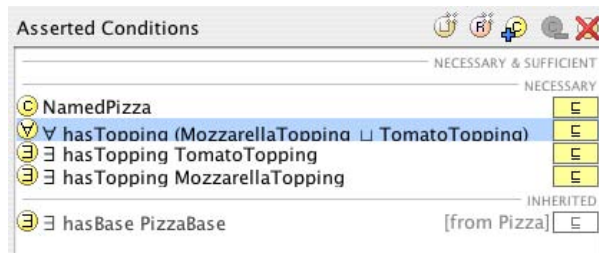
**Figure 4.57:** Conditions Widget: Margherita Pizza With a Closure Axiom for the hasTopping property

**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 ∀ hasTopping (MozzarellaTopping ⊔ 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. Select SohoPizza in the class hierarchy on the '**OWLClasses**' tab.

2. Select the "NECESSARY" header in the '**Conditions Widget**'.

3. Press the '**Create restriction**' button to display the '**Create Restriction dialog**'.

4. Select the restriction type as, '∀ **allValuesFrom**', as we want to create a universally quantified restriction.

5. Select hasTopping as the property to be restricted.

6. In the filler edit box enter the union of the toppings for SohoPizza by typing `ParmezanTopping or MozzarellaTopping or TomatoTopping or OliveTopping`. Note that the "or" keywords will automatically be converted to the *unionOf* symbol (⊔) as you type to give "ParmezanTopping ⊔ MozzarellaTopping ⊔ TomatoTopping ⊔ OliveTopping".

7. Press the OK button to create the restriction and close the dialog. If the dialog will not close due to errors, check that the class names have been spelt correctly.

---

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-OWL 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 OWLClasses tab.

2. In the '**Conditions Widget**' 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. 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** AmericanHotPizza

1. Select AmericanHotPizza in the class hierarchy on the OWLClasses tab.

2. In the '**Conditions Widget**' 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 taxonomy**' button on the OWL toolbar 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 4.58 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.
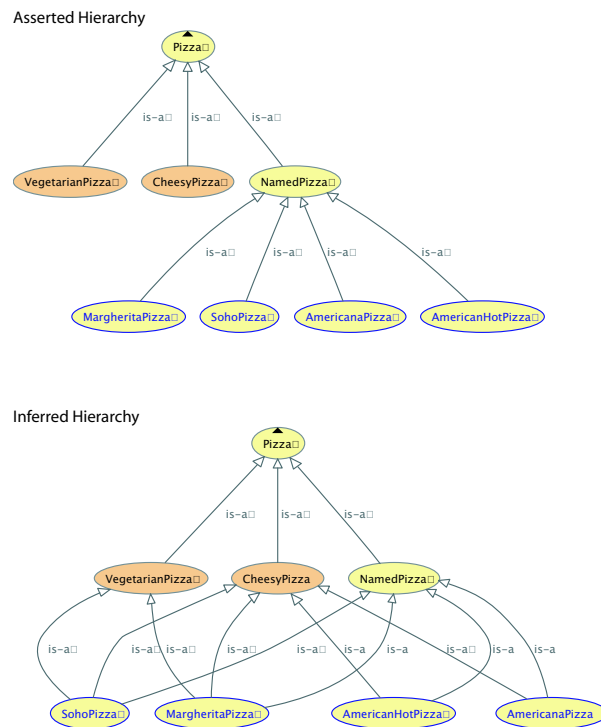
**Figure 4.58:** The asserted and inferred hierarchies showing the "before and after" classification of Pizzas into CheesyPizzas and VegetarianPizzas.

## 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 SpicynessValuePartition 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.

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 Wizards package contains a wizard for creating ValuePartitions – appropriately named the 'Create ValuePartition' wizard.

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. Select '**Create Value Partition**' from the Wizards menu on the Protégé menu bar to invoke the ValuePartition wizard.

2. On the first page of the wizard type `SpicinessValuePartition` as the name of the ValuePartition class and press the Next button.

3. Now enter `hasSpiciness` for the ValuePartition property name, and press the '**Next**' button.

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` and press return. This will create Mild, Medium and Hot as subclasses of the SpicinessValuePartition class. Press the '**Next**' button to continue.

5. The ValuePartition names will be verified. Press the '**Next**' button.

6. The annotations page will be visible. At this point we could add annotations to the ValuePartition if we wanted. However, at the moment we won't, so press the '**Next**' button to continue.

7. The final page of the wizard prompts us to specify a class that will act as a 'root' under which all ValuePartitions will be created. We recommend that ValuePartitions are created under a class named ValuePartition, which is the default option. Press the Finish button to create the ValuePartition.

Let's look at what the wizard has done for us (refer to Figure 4.59 and Figure 4.60):

1. A ValuePartition class has been created as a subclass of owl:Thing.

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.
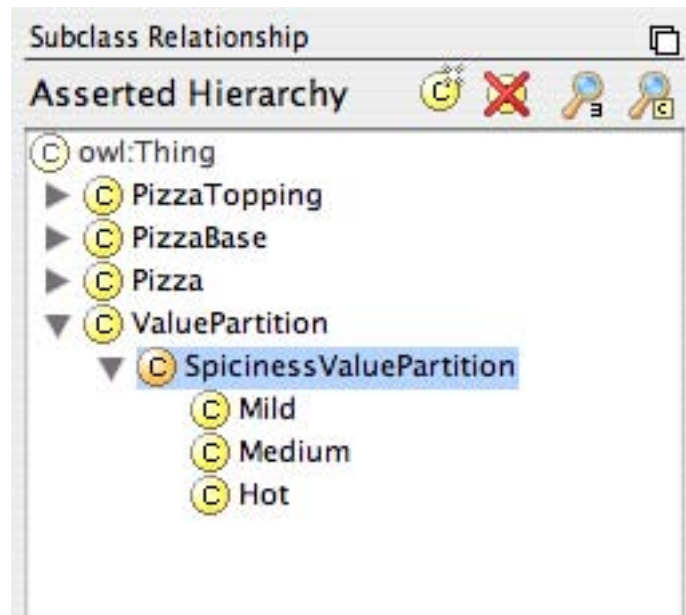
**Figure 4.59:** Classes Added by the 'Create ValuePartition' Wizard



**Figure 4.60:** The Conditions Widget Displaying the Description of the SpicinessValuePartition Class

5. A class that is the *union* of Mild, Medium and Hot has been created as the subclass of Spiciness-ValuePartition (see Figure 4.60).

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 if A an individual may be a member of A without being a member of either B or C.

In Protégé-OWL 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 ⊔ C. The effect of a covering axiom is depicted in Figure 4.61.
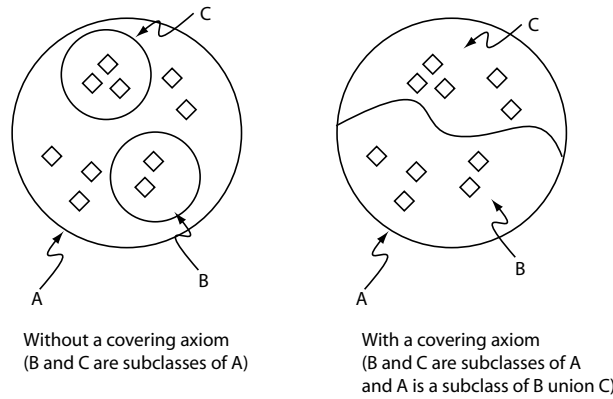


**Figure 4.61:** A schematic diagram that shows the effect of using a Covering Axiom to cover class A with classes B and C

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 ⊔ Medium ⊔ 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 4.62. 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 — SpicynessValuePartition is *not* covered by Mild, Medium and Hot. Contrast this with the case when a covering axiom *is* used. It can be seen that if an individual is a member of the class SpicinessValuePartition, it *must* be a member of one of the three subclasses Mild, Medium or Hot — SpicinessValuePartition is *covered* by Mild, Medium and Hot.
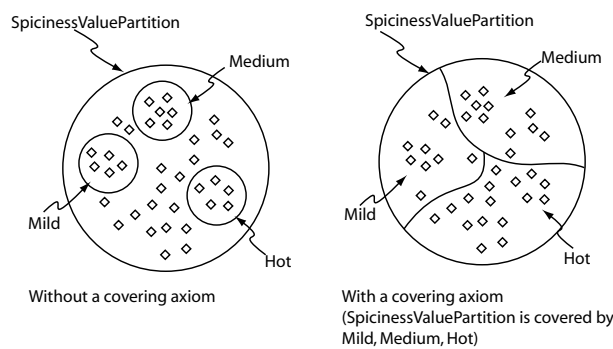


**Figure 4.62:** The effect of using a covering axiom on the SpicinessValuePartition

## 4.15 Using the Properties Matix 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 SpicynessValuePartition**, 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 *Properties Matrix Wizard* can help to speed things up. The properties matrix wizard can be used to add existential restrictions along specified properties to many classes in a quick and efficient manner.

---

**Exercise 40: Use the properties matrix wizard to specify the spiciness of pizza toppings**

---

1. Invoke the *property matrix* wizard by selecting the '**Properties Matrix**' item from the '**Wizards**' menu on the Protégé menu bar.

2. The first page to be displayed in the property matrix wizard is the classes selection page shown in Figure 4.63. By selecting toppings in the class hierarchy, and using the buttons in the middle of the page ('>>' and '<<') classes may be transferred to the right hand side list. Select all of the pizza topping classes and transfer them to the right hand side list as shown in Figure 4.63. You should only select the classes that are 'actual' toppings, so classes such as **CheeseTopping** should not be selected. After selecting the toppings press the '**Next**' button on the wizard.

3. The wizard should now be displaying the page shown in Figure 4.64. Select the **has-Spiciness** property and use the (>>) button to move the property to the right hand column (as shown in Figure 4.64). Press the '**Next**' button on the wizard.

4. In the final page on the wizard, the property fillers should be specified. This is done by double clicking on each class that is listed and selecting a filler of either **Mild**, **Medium** or **Hot**. Select fillers of **Mild** for everything except **PepperoniTopping** and **SalamiTopping**, which should have fillers of **Medium**, and **JalapenoPepperTopping** and **SpicyBeef**, which should have fillers of **Hot**. After selecting fillers, the wizard page should resemble Figure 4.65.

5. Press the '**Finish**' button to create the restrictions on the toppings and close the wizard. After the wizard has closed, select some different toppings and notice that they have restrictions on the along the **hasSpiciness** property, with fillers of subclasses of the **SpicinessValuePartition**.

---

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
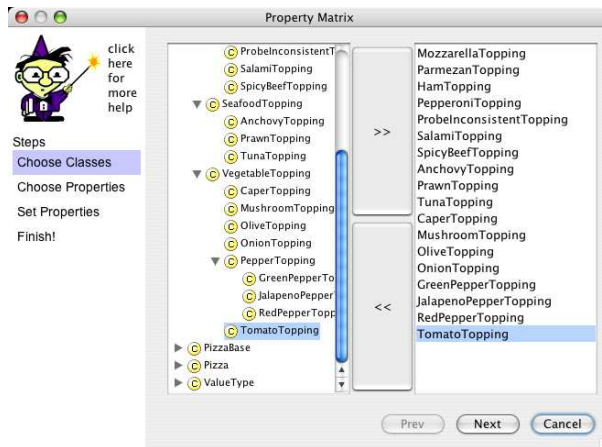
**Figure 4.63:** Property Matrix Wizard: Class Selection Page
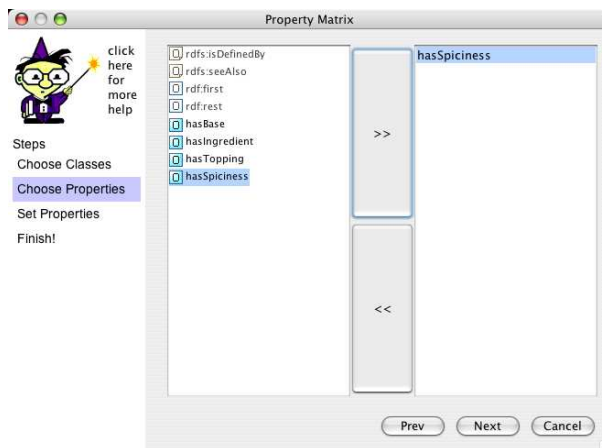


**Figure 4.64:** Property Matrix Wizard: Property Selection Page
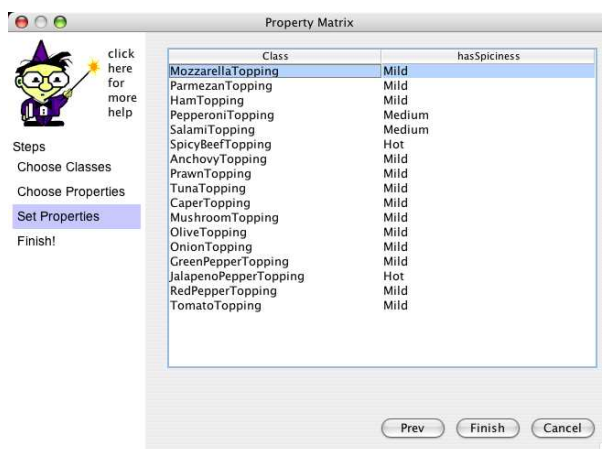


**Figure 4.65:** Property Matrix Wizard: Restriction Fillers Page

**Figure 4.66:** 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 "NECESSARY & SUFFI-CIENT" header in the conditions widget.

3. Press the '**Create restriction**' button on the conditions widget to show the 'Create Restriction Dialog'.

4. Select '∃ **someValuesFrom**' as the type of restriction.

5. Select **hasTopping** as the property to be restricted.

6. The filler should be: **PizzaTopping** ⊓ ∃ **hasSpiciness Hot**. This filler describes an anonymous class, which contains the individuals that are members of the class **Pizza-Topping** 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 **Piz-zaTopping**s and have a spiciness that is **Hot**. To enter this restriction as a filler type, `PizzaTopping and some hasSpiciness Hot`. The "and" keyword will be converted to the intersection symbol ⊓, the "some" keyword will be converted to the existential quantifier symbol ∃.

7. The '**Create Restriction Dialog**' should now appear similar to the picture shown in Figure 4.67. Press the OK button to close the dialog and create the restriction.

8. Finally, drag **Pizza** from under the "NECESSARY" header to on top of the newly created restriction (∃ **hasTopping** (**PizzaTopping** ⊓ ∃ **hasSpiciness Hot**)).

---

The conditions widget should now look like the picture shown in Figure 4.66
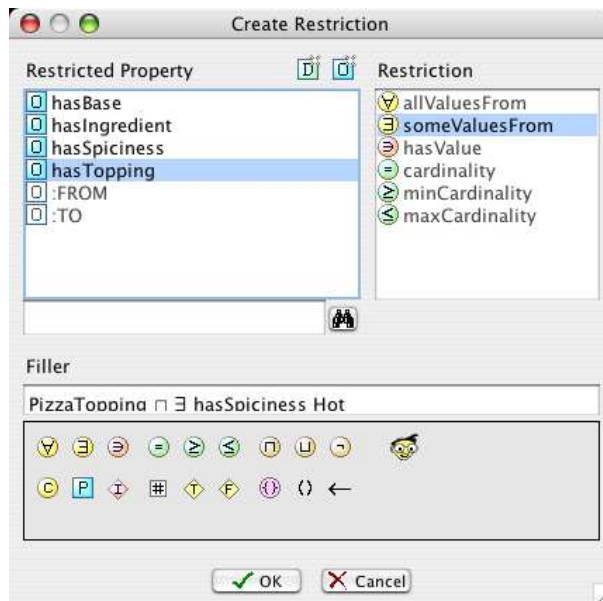
**Figure 4.67:** Create Restriction Dialog: A Restriction Describing a Spicy Topping

**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 ⊓ ∃ hasSpiciness 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 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 Hot. We could have then used ∃ hasTopping 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 the '**Classifiy Taxonomy**' button on the OWL toolbar to invoke the reasoner and classify the ontology.

**Figure 4.68:** Cardinality Restrictions: Counting Relationships

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 givien 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 4.68 depicts the individual Matthew related to the individuals Nick and the individual Hai via the worksWith property. The individual Matthew satisfies a *minimum cardianlity* restriction of 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
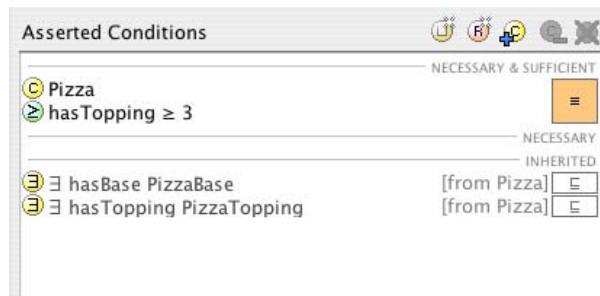
**Figure 4.69:** The Conditions Widget Displaying the Description of an InterestingPizza

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 OWLClasses tab and make sure that the Pizza class is selected.

2. Create a subclass of Pizza called InterestingPizza.

3. Select the "NECESSARY & SUFFICIENT" header in the conditions widget.

4. Press the '**Create restriction**' button to bring up the '**Create restriction dialog**'.

5. Select '≥ **minCardinality**' as the type of restriction to be created.

6. Select hasTopping as property to be restricted.

7. Specify a minimum cardinality of three by typing 3 into the restriction filler edit box.

8. Press the '**OK**' button to close the dialog and create the restriction.

9. The conditions widget should now have a "NECESSARY" condition of Pizza, and a "NECESSARY & SUFFICIENT" condition of hasTopping ≥ 3. We need to make Pizza part of the necessary and sufficient conditions. Drag Pizza and drop it *on top of* the hasTopping ≥ 3 condition.

The conditions widget should now appear like the picture shown in Figure 4.69.

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 the '**Classify Taxonomy**' button on the OWL toolbar.

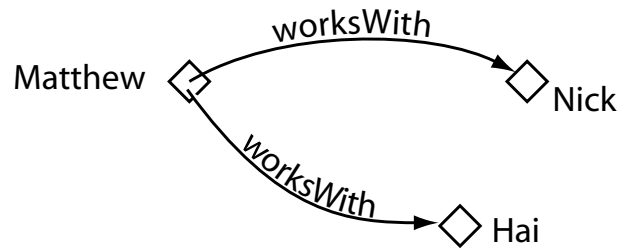After the reasoner has classified the ontology, the '**Inferred Hierarchy**' window will pop open. Expand the hierarchy so that InterestingPizza is visible. Notice that InterestingPizza now has subclasses AmericanaPizza, AmericanHotPizza and SohoPizza — notice MargheritaPizza has *not* been classified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

# Chapter 5

# More On Open World Reasoning

This 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 45: Create** NonVegetarianPizza **as a subclass of** Pizza **and make it disjoint to** Vegetarian-
Pizza

---

1. Select Pizza in the class hierarchy on the '**OWLClasses**' tab. Press the '**Create subclass**' button to create a new class as the subclass of Pizza.

2. Rename the new class to NonVegetarianPizza.

3. Make NonVegetarianPizza disjoint with VegetarianPizza — while NonVegetarian-Pizza is selected, press the '**Add named class**' button on the disjoint classes widget (Figure 4.5).

---

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

**Exercise 46: Make** VegetarianPizza **the complement of** VegetarianPizza

1. Make sure that NonVegetarianPizza is selected in the class hierarchy on the '**OWLClasses tab**'.

2. Select the "NECESSARY & SUFFICIENT" header in the '**Conditions Widget**'.

3. Press the '**Create new expression**' button, which will 'pop' open an 'inline expression editor' in the '**Conditions Widget**' as shown in Figure 5.1. The inline expression editor contains an edit box for typing expressions into, and the expression builder panel (the same one that is found in the '**Create restriction dialog**'), which can be used to insert class names and logical symbols into the edit box.

4. Type `not VegetarianPizza` into the edit box. The "not" keyword will be converted into the '*complement of*' symbol (¬). Alternatively, to enter the expression using the expression builder panel, use the '**Insert complementOf**' button shown in Figure 5.3 to insert the *complementOf* symbol, and the use the '**Insert class**' button (Figure 5.3) to display a dialog from which VegetarianPizza can be selected.

5. 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 expresion editor and pressed the tab key, the choices to complete the word Vege would have poped up in a list as shown in Figure 5.2. 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 conditions widget should now resemble to picture shown in 5.4. 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.
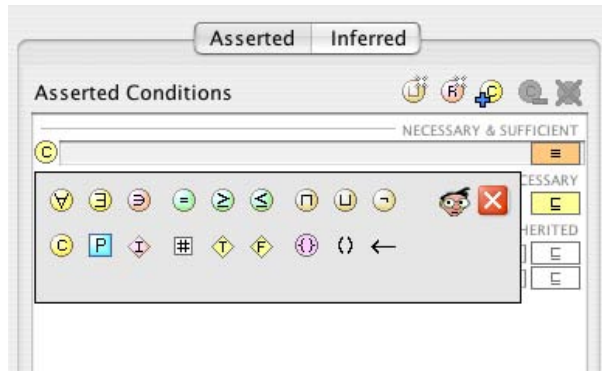
**Figure 5.1:** Conditions Widget: Inline Expression Editor
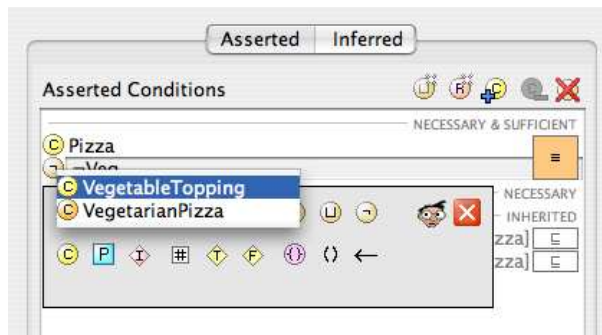


**Figure 5.2:** Conditions Widget: Inline Expression Editor Auto Completion
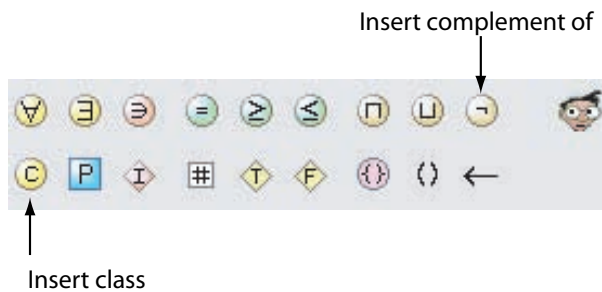


**Figure 5.3:** Using the Expression Builder Panel to insert Complement Of



**Figure 5.4:** The Conditions Widget Displaying the Intermediate Step of Creating a Definition forNonVegetarianPizza
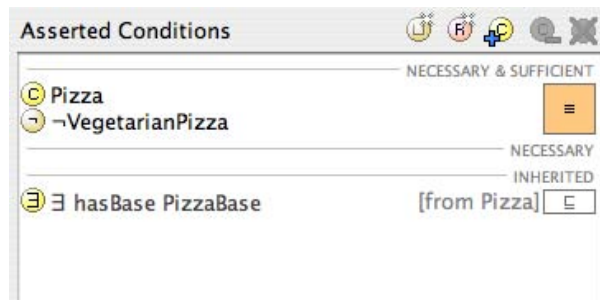
**Figure 5.5:** The Conditions Widget Displaying the Definition for NonVegetarianPizza

**Exercise 47: Add** Pizza **to the necessary and sufficient conditions for** NonVegetarianPizza

1. Make sure NonVegetarianPizza is selected in the class hierarchy on the '**OWLClases**' tab.

2. Select Pizza in the '**Conditions Widget**'.

3. Drag Pizza from under the "NECESSARY" header, and drop it onto the '¬ VegetarianPizza' condition to add it to the same set of *necessary and sufficient* conditions as ¬ VegetarianPizza.

The '**Conditions Widget**' should now look like the picture shown in Figure 5.5.

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 48: Use the reasoner to classify the ontology**

1. Press the '**Classify taxonomy**' button on the OWL 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 5.6. As can be seen, Margher-

itaPizza 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 49: 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 '**Conditions Widget**' select the "NECESSARY" header.

3. Press the '**Create restriction**' button to display the '**Create restriction dialog**'.

4. Select '∃ **someValuesFrom**' in order to create an existential restriction.

5. Select hasTopping as the property to be restricted.

6. Type `MozzarellaTopping` into the filler edit box to specify that the toppings must be individuals that are members of the class MozzarellaTopping.

7. Press the '**OK**' button to close the dialog and create the restriction.

---

**MEANING**

**?❔?**

> If an individual is a member of UnclosedPizza it is necessary for it to be a Named-Pizza 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 50: Use the reasoner to classify the ontology**

---

1. Press the '**Classify taxonomy**' button on the OWL toolbar.

---

Examine the class hierarchy. Notice that UnclosedPizza is neither a VegetarianPizza or NonVegetarianPizza.
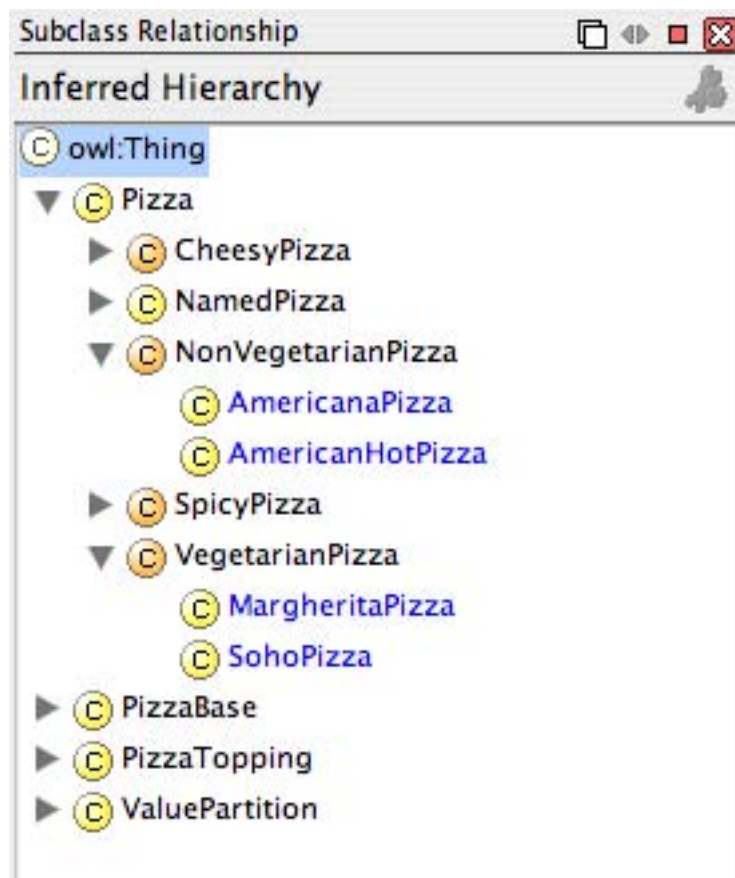
**Figure 5.6:** The Inferred Class Hierarchy Showing Inferred Subclasses of VegetarianPizza and NonVegetarian-Pizza

**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 Unclosed-Pizza 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, UnclosePizza cannot be classified as a NonVegetarian-Pizza.

# Chapter 6

# Creating Other OWL Constructs In Protégé-OWL

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

## 6.1 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 6.2 and section 6.3 respectively. To create individuals in Protégé-OWL 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
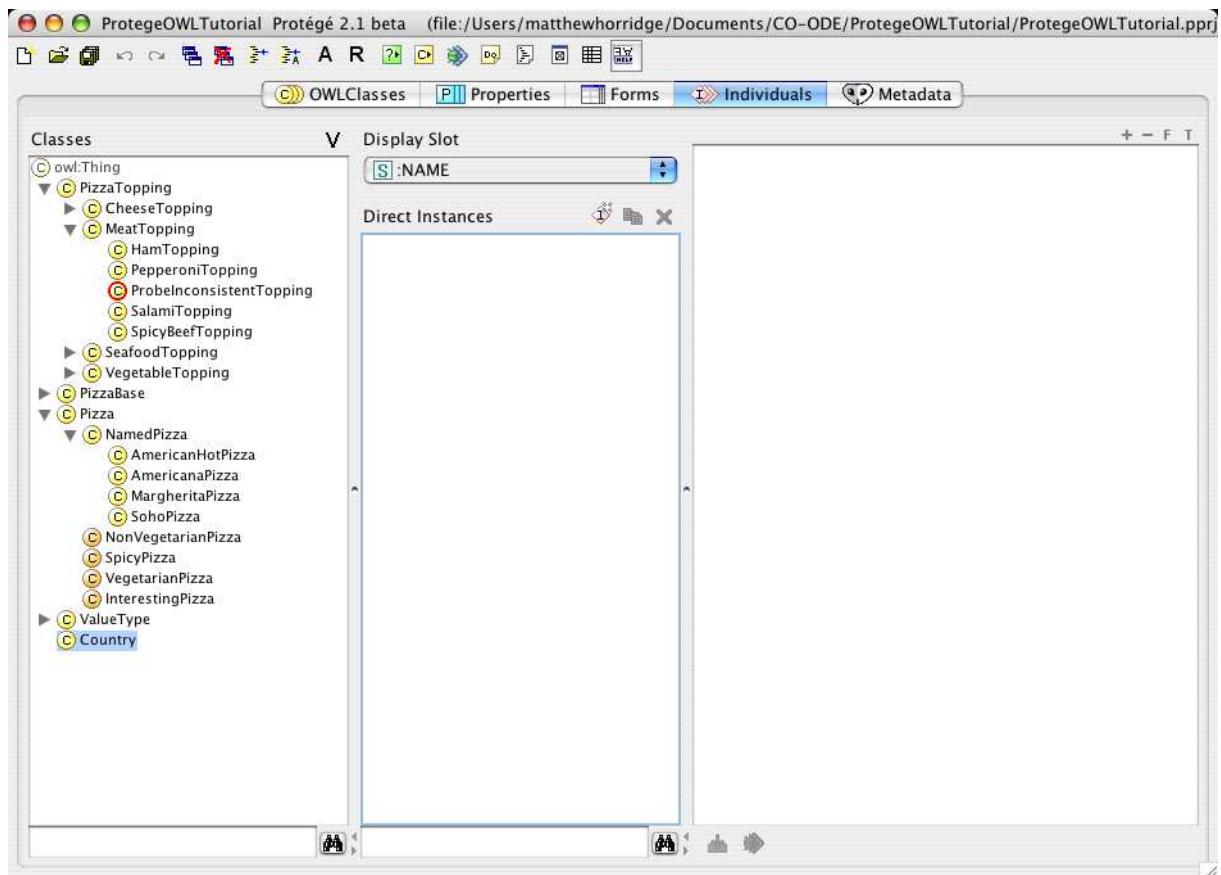
**Figure 6.1:** The Individuals Tab

will create a class Country and then 'populate' it with individuals:

**Exercise 51: Create a class called** Country **and populate it with some individuals**

1. Create Country as a subclass of owl:Thing.

2. Switch to the '**Individuals Tab**' shown in Figure 6.1 and select the class Country in the '**Classes**' tree.

3. Press the '**Create Instance**' button shown in Figure 6.2. (Remember that 'Instance' is another name for 'Individual' in ontology terminology).

4. An individual that is a member of Country will be created with a auto-generated name. Rename the individual using the '**Name**' widget (located on the individuals tab to the right of the class view and instances list) to Italy.

5. Use the above steps to create some more individuals that are members of the class Country called America, England, France, and Germany.
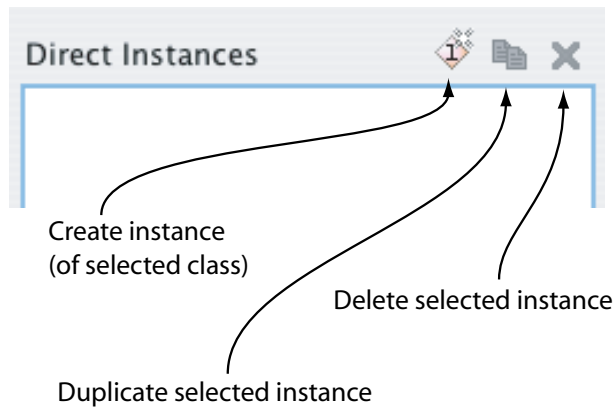
**Figure 6.2:** Instances Manipulation Buttons
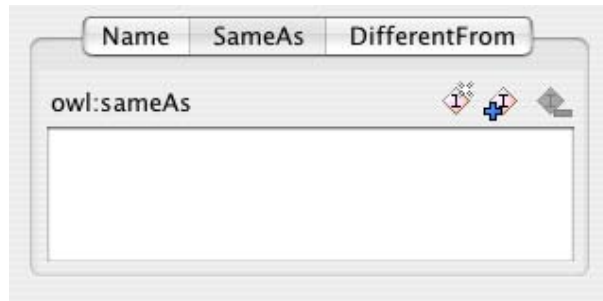


**Figure 6.3:** The SameAs Widget

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-OWL these assertions can be made using the '**SameAs**' and '**DifferentFrom**' tabs shown in Figure 6.3, which are located with the '**Name**' widget on the '**Individuals**' tab.

Having created some individuals we can now use these individuals in class descriptions as described in section 6.2 and section 6.3.

## 6.2 hasValue Restrictions

A hasValue restriction, denoted by the symbol ∋, describes the set of individuals that have *at least one* relationship along a specified property to a *specific individual*. For example, the hasValue restriction hasCountryOfOrigin ∋ 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 A.2.

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 6.4:** The Conditions Widget Displaying The hasValue Restriction for MozzarellaTopping

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

**Exercise 52: Create a hasValue restriction to specify that** MozzarellaTopping **has** Italy **as its country of origin.**

1. Switch to the '**Properties**' tab. Create a new object property and name it hasCountryOfOrigin.

2. Switch to the '**OWLClasses**' tab and select the class MozzarellaTopping.

3. Select the "NECESSARY" header in the '**Conditions Widget**'.

4. Press the '**Create restriction**' button on the '**Conditions Widget**' to bring up the '**Create restriction dialog**'.

5. Select ∋ hasValue as the type of restriction to be created.

6. Select hasCountryOfOrigin as the property to be restricted.

7. In the restriction filler box enter the individual Italy as a filler — either type `Italy` into the filler edit box, or, press the '**Insert individual**' button on the expression builder panel, which will pop open a dialog box from which the individual Italy may be selected.

8. Press '**OK**' to close the dialog and create the restriction.

The '**Conditions Widget**' should now look similar to the picture shown in Figure 6.4.

**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.

94

| | 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. |
|---|---|

## 6.3 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-OWL enumerated classes are defined using the '**Conditions Widget**' 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-OWL by creating the enumeration as a "NECESSARY & SUFFICIENT" condition.

---

**Exercise 53: Convert the class** Country **into an enumerated class**

---

1. Switch the the '**OWLClasses**' tab and select the class Country.

2. Select the "NECESSARY & SUFFICIENT" header in the '**Conditions Widget**'.

3. Press the '**Create new expression**' button. The inline expression editor will pop open.

4. Type `{America England France Germany Italy}` into the expression edit 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 '**Conditions Widget**' should now look similar to the picture shown in Figure 6.5.

**Figure 6.5:** The Conditions Widget Displaying An Enumeration Class

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 6.6.

---

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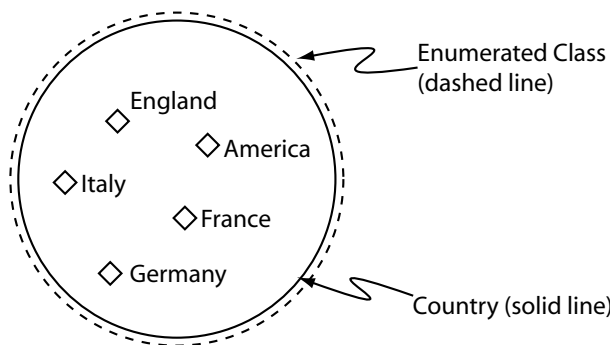


**Figure 6.6:** A Schematic Diagram Of The **Country** Class Being Equivalent to an Enumerated Class

TIP

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

## 6.4 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 constaints are:

- The filler for annotation properties must either be a data literal[1], 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. The 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:

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 the Protégé-OWL plugin. 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 the 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.

To create annotation properties the '**Create annotation datatype property**' and '**Create annotation object property**' buttons on the '**Properties**' tab should be used. To use annotation properties the annotations widgets shown in Figure 6.7 is used. An annotations widget is located on the OWL-Classes, Properties, Individuals and Metadata 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 conditions widget and selecting '**Edit annotation properties...**'.

---

[1]A data literal is the character representation of a datatype value, for example, "Matthew", 25, 3.11.
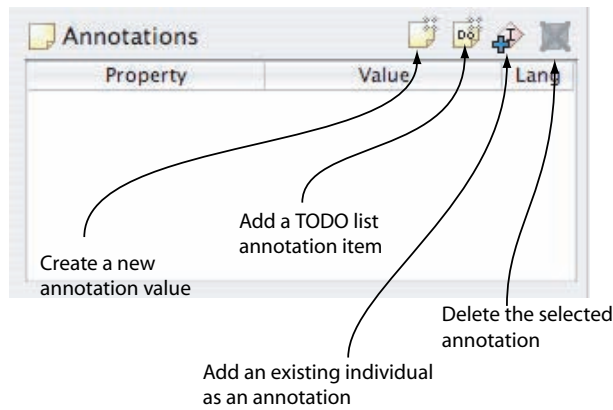
**Figure 6.7:** An annotations widget

## 6.5 Multiple Sets Of Necessary & Sufficient Conditions

In OWL it is possible to have multiple sets of necessary and sufficient conditions as depicted in Figure 6.8. In the '**Conditions Widget**', multiple sets of necessary and sufficient conditions are represented using multiple "NECESSARY & SUFFICIENT" headers with necessary and sufficient conditions listed under each header as shown in Figure 6.8. To create a *new* set of necessary and sufficient conditions, any "NECESSARY & SUFFICIENT" 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 "NECESSARY & SUFFICIENT" 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 "NECESSARY & SUFFICIENT" header) to add the condition to the existing set.
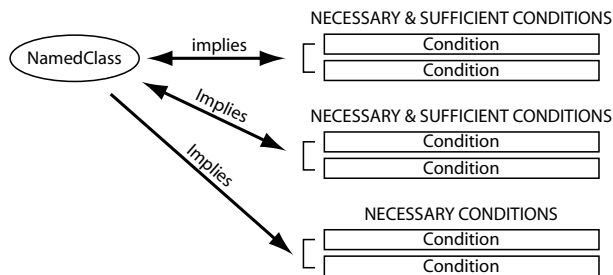


**Figure 6.8:** Necessary Conditions, and Multiple Sets of Necessary And Sufficient Conditions
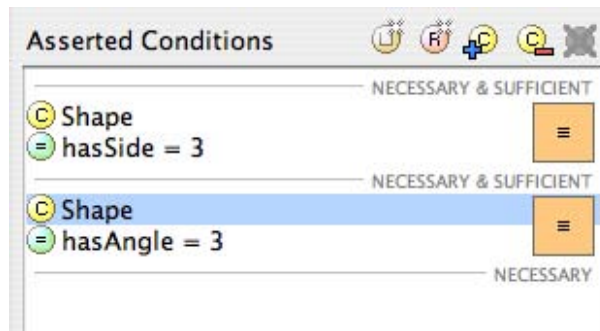
**Figure 6.9:** The Definition of a Triangle Using Multiple Necessary & Sufficient Conditions

**Exercise 54: Create a class to define a** Triangle **using multiple sets of Necessary & Sufficient conditions**

1. Create a subclass of owl: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 '**OWLClasses**' tab select the Triangle class. Select the "NECESSARY & SUFFICIENT" header in the '**Conditions Widget**'. Press the '**Create restriction button**' on the '**Conditions Widget**' to display the '**Create restriction dialog**'.

6. Select = cardinality as the type of restriction to be created. Select hasSide as the property to be restricted. In the filler edit box type 3. Press '**OK**' to close the dialog and create the restriction

7. Select the "NECESSARY & SUFFICIENT" header in the '**Conditions Widget**' again. Press the '**Create restriction**' button to display the '**Create restriction**' dialog.

8. Select = cardinality as the type of restriction to be created. Select hasAngle as the property to be restricted. In the filler edit box type 3. Press '**OK**' to close the dialog and create the restriction.

9. Drag Polygon from under the "NECESSARY" header and drop it onto the hasSide = 3 restriction.

10. Select the hasAngle = 3 restriction. Click the '**Add named class...**' button to display a dialog containing the class hierarchy. Select the Polygon class and click the '**OK**' button to close the dialog.

The '**Conditions Widget**' should now look like the picture shown in Figure 6.9.

# Chapter 7

# Other Topics

## 7.1 Language Profile

As explained in section 3.1 on page 11, there are three sub-languages of OWL: OWL-Lite, OWL-DL and OWL-Full. When editing an ontology Protégé-OWL offers the ability to constrain the constructs used in class expressions etc. so that the ontology being edited falls into either the OWL-DL or OWL-Full sub-langauges. The desired sub-language, or 'language profile' to be used can be set via the Protégé-OWL preferences dialog shown in Figure 4.41 on page 52. To choose between OWL-DL and OWL-Full, the 'OWL (supports one of the following OWL species)' should be ticked, and then either the 'OWL DL (optimized for reasoning)' or the 'OWL Full (supports the complete range of OWL elements)' should be selected.

## 7.2 Namespaces And Importing Ontologies

OWL ontologies are able to import other OWL ontologies rather like importing packages in java, or including files in C/C++. This section describes *namespaces*, which are a general naming mechanism and are usually used to facilitate ontology importing. It then describes how to import ontologies in general.

### 7.2.1 Namespaces

Every ontology has its own *namespace* — this is known as the *default namespace*. An ontology may also use other namespaces. A namespace is a string of characters that prefixes the class, property and individual identifiers in an ontology. By maintaining different namespaces for different ontologies it is possible for one ontology to reference classes, properties and individuals in another ontology in an unambiguous manner and without causing name clashes. For example, all OWL ontologies (including the Pizza ontology developed in this tutorial) reference the class owl:Thing. This class resides in the OWL vocabulary ontology that has the namespace http://www.w3.org/2002/07/owl#.

In order to ensure that namespaces are unique they manifest themselves as Unique Resource Identifiers

(URIs)[1] ending in either '/' or '#'. For example, the default namespace in Protégé-OWL (the namespace that is assigned to newly created ontologies in Protégé-OWL ) is http://a.com/ontology#. This means that all identifiers for classes, properties and individuals that are created in Protégé-OWL (by default) are prefixed with http://a.com/ontology#. For example, the full name for the class PizzaTopping is http://a.com/ontology#PizzaTopping. The full name for the class MargheritaPizza is http://a.com/ontology#MargheritaPizza. Fortunately, Protégé-OWL hides these namespace prefixes which means that we don't have to type in these long winded names every time we want to use a class, property or individual identifier.

Namespaces help to avoid name clashes when one ontology references classes, properties and individuals in another ontology. For example, suppose an ontology about aircraft, AircraftOntology has a class named Wing, which describes the wing of an aeroplane. An ontology about birds, BirdOntology also has a class named Wing, which describes the wing of a bird. The namespace for the AircraftOntology is http://www.ontologies.com/aircraft#. The namespace for the BirdOntology is http://www.birds.com/ontologies/BirdOntology#. Evidently, the Wing class in the AircraftOntology is not the same as the Wing class in the BirdOntology. Now suppose that the AircraftOntology imports the BirdOntology. Because of the namespace mechanism, the full name for the Wing class in the AircraftOntology is http://www.ontologies.com/aircraft#Wing. The full name for the Wing class in the BirdOntology is http://www.birds.com/ontologies/BirdOntology#Wing. Hence, when the AircraftOntology refers to classes in the BirdOntology no name clash will occur. Note that neither of the above namespace URIs necessarily have to be URLs i.e. they don't necessarily have to have a physical location (on the web) — URIs are used because they ensure Uniqueness.

In order to make referencing classes, properties and individuals more manageable when using multiple namespaces, *namespace prefixes* are used. A namespace prefix is a short string, usually a sequence of around two or three characters that represents a full namespace. For example, we could use "ac" to represent the above 'aircraft ontology' namespace http://www.ontologies.com/aircraft# and the prefix "bird" to represent the 'bird ontology' namespace http://www.birds.com/ontologies/BirdOntology#. When we now use identifiers such as class names, we prefix the identifier with the namespace prefix and a colon. For example ac:Wing or bird:Wing.

For a given ontology, the *default namespace* is the namespace for that ontology — in Protégé-OWL the *default namespace* corresponds to the namespace of the ontology that is being edited. When using identifiers that belong to the default namespace (the ontology being edited) a namespace prefix is not used — classes, properties and individuals are simply referenced using their 'local' name. However, for imported ontologies we must use a namespace prefix to refer to classes, properties and individuals in the imported ontology. For example, suppose we were editing the 'aircraft ontology', which has a namespace of http://www.ontologies.com/aircraft# and we wanted to refer to classes in the 'bird ontology' with the namespace of http://www.birds.com/ontologies/BirdOntology# and the namespace prefix of "bird". When we refer to classes without a namespace prefix, for example Wing, we are talking about classes in the aircraft ontology. When we refer to classes with a namespace prefix 'bird', for example bird:Wing, we are talking about classes in the bird ontology.

### 7.2.2  Creating And Editing Namespaces in Protégé-OWL

**Editing The Default Namespace**

The default namespace can be set using the '**Default Namespace**' widget, which is located in the top left corner of the '**Metadata**' tab and is shown in Figure 7.1. To change the default namespace simply type a new namespace into the edit box. The namespace must be a valid URI and must end in either '/' or '#'. Some examples of valid namespaces are listed below:

---

[1]Note that Unique Resource Locators (URLs), which identify physical locations of documents (e.g. web pages) are a special form of URI.
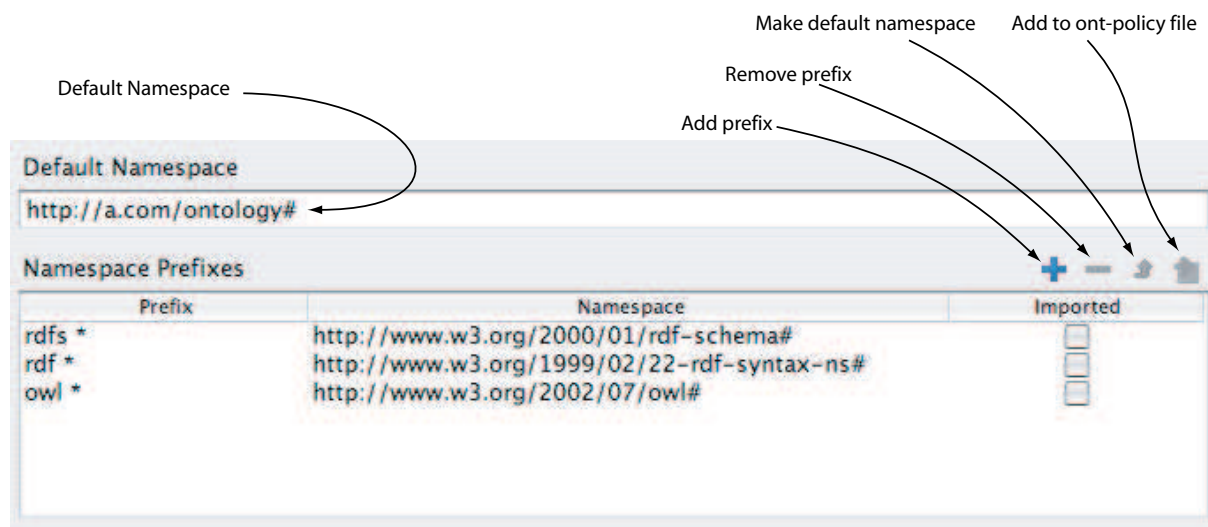
**Figure 7.1:** The Default Namespace and Namespaces Widget

- myNameSpace#

- universityOfManchester:ontologies:pizzas#

- http://www.cs.man.ac.uk/ontologies/pizzas/

**Creating Other Namespaces**

As well as specifying a default namespace for the ontology it is possible to setup other namespace prefix
- namespace mappings. This makes it possible to refer to classes, properties and individuals in other
ontologies.

To create/setup namespaces and their associated prefixes in Protégé-OWL the '**Namespace Prefixes**'
widget shown in Figure 7.1 is used. This widget contains three columns: '**Prefix**', '**Namespace**' and
'**Imported**' — we will deal with the '**Imported**' column later. When Protégé-OWL creates a new OWL
project it automatically creates/sets up three namespaces:

- **rdf** — http://www.w3.org/1999/02/22-rdf-syntax-ns# (The Resource Description Framework names-
  pace)

- **rdfs** — http://www.w3.org/2000/01/rdf-schema# (The RDF-Schema namespace)

- **owl** — http://www.w3.org/2002/07/owl# (The OWL vocabulary namespace)

Let's add a new namespace and prefix, which we can use in our ontology. For the purposes of this example
we will add a prefix and namespace for the wine ontology[2], which has the namespace http://www.w3.org/TR/2004/REC-

---

[2]The wine ontology is discussed and used as an example in the W3C OWL Guide. It contains information about various
types of wine and wineries.

owl-guide-20040210/wine#.

---

**Exercise 55:  Create a namespace and prefix to refer to classes, properties and individuals in the Wine ontology**

---

1. Press the '**Add prefix**' button on the '**Namespace prefix**' widget shown in Figure 7.1 to create a new namespace. A new namespace http://www.domain2.com# will be created with a prefix of p1.

2. Double click on the prefix p1 to edit it. Change it to vin, which is the namespace prefix used in the wine ontology.

3. Double click on the namespace http://www.domain2.com# to edit it. Change it to http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#. If the namespace is entered incorrectly i.e. if the namespace is not a valid URI and it does not end in '/' or '#' Protégé-OWL will reject the entry and revert to the previous value for the namespace.

4. We can now reference concepts in the wine ontology, and create classes and properties in the wine ontology namespace. For example create a new object property and name it vin:myWineProperty.

---

The property myWineProperty resides in the vin namespace http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine# (hence the prefixed name vin:myWineProperty). The full name of the property is therefore http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#myWineProperty.


## 7.2.3   Ontology Imports in OWL

OWL ontologies may import one or more other OWL ontologies. For example, suppose that the AircraftOntology imports the BirdOntology which contains descriptions of various birds (perhaps we want to simulate bird strikes on aircraft) — all of the classes, properties, individuals *and axioms* that are contained in the BirdOntology will be available to be used in the AircraftOntology. This makes it possible to use classes, properties and individuals from the BirdOntology in class descriptions in the AircraftOntology. It also makes it possible to *extend* the descriptions of classes, properties and individuals in the BirdOntology by creating the extention descriptions in the AircraftOntology. Notice the distinction between *referring* to classes, properties and individuals in an other ontology using namespaces, and completely *importing* an ontology. When an ontology imports another ontology, not only can classes, properties and individuals be referenced by the importing ontology, the axioms and facts that are contained in the ontology being imported are actually included in the importing ontology. It should be noted that OWL allows ontology imports to be cyclic so for example OntologyA may import OntologyB and OntologyB may import OntologyA.


## 7.2.4   Importing Ontologies in Protégé-OWL

Ontology imports are usually co-ordinated using namespaces. The ontology being imported has its namespace and also namespace prefix set up and is then imported. To import an ontology in Protégé-OWL we must first locate the ontology that we want to import and determine its URL. For the purposes

of this example we will import the *koala* ontology, which is a simple ontology created by Holger Knublauch (author of the Protégé-OWL plugin) that demonstrates the constructs of OWL. The koala ontology is located on the web at `http://protege.stanford.edu/plugins/owl/owl-library/koala.owl`.

Let's import the koala ontology — for the purposes of this tutorial the koala ontology may be imported into a new, empty OWL ontology.

### Exercise 56: Import the koala ontology into an ontology

1. Switch to the '**Metadata**' tab.

2. Press the '**Add prefix**' button located on the '**Namespace prefix**' widget. A new namespace and namespace prefix will be created.

3. Edit the namespace prefix, changing it to koala.

4. We now need to specify the namespace. When importing ontologies the namespace should be the *actual URL* that the ontology is located at, followed by '/' or '#'. Edit the namespace for the koala prefix, changing it to http://protege.stanford.edu/plugins/owl/owl-library/koala.owl#.

5. Now tick the '**Import tick box**' that lies on the same line as the koala prefix and namespace. You will be presented with a dialog stating that the changes will not take place until the file is saved and reloaded. Click the '**Yes**' button on the dialog. If you haven't already saved the project you will be presented with the save dialog box — give the project a name and save it.

After these steps have been performed Protégé-OWL will import the koala ontology and then save and reload the project. When the project is reloaded the '**OWLClasses**' tab will be displayed and will contain classes from the koala ontology – likewise, the properties tab will also display properties from the koala ontology. It should be noted that imported classes cannot be edited (they cannot have information retracted) or deleted – class descriptions can only have information added to them.

### Alternative Locations

When it is intended that an ontology should be imported, it is usual for the namespace URI to actually be a URL (i.e. a pointer to a physical location) that points to the location where the ontology may be found. In most situations this is usually a web address. By ticking the '**Imported**' tickbox Protégé-OWL will attempt to find the ontology at the location that is specified by the namespace URI (URL). This sounds great, but what if there is no internet connection available, or the ontology doesn't actually exist at the namespace URI? Fortunately, it is possible to specify an alternative URI (URL), which can point to a 'local' copy of the ontology — for example a URL that points to a location on the hard disk, or server on the local area network. Alternative locations are specified in the *ontology policy* file, which is located in the Protégé-OWL plugin folder. This file does not need to be edited by hand – it can be edited using the ontology policy dialog.
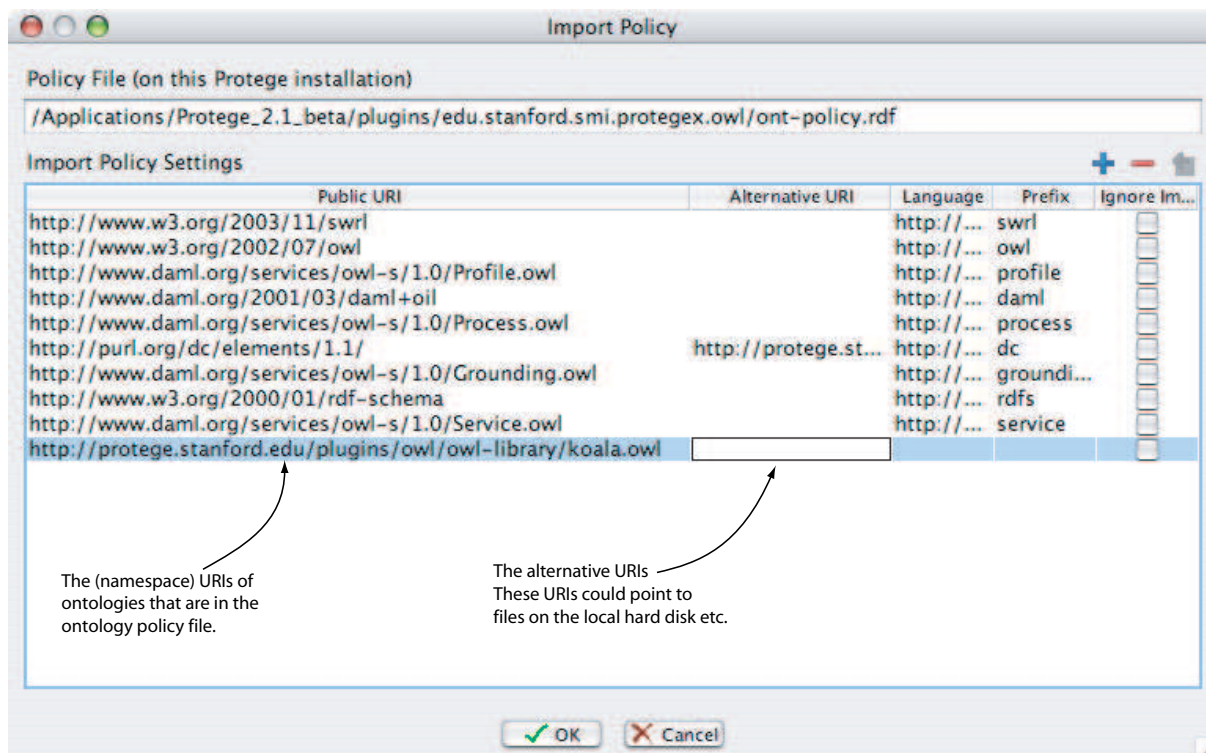
**Figure 7.2:** The Ontology Policy File Dialog

To specify an alternative location for an imported ontology follow these steps:

**Exercise 57: Specifing an alternative location for an imported ontology**

1. Select the ontology concerned in the '**Namespaces Prefixes**' widget.

2. Press the '**Add to ont-policy file**' button that is located on the '**Namespaces Prefixes**' widget shown in Figure 7.1. This will open the ontology policy file dialog shown in Figure 7.2.

3. As can be seen from 7.2 the koala ontology will have been added to the import policy (last line). To specify an alternative URI double click on the Alternative URI box on the koala ontology row – any valid URI may be entered. Press '**OK**' to close the dialog.

## 7.2.5 Importing The Dublin Core Ontology

The Dublin Core ontology is based on the *Dublin Core Meta Data Terms*. The Dublin Core Meta Data Terms were standardised/developed by The Dublin Core Meta Data Initiative[3]. They are a set of

---

[3]http://www.dublincore.org/

elements/terms that can be used to describe resources — in our case, we can use these terms to describe the 'resources' such as classes, properties and individuals in an ontology. The full set of Dublin Core Meta Data Terms is described at `http://www.dublincore.org/documents/dcmi-terms/`, the following list contains a few examples:

- **title** — Typically, a Title will be a name by which the resource is formally known.

- **creator** — Examples of a Creator include a person, an organisation, or a service. Typically, the name of a Creator should be used to indicate the entity.

- **subject** — Typically, a Subject will be expressed as keywords, key phrases or classification codes that describe a topic of the resource. Recommended best practice is to select a value from a controlled vocabulary or formal classification scheme.

- **description** — Description may include but is not limited to: an abstract, table of contents, reference to a graphical representation of content or a free-text account of the content.

- **contributor** — Examples of a Contributor include a person, an organisation, or a service. Typically, the name of a Contributor should be used to indicate the entity.

In order to annotate classes and other ontology entities with the above information and other Dublin Core Meta Data Terms the Dublin Core Meta Data ontology (DC Ontology) must be imported. Because Dublin Core Meta Data is so frequently used Protégé-OWL has an automated mechanism for importing the Dublin Core Meta Data ontology. The ontology can be imported in following manner:

**Exercise 58: Import the Dublin Core Meta Data Elements Ontology**

1. From the '**OWL Menu**' select '**Dublin Core metadata...**'.

2. A dialog box will appear. Tick the tickbox on the dialog to import the ontology.

3. A message will be displayed saying that Protégé-OWL needs to reload the ontology. Press the '**Yes**' button.

4. After a few seconds the Dublin Core Meta Data ontology will have been imported. Close the '**Dublin Core metadata**' dialog with the '**Close**' button.

5. Switch to the '**Properties**' tab. As shown in Figure 7.3, a number of annotation properties (representing the Dublin Core Meta Data Terms) will have been imported. These annotation properties may be used in the standard way (described in section 6.4).

## 7.2.6   The Protégé-OWL Meta Data Ontology

Several features used by the Protégé-OWL plugin (such as marking classes so that any primitive subclasses are automatically made disjoint) rely on the use of various Protégé-OWL annotation properties. These annotation properties are contained in the Protégé-OWL Meta Data Ontology, which is located in the

**Figure 7.3:** Imported Dublin Core Elements Available As Annotation Properties

Protégé-OWL plugin folder. In order for these annotation properties to be used, the Protégé-OWL Meta Data Ontology must be imported in the following manner:

---

**Exercise 59: Import the Protégé-OWL Meta Data Ontology**

1. Select the '**Preferences...**' item from the '**OWL Menu**' to display the preferences dialog shown in Figure 4.41 on page 52.

2. Tick the '**Import protege metadata ontology**' tickbox. You will be presented with a '**Confirm Reload**' dialog box asking you to reload the ontology for the changes to take effect. Press the '**Yes**' button.

---

## 7.3  Ontology Tests

Protégé-OWL provides a *test framework*, which contains various tests that may be run on the ontology being edited. These tests range from sanity tests such as checking that a property's characteristics correspond correctly with its inverse property's characteristics, to OWL-DL tests, which are designed to find constructs such as metaclasses that put an ontology into OWL-Full. The test framework is based upon a plugin architecture that enables new tests to be added by third party programmers – check the Protégé-OWL website for the availability of addon tests.

The various tests may be configured via the '**Test Settings**' dialog shown in Figure 7.4, which is accessible via the '**Test Settings...**' item on the '**OWL Menu**'. To run the tests the '**Run Ontology Tests...**' item should be selected from the '**OWL Menu**', or the '**Run Ontology Tests...**' button should be pressed on the OWL Toolbar.

After the ontology test have been run the results are displayed in a popup pane at the bottom of the screen as shown in Figure 7.5. The test results pane has the following columns:

- **Type** — The type of test result (a warning, and error etc.).

- **Source** — The source of the test result (e.g. a class or property). Double clicking on the source will automatically navigate to the source, by automatically selecting a class on the '**OWLClasses**' tab, or a property on the '**Properties**' tab for example.

- **Test Result** — A message that describes the result obtained.

In some cases Protégé-OWL is able to modify/correct aspects of the ontology that the tests have found to be at fault. In these cases, when the test is selected the small 'spanner' button on the left hand side of the test results pane will be enabled. Clicking this button will repair the ontology fault that gave rise to the test result.

## 7.4  TODO List

Protégé-OWL features a simple but useful *TODO List* mechanism. Classes, properties and the ontology itself can be annotated with TODO items. These can be attached to classes, properties and the ontology
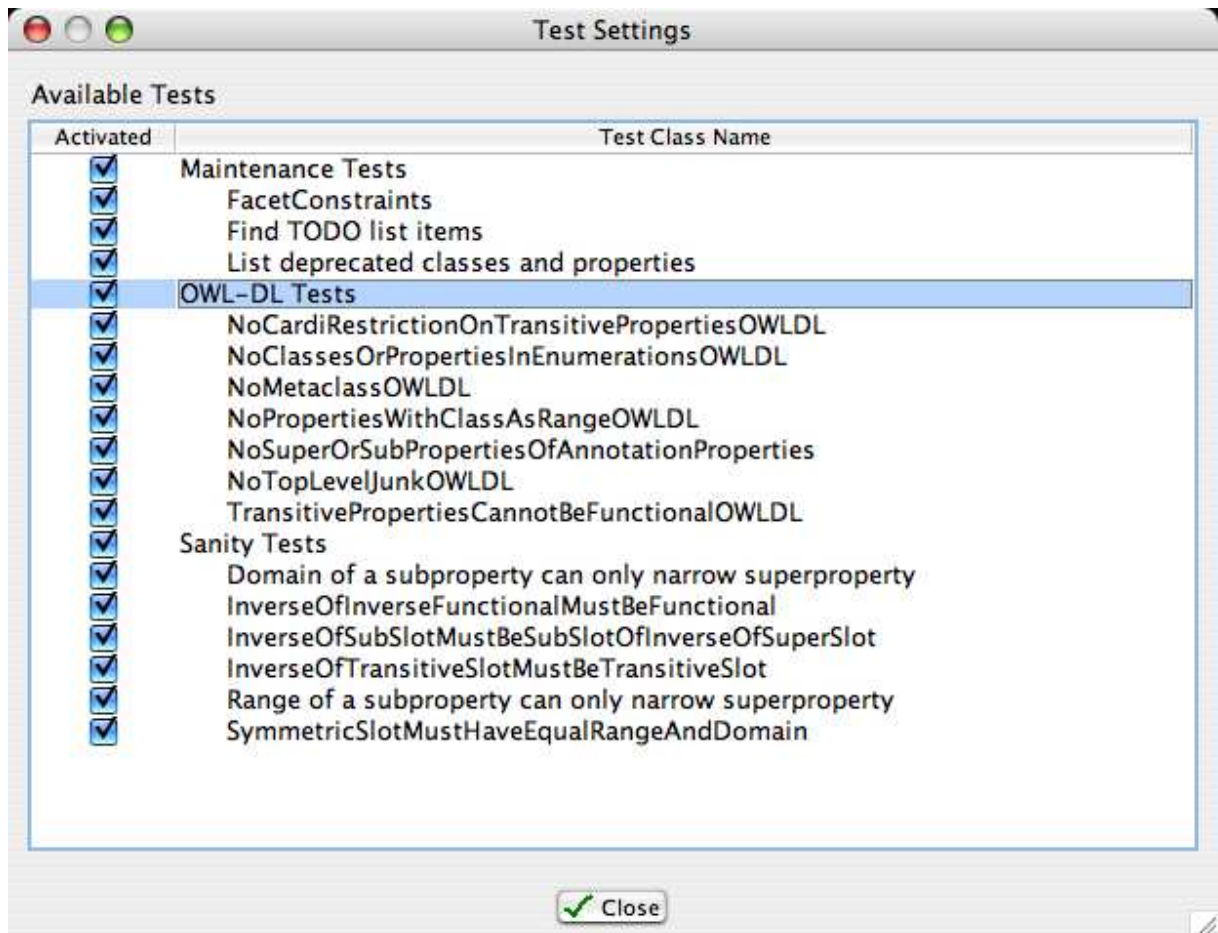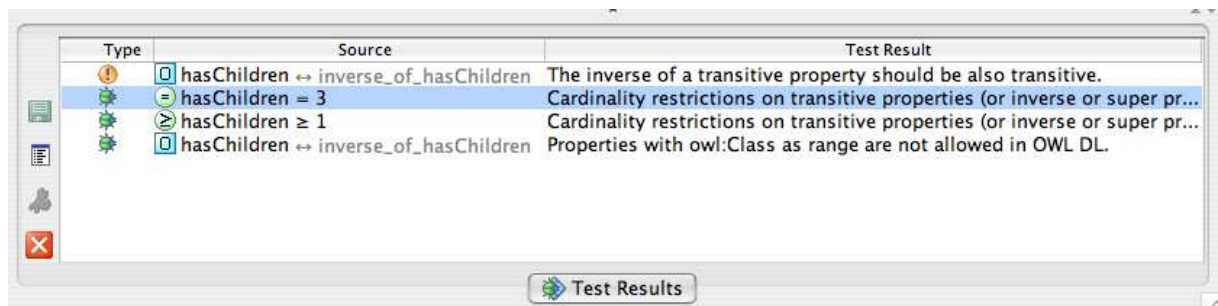
**Figure 7.4:** The Ontology Test Settings Dialog



**Figure 7.5:** Ontology TestResults

by using the '**Add TODO List Item**' button that is located on the '**Annotation Widgets**'. Pressing the '**Add TODO List Item**' creates a new annotation property that may be filled in with a textual description of the TODO task. To locate TODO items the '**Show TODO List...**' item should be selected from the '**OWL Menu**' or the '**Show TODO List...**' button pressed on the OWL Toolbar. This will display a list of TODO items in a popup pane at the bottom of the screen. Double clicking on each TODO item in this list will cause Protégé-OWL to automatically navigate to the TODO item in the ontology.

# Appendix A

# Restriction Types

This appendix contains further information about the types of property restrictions in OWL. It is intended for readers who aren't too familiar with the notions of logic that OWL is based upon.

All types of restrictions describe an unnamed set that could contain some individuals. This set can be thought of as an *anonymous class*. Any individuals that are members of this anonymous class satisfy the restriction that describes the class (Figure A.1). Restrictions describe the constraints on relationships that the individuals participate in for a given property.

When we describe a named class using restrictions, what we are effectively doing is describing anonymous superclasses of the named class.

## A.1   Quantifier Restrictions

Quantifier restrictions consist of three parts:

1. A quantifier, which is either the existential quantifier ($\exists$), or the universal quantifier ($\forall$).

2. A property, along which the restriction acts.

3. A filler that is a class description.

For a given individual, the quantifier effectively puts constraints on the relationships that the individual participates in. It does this by either specifying that *at least one* kind of relationship must exist, or by specifying the *only* kinds of relationships that can exist (if they exist).
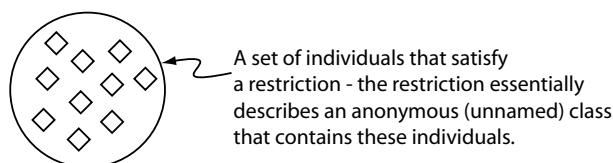


A set of individuals that satisfy a restriction - the restriction essentially describes an anonymous (unnamed) class that contains these individuals.

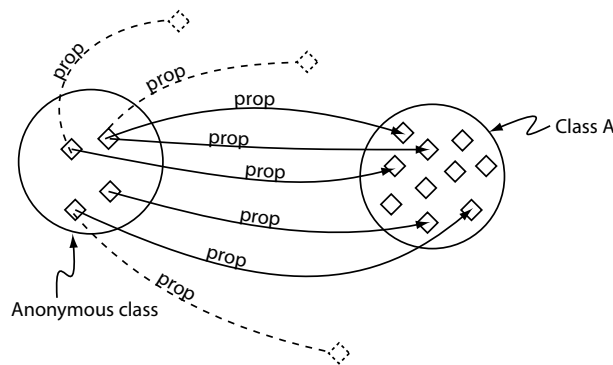**Figure A.1:** Restrictions Describe Anonymous Classes Of Individuals

**Figure A.2:** A Schematic Of An Existential Restriction (∃ prop ClassA)

## A.1.1 someValuesFrom – Existential Restrictions

Existential restrictions, also known as 'someValuesFrom' restrictions, or 'some' restrictions are denoted using ∃ – a backwards facing E. Existential restrictions describe the set of individuals that have *at least one* specific kind of relationship to individuals that are members of a specific class. Figure A.2 shows an abstracted schematic view of an existential restriction, ∃ prop ClassA – i.e. a restriction along the property prop with a filler of ClassA. Notice that all the individuals in the anonymous class that the restriction defines have *at least one* relationship along the property prop to an individual that is a member of the class ClassA. The dashed lines in Figure A.2 represent the fact that the individuals *may* have other prop relationships with other individuals that are *not* members of the class ClassA even though this has not been explicitly stated — The existential restriction does not constrain the prop relationship to members of the class ClassA, it just states that every individual must have *at least one* prop relationship with a member of ClassA — this is the *open world assumption* (OWA).

For a more concrete example, the existential restriction, ∃ hasTopping MozzarellaTopping, describes the set of individuals that take place in *at least one* hasTopping relationship with an other individual that is a member of the class MozzarellaTopping — in more natural English this restriction could be viewed as describing the things that 'have a Mozzarella topping'. The fact that we are using an existential restriction to describe the group of individuals that have *at least one* relationship along the hasTopping property with an individual that is a member of the class MozzarellaTopping does *not* mean that these individuals *only* have a relationship along the hasTopping property with an individual that is a member of the class MozzarellaTopping (there could be other hasTopping relationships that just haven't been explicity specified).

## A.1.2 allValuesFrom – Universal Restrictions

Universal restrictions are also known as 'allValuesFrom' restrictions, or 'All' restrictions since they *constrain* the filler for a given property to a *specific* class. Universal restrictions are given the symbol ∀ – i.e. an upside down A. Universal restrictions describe the set of individuals that, for a given property, *only* have relationships to other individuals that are members of a specific class. A feature of universal restrictions, is that for the given property, the set of individuals that the restriction describes will also contain the individuals that *do not have any* relationship along this property to any other individuals. A universal restriction along the property prop with a filler of ClassA is depicted in Figure A.3. Once again, an important point to note is that universal restrictions do not 'guarentee' the existence of a relationship for a given property. They merely state that if such a relationship for the given property exists, then it *must* be with an individual that is a member of a specified class.
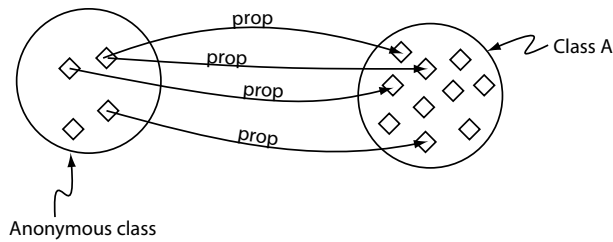
**Figure A.3:** A Schematic View Of The Universal Restriction, ∀ prop ClassA

Let's take a look at an example of a universal restriction. The restriction, ∀ hasTopping TomatoTopping describes the anonymous class of individuals that *only* have hasTopping relationships to individuals that are members of the class TomatoTopping, OR, individuals that definitely *do not* participate in any hasTopping relationships at all.

### A.1.3 Combining Existential And Universal Restrictions in Class Descriptions

A common 'pattern' is to combine existential and universal restrictions in class definitions for a given property. For example the following two restrictions might be used together, ∃ hasTopping MozzarellaTopping, and also, ∀ hasTopping MozzarellaTopping. This describes the set of individuals that have *at least* one hasTopping relationship to an individual from the class MozzarellaTopping, *and only* hasTopping relationships to individuals from the class MozzarellaTopping.

It is worth noting that is particularly unusual (and probably an error), if when describing a class, a universal restriction along a given property is used *without* using a 'corresponding' existential restriction along the same property. In the above example, if we had only used the universal restriction ∀ hasTopping Mozzarella, then we would have described the set of individuals that *only* participate in the hasTopping relationship with members of the class Mozzarella, *and also* those individuals that *do not participate in any* hasTopping relationships – probably a mistake.

## A.2 hasValue Restrictions

A hasValue restriction, denoted by the symbol ∋, describes an anonymous class of individuals that are related to another *specific individual* along a specified property. Contrast this with a quantifier restriction where the individuals that are described by the quantifier restriction are related to *any* indvidual from a *specified* class along a specified property. Figure A.4 shows a schematic view of the hasValue restriction prop ∋ abc. This restriction describes the anonymous class of individuals that have at least one relationship along the prop property to the specific individual abc. The dashed lines in Figure A.4 represent the fact that for a given individual the hasValue restriction does not constrain the property used in the restriction to a relationship with the individual used in the restriction i.e. there could be other relationships along the prop property. It should be noted that hasValue restrictions are semantically equivalent to an existential restriction along the same property as the hasValue restriction, which has a filler that is an enumerated class that contains the individual (and only the individual) used in the hasValue restriction.
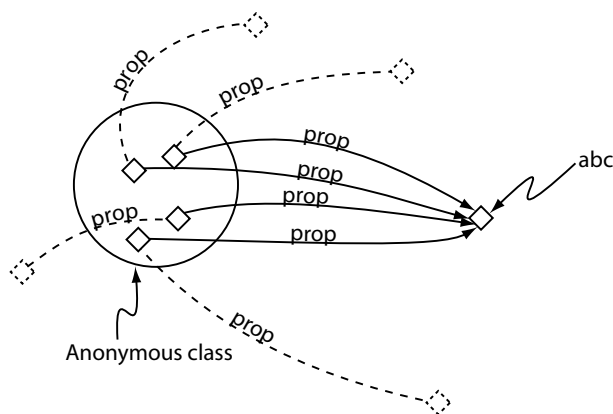
**Figure A.4:** A Schematic View Of The hasValue Restriction, prop ∋ abc — dashed lines indicate that this type of restriction does not constrain the property used in the hasValue restriction solely to the individual used in the hasValue restriction

# A.3 Cardinality Restrictions

Cardinality restrictions are used to talk about the number of relationships that an individual may participate in for a given property. Cardinality restrictions are conceptually easier to understand than quantifier restrictions, and come in three flavours: Minumum cardinality restrictions, Maximum cardinality restrictions, and Cardinality restrictions.

## A.3.1 Minimum Cardinality Restrictions

Minimum cardinality restrictions specify the *minimum* number of relationships that an individual must participate in for a given property. The symbol for a minimum cardinality restriction is the 'greater than or equal to' symbol ($\geq$). For example the minimum cardinality restriction, $\geq$ hasTopping 3, describes the individuals (an anonymous class containing the individuals) that participate in *at least* three hasTopping relationships. Minimum cardinality restrictions place no maximum limit on the number of relationships that an individual can participate in for a given property.

## A.3.2 Maximum Cardinality Restrictions

Maximum cardinality restrictions specify the *maximum* number of relationships that an individual can participate in for a given property. The symbol for maximum cardinality restrictions is the 'less than or equal to' symbol ($\leq$). For example the maximum cardinality restriction, $\leq$ hasTopping 2, describes the class of individuals that participate in at most two hasTopping relationships. Note that maximum cardinality restrictions place no minimum limit on the number of relationships that an individual must participate in for a specific property.

### A.3.3 Cardinality Restrictions

Cardinality restrictions specify the *exact* number of relationships that an individual must participate in for a given property. The symbol for a cardinality restrictions is the 'equals' symbol (=). For example, the cardinality restriction, = **hasTopping** 5, describes the set of individuals (the anonymous class of individuals) that participate in *exactly* five **hasTopping** relationships. Note that a cardinality restriction is really a syntactic short hand for using a combination of a minimum cardinality restriction and a maximum cardinality restriction. For example the above cardinality restriction could be represented by using the intersection of the two restrictions: $\leq$ **hasTopping** 5, and, $\geq$ **hasTopping** 5.

### A.3.4 The Unique Name Assumption And Cardinality Restrictions

OWL does *not* use the Unique Name Assumption (UNA)[1]. This means that different names *may* refer to the same individual, for example, the names "Matt" and "Matthew" may refer to the same individual (or they may not). Cardinality restrictions rely on 'counting' *distinct* individuals, therefore it is important to specify that either "Matt" and "Matthew" are the same individual, or that they are different individuals. Suppose that an individual "Nick" is related to the individuals "Matt", "Matthew" and "Matthew Horridge", via the **worksWith** property. Imagine that it has also been stated that the individual "Nick" is a member of the class of individuals that work with at the most two other individuals (people). Because OWL does not use the Unique Name Assumption, rather than being viewed as an error, it will be inferred that two of the names refer to the same individual[2].

---

[1] Confusingly, some reasoners (such as RACER) *do* use the Unique Name Assumstion!

[2] If "Matt", "Matthew" and "Matthew Horridge" have been asserted to be different individuals, then this will make the knowledge base inconsistent.

# Appendix B

# Complex Class Descriptions

An OWL class is specified in terms of its superclasses. These superclasses are typically named classes and restrictions that are in fact anonymous classes. Superclasses may also take the form of 'complex descriptions'. These complex descriptions can be built up using simpler class descriptions that are cemented together using logical operators. In particular:

- AND ($\sqcap$) — a class formed by using the AND operator is known as an *intersection* class. The class is the *intersection* of the individual classes.

- OR ($\sqcup$) — A class formed by using the OR operator is known as a *union* class. The class formed is the *union* of the individual classes.

## B.1   Intersection Classes ($\sqcap$)

An intersection class is described by combining two or more classes using the AND operator ($\sqcap$). For example, consider the intersection of **Human** and **Male** — depicted in Figure B.1. This describes an *anonymous* class that contains the individuals that are members of the class **Human** and the class **Male**. The semantics of an intersection class mean that the anonymous class that is described is a subclass of **Human** and a subclass of **Male**.

The anonymous intersection class above can be used in another class description. For example, suppose we wanted to build a description of the class **Man**. We might specify that **Man** is a subclass of the anonymous class described by the intersection of **Human** and **Male**. In other words, **Man** is a subclass of **Human** and **Male**.

## B.2   Union Classes ($\sqcup$)

A union class is created by combining two or more classes using the OR operator ($\sqcup$). For example, consider the union of **Man** and '**Woman**' — depicted in Figure B.2. This describes an anonymous class that contains the individuals that belong to either the class **Man** or the class **Woman** (or both).
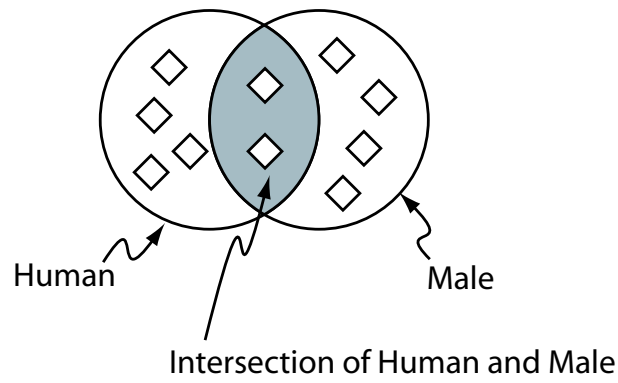
**Figure B.1:** The intersection of Human and Male (Human ⊓ Male) — The shaded area represents the intersection



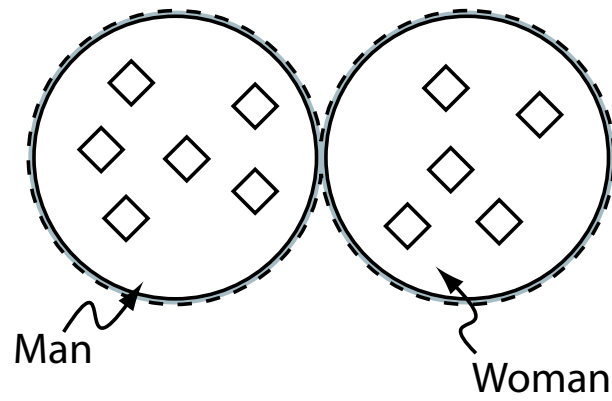**Figure B.2:** The union of Man and Woman (Man ⊔ Woman) — The shaded area represents the union

The anonymous class that is described can be used in another class description. For example, the class Person might be equivalent of the union of Man and Woman.