A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools
Edition 1.2

Matthew Horridge
Contributors
v 1.0 - Holger Knublauch , Alan Rector , Robert Stevens , Chris Wroe
v 1.1 - Simon Jupp, Georgina Moulton, Robert Stevens
v 1.2 - Nick Drummond, Simon Jupp, Georgina Moulton, Robert Stevens

The University Of Manchester

March 13, 2009

Chapter 1

Introduction

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

1.1 Conventions

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

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

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

Exercises and required tutorial steps are presented like this:

Exercise 1: Accomplish this

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

TIP

Tips and suggestions related to using Protégé 4 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 3

What are OWL Ontologies?

Ontologies are used to capture knowledge about some domain of interest. An ontology describes the concepts in the domain and also the relationships that hold between those concepts. Different ontology languages provide different facilities. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C)[1]. Like Protégé , OWL makes it possible to describe concepts but it also provides new facilities. It has a richer set of operators - e.g. intersection, union and negation. It is based on a different logical model which makes it possible for concepts to be defined as well as described. Complex concepts can therefore be built up in definitions out of simpler concepts. Furthermore, the logical model allows the use of a reasoner which can check whether or not all of the statements and definitions in the ontology are mutually consistent and can also recognise which concepts fit under which definitions. The reasoner can therefore help to maintain the hierarchy correctly. This is particularly useful when dealing with cases where classes can have more than one parent.

3.1 Components of OWL Ontologies

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

3.1.1 Individuals

Individuals, represent objects in the domain in which we are interested [2]. 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.

[1]`http://www.w3.org/TR/owl-guide/`

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

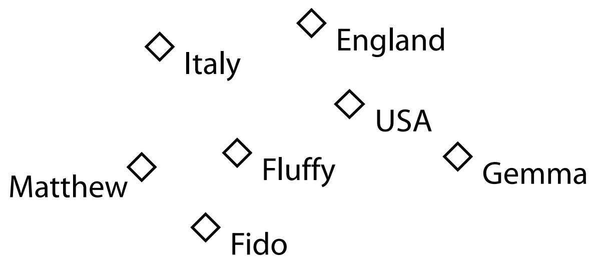


Figure 3.1: Representation Of Individuals

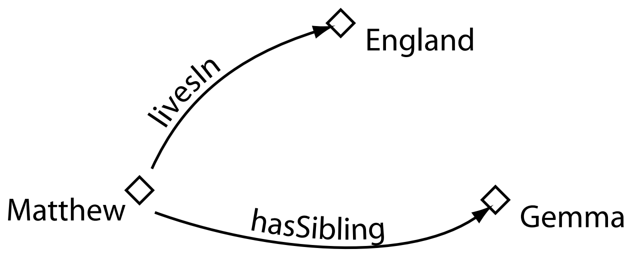


Figure 3.2: Representation Of Properties

Vocabulary

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

3.1.2 Properties

Properties are *binary* relations[3] on *individuals* - i.e. properties link *two* individuals together[4]. For example, the property hasSibling might link the individual Matthew to the individual Gemma, or the property hasChild might link the individual Peter to the individual Matthew. Properties can have inverses. For example, the inverse of hasOwner is isOwnedBy. Properties can be limited to having a single value – i.e. to being *functional*. They can also be either *transitive* or *symmetric*. These 'property characteristics' are explained in detail in Section 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*.

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

[4]Strictly speaking we should speak of 'instances of properties' linking individuals, but for the sake of brevity we will keep it simple.

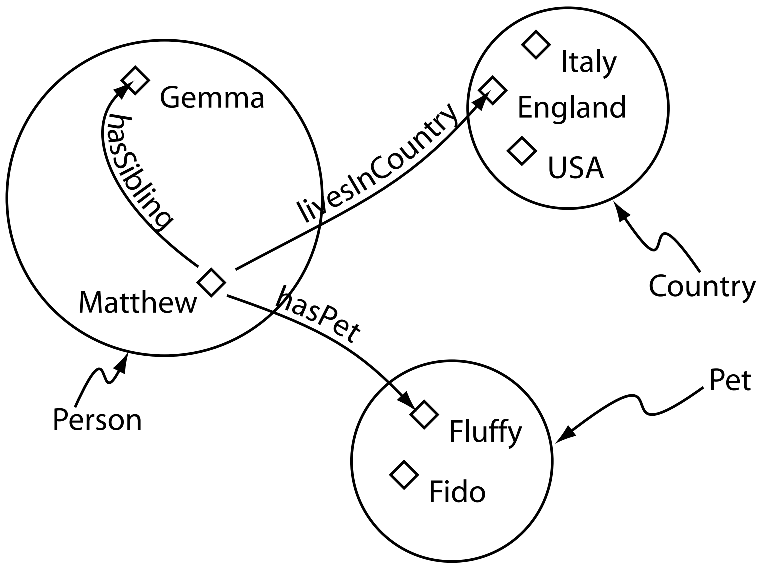


Figure 3.3: Representation Of Classes (Containing Individuals)

3.1.3 Classes

OWL classes are interpreted as *sets* that contain individuals. They are *described* using formal (mathematical) descriptions that state precisely the requirements for membership of the class. For example, the class Cat would contain all the individuals that are cats in our domain of interest.[5] 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

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

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

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

Chapter 4

Building An OWL Ontology

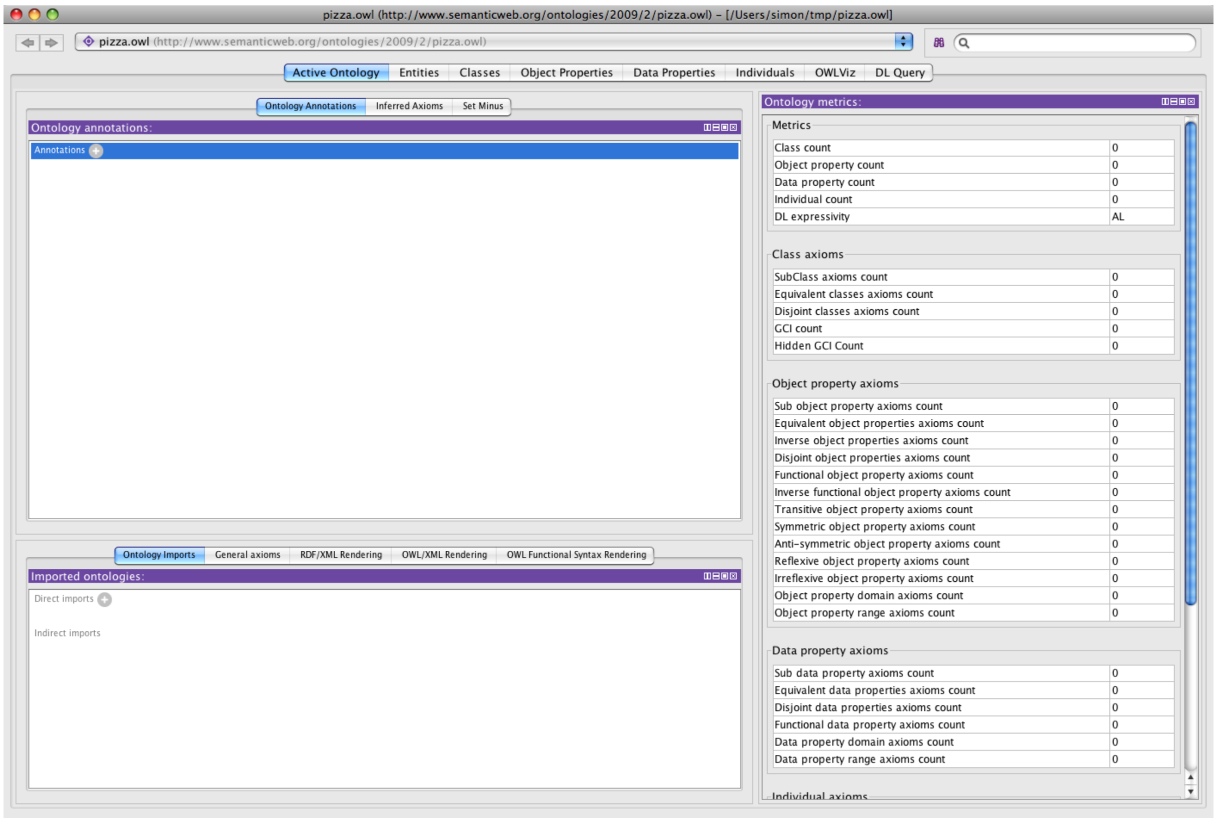
This chapter describes how to create an ontology of Pizzas. We use Pizzas because we have found them to provide many useful examples.[1]

Exercise 2: Create a new OWL Ontology

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

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

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



Metrics

Class count	0
Object property count	0
Data property count	0
Individual count	0
DL expressivity	AL

Class axioms

SubClass axioms count	0
Equivalent classes axioms count	0
Disjoint classes axioms count	0
GCI count	0
Hidden GCI Count	0

Object property axioms

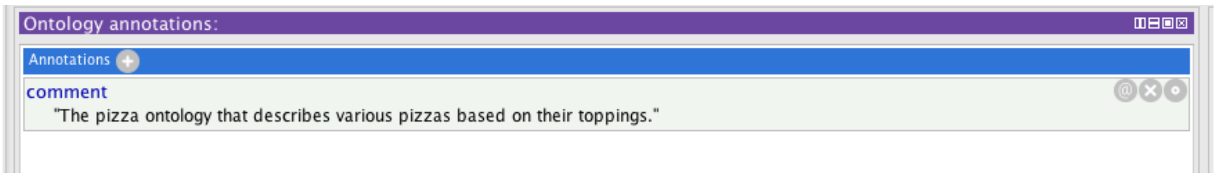
Sub object property axioms count	0
Equivalent object properties axioms count	0
Inverse object properties axioms count	0
Disjoint object properties axioms count	0
Functional object property axioms count	0
Inverse functional object property axioms count	0
Transitive object property axioms count	0
Symmetric object property axioms count	0
Anti-symmetric object property axioms count	0
Reflexive object property axioms count	0
Irreflexive object property axioms count	0
Object property domain axioms count	0
Object property range axioms count	0

Data property axioms

Sub data property axioms count	0
Equivalent data properties axioms count	0
Disjoint data properties axioms count	0
Functional data property axioms count	0
Data property domain axioms count	0
Data property range axioms count	0

Individual axioms

Figure 4.1: The Active Ontology Tab



comment
"The pizza ontology that describes various pizzas based on their toppings."

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

Exercise 3: Add a comment to the ontology

1. Ensure that the '**Active Ontology Tab**' is selected.
2. In the '**Ontology Annotations**' view, click the add icon (+) next to Annotations. An editing window will appear in the table. Select 'comment' from the list of built in annotation URIs and type your comment in the text box in the right hand pane.
3. Enter a comment such as `A pizza ontology that describes various pizzas based on their toppings.` and press OK to assign the comment. The annotations view on the '**Active Ontology Tab**' should look like the picture shown in Figure 4.2

4.1 Named Classes

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

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

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

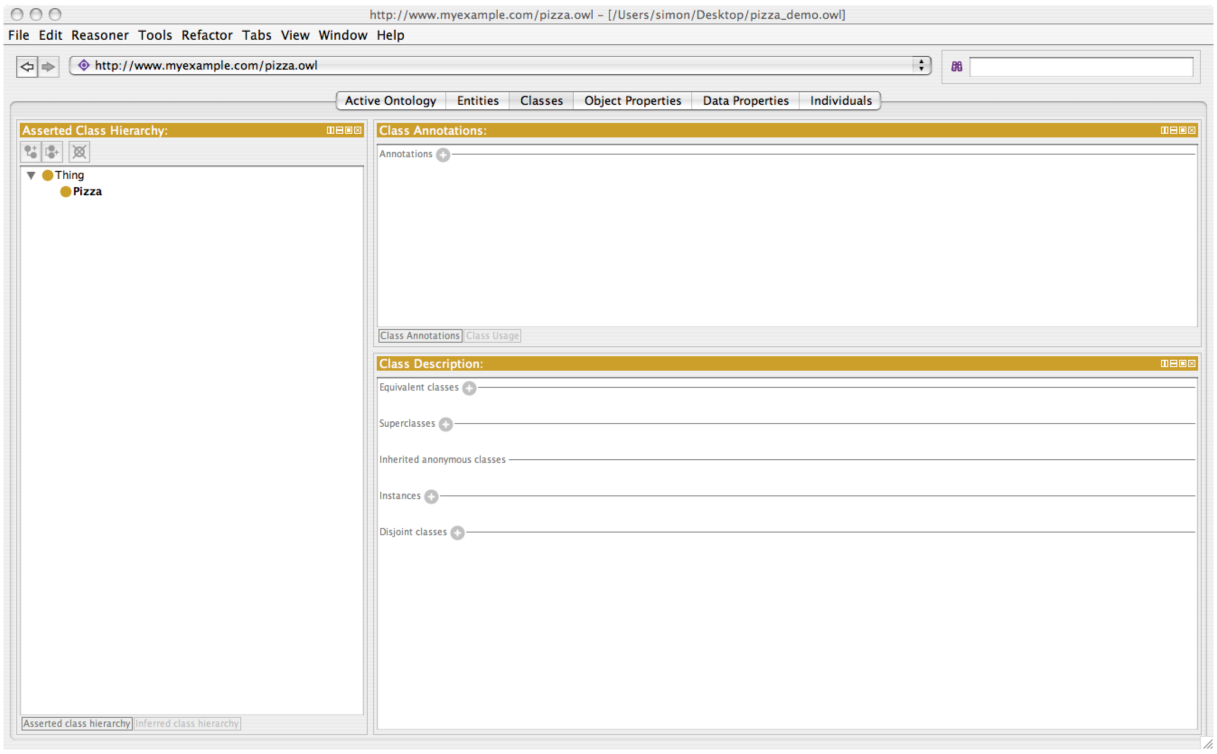


Figure 4.3: The Classes Tab

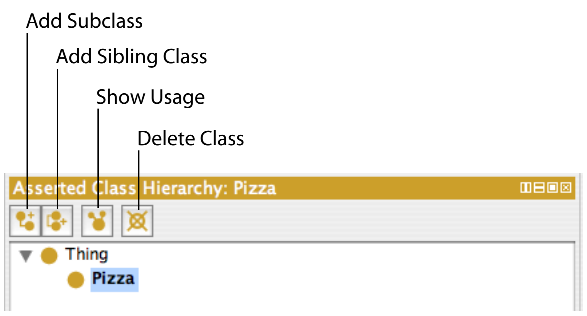


Figure 4.4: The Class Hierarchy Pane

Exercise 4: Create classes Pizza, PizzaTopping and PizzaBase

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

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

TIP

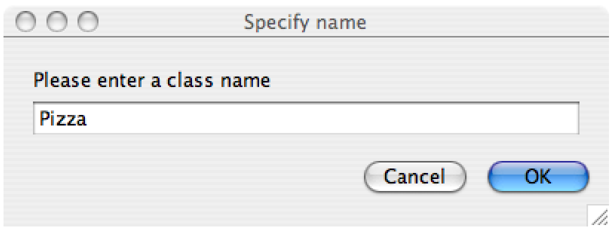
After creating Pizza, instead of re-selecting Thing and using the '**Create subclass**' button to create PizzaTopping and PizzaBase as further subclasses of Thing, the '**Add sibling class**' button (shown in Figure 4.4) can be used. While Pizza is selected, use the '**Create sibling class**' button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

Vocabulary

A class hierarchy may also be called a taxonomy.

TIP

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

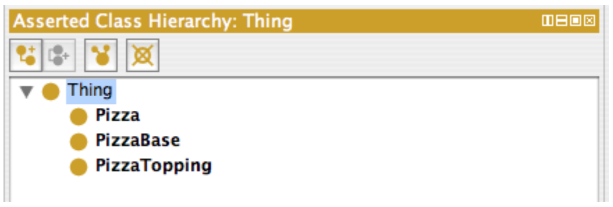


Specify name

Please enter a class name

Pizza

Figure 4.5: Class Name Dialog



Asserted Class Hierarchy: Thing

- Thing
  - Pizza
  - PizzaBase
  - PizzaTopping

Figure 4.6: The Initial Class Hierarchy

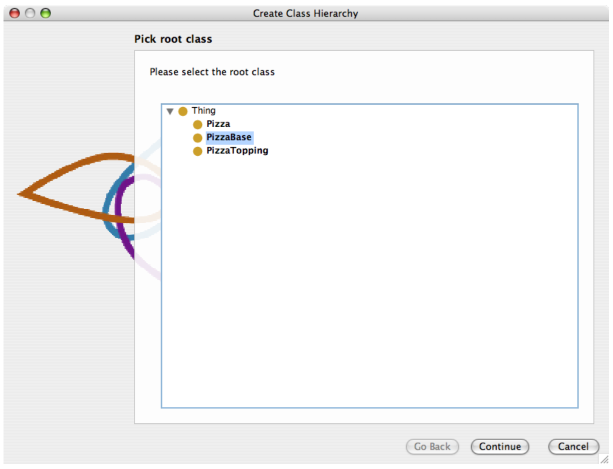
4.2 Disjoint Classes

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

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

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

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



Pick root class

Please select the root class

- Thing
  - Pizza
  - PizzaBase
  - PizzaTopping

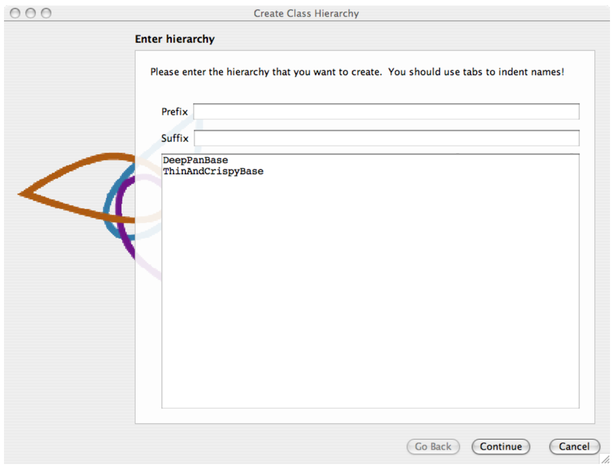
Figure 4.7: Create Class Hierarchy: Select class page

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 Create Class Hierarchy To Create Classes

In this section we will use the '**Create Class Hierarchy**' tool to add some subclasses of the class PizzaBase.



Please enter the hierarchy that you want to create. You should use tabs to indent names!

Prefix

Suffix

```
DeepPanBase
ThinAndCrispyBase
```

Figure 4.8: Create Class Hierarchy: Enter classes page

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

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

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

TIP

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

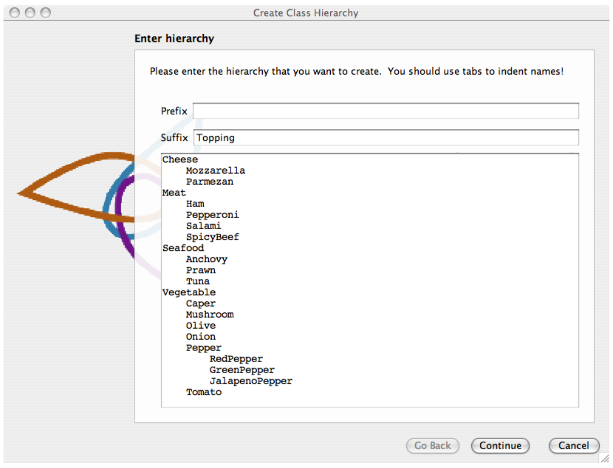
Creating Some Pizza Toppings

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

Exercise 7: Create some subclasses of PizzaTopping

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

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


Create Class Hierarchy

Enter hierarchy

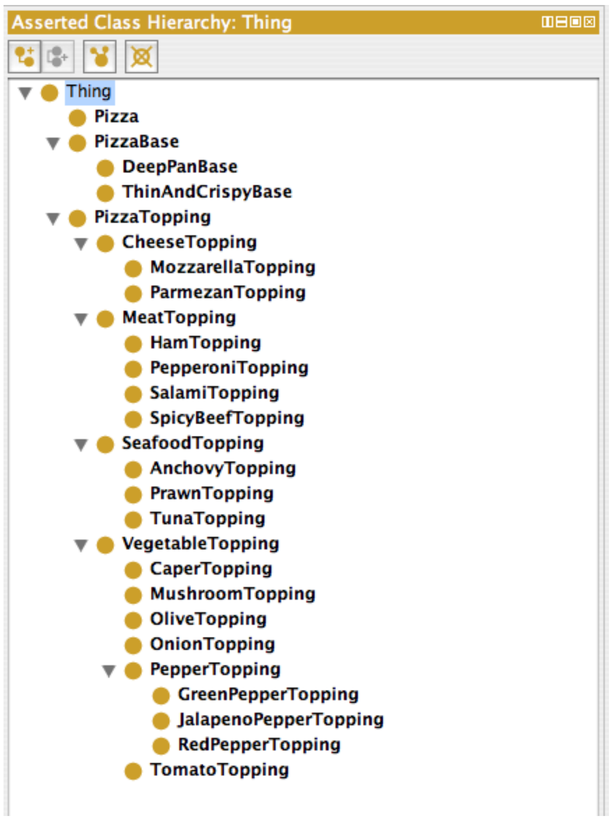
Please enter the hierarchy that you want to create. You should use tabs to indent names!

Prefix

Suffix Topping

```
Cheese
    Mozzarella
    Parmezan
Meat
    Ham
    Pepperoni
    Salami
    SpicyBeef
Seafood
    Anchovy
    Prawn
    Tuna
Vegetable
    Caper
    Mushroom
    Olive
    Onion
    Pepper
        RedPepper
        GreenPepper
        JalapenoPepper
    Tomato
```

Figure 4.9: Topping Hierarchy


Asserted Class Hierarchy: Thing

```
Thing
    Pizza
    PizzaBase
        DeepPanBase
        ThinAndCrispyBase
    PizzaTopping
        CheeseTopping
            MozzarellaTopping
            ParmezanTopping
        MeatTopping
            HamTopping
            PepperoniTopping
            SalamiTopping
            SpicyBeefTopping
        SeafoodTopping
            AnchovyTopping
            PrawnTopping
            TunaTopping
        VegetableTopping
            CaperTopping
            MushroomTopping
            OliveTopping
            OnionTopping
            PepperTopping
                GreenPepperTopping
                JalapenoPepperTopping
                RedPepperTopping
            TomatoTopping
```

Figure 4.10: Class Hierarchy

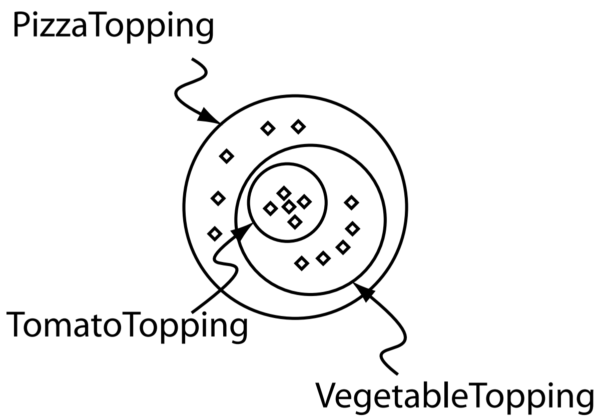


Figure 4.11: The Meaning Of Subclass — *All* individuals that are members of the class TomatoTopping are members of the class VegetableTopping and members of the class PizzaTopping as we have stated that TomatoTopping is a subclass of VegetableTopping which is a subclass of PizzaTopping

MEANING

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

[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.

4.4 OWL Properties

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

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

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

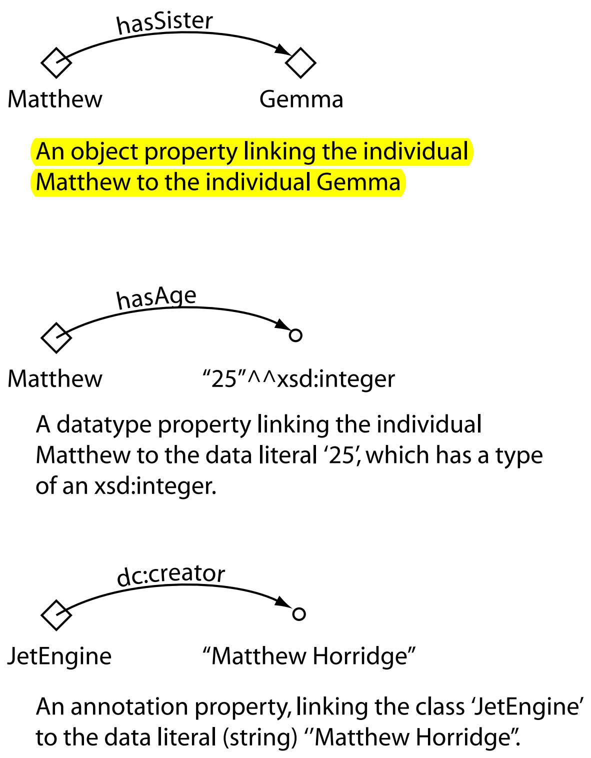

hasSister
Matthew
Gemma
An object property linking the individual Matthew to the individual Gemma
hasAge
Matthew
"25"^^xsd:integer
A datatype property linking the individual Matthew to the data literal '25', which has a type of an xsd:integer.
dc:creator
JetEngine
"Matthew Horridge"
An annotation property, linking the class 'JetEngine' to the data literal (string) "Matthew Horridge".

Figure 4.12: The Different types of OWL Properties

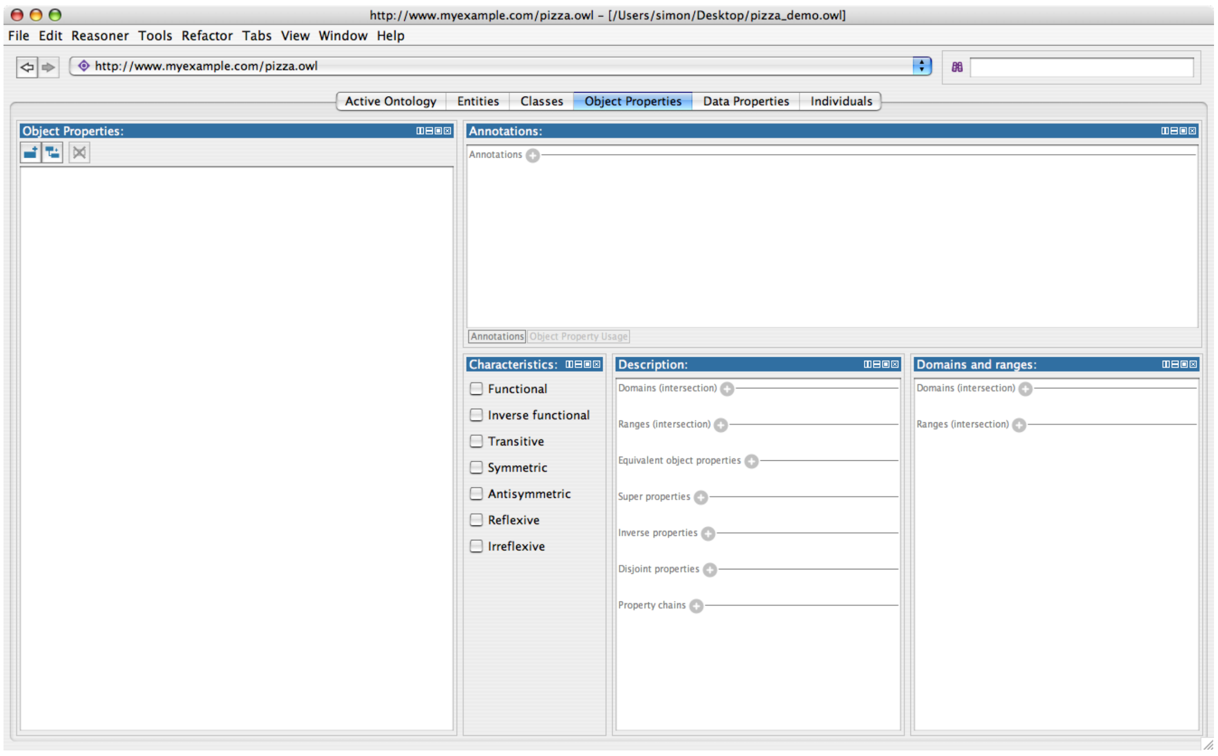


Figure 4.13: The PropertiesTab

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

Exercise 8: Create an object property called hasIngredient

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

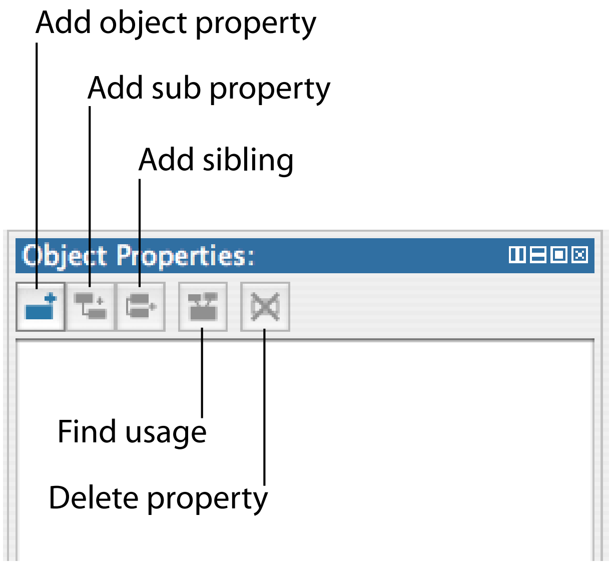


Figure 4.14: Property Creation Buttons — located on the Properties Tab above the property list/tree

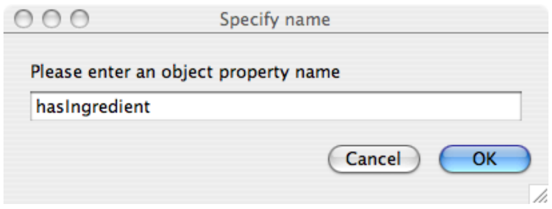


Figure 4.15: Property Name Dialog

TIP

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

[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

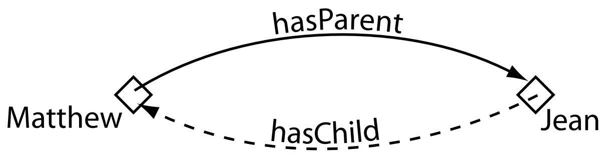


Figure 4.16: An Example Of An Inverse Property: hasParent has an inverse property that is hasChild

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

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

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

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

4.5 Inverse Properties

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

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

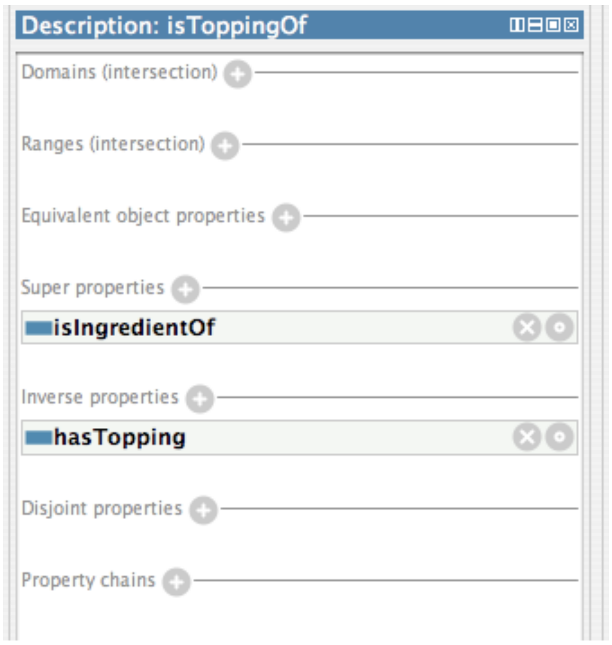


Figure 4.17: The Inv erse Property View

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

Exercise 10: Create some inverse properties

1. Use the '**Add object property**' button on the '**Object Properties**' tab to create a new Object property called isIngredientOf (this will become the inverse property of hasIngredient).
2. Press the add icon (+) next to '**Inverse properties**' button on the '**Property Description**' view shown in Figure 4.17. This will display a dialog from which properties may be selected. Select the hasIngredient property and press '**OK**'. The property hasIngredient should now be displayed in the '**Inverse Property**' view.
3. Select the hasBase property.
4. Press add icon (+) next to '**Inverse properties**' on the '**Property Description**' view. Create a new property in this dialog called isBaseOf. Select this property and click '**OK**'. Notice that hasBase now has a inverse property assigned called isBaseOf. You can optionally place the new isBaseOf property as a sub-property of isIngredientOf (N.B This will get inferred later anyway when you use the reasoner).
5. Select the hasTopping property.
6. Press add icon (+) next to '**Inverse properties**' on the '**Property Description**' view. Use the property dialog that pops up to create the property isToppingOf and press '**OK**'.

4.6 OWL Object Property Characteristics

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

4.6.1 Functional Properties

If a property is functional, for a given individual, there can be at most one individual that is related to the individual via the property. Figure 4.18 shows an example of a functional property hasBirthMother — something can only have *one* birth mother. If we say that the individual Jean hasBirthMother Peggy and we also say that the individual Jean hasBirthMother Margaret[4], then because hasBirthMother is a functional property, we can infer that Peggy and Margaret must be the same individual. It should be noted however, that if Peggy and Margaret were explicitly stated to be two different individuals then the above statements would lead to an inconsistency.

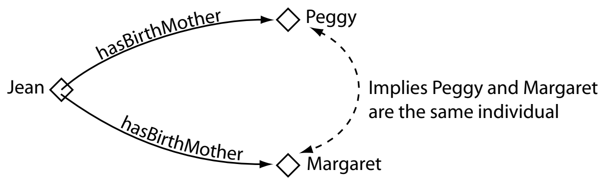


Figure 4.18: An Example Of A Functional Property: hasBirthMother

Vocabulary

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 4.19 shows an example of an inverse functional property isBirthMotherOf. This is the inverse property of hasBirthMother — since hasBirthMother is functional, isBirthMotherOf is inverse functional. If we state that Peggy is the birth mother of Jean, and we also state that Margaret is the birth mother of Jean, then we can infer that Peggy and Margaret are the same individual.

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 4.20 shows an example of the transitive property hasAncestor. If the individual Matthew has an ancestor that is Peter, and Peter has an ancestor that is William, then we can infer that Matthew has an ancestor that is William – this is indicated by the dashed line in Figure 4.20.

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

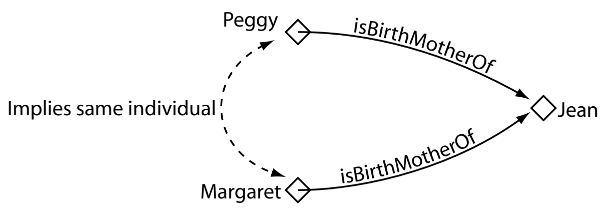


Figure 4.19: An Example Of An Inverse Functional Property: isBirthMotherOf

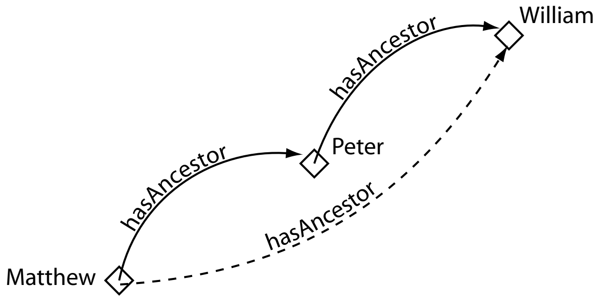


Figure 4.20: An Example Of A Transitive Property: hasAncestor

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

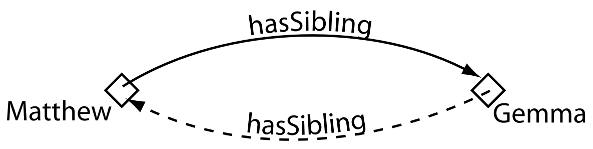
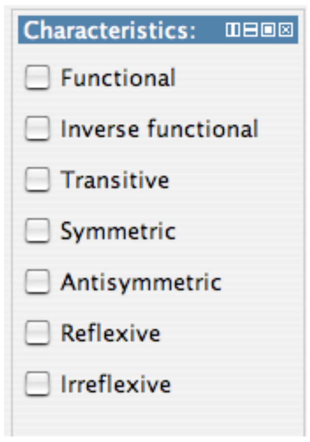


Figure 4.21: An Example Of A Symmetric Property: hasSibling

We want to make the hasIngredient property transitive, so that for example if a pizza topping has an ingredient, then the pizza itself also has that ingredient. To set the property characteristics of a property the property characteristics view shown in Figure 4.22 which is located in the lower right hand corner of


Characteristics:

- Functional
- Inverse functional
- Transitive
- Symmetric
- Antisymmetric
- Reflexive
- Irreflexive

Figure 4.22: Property Characteristics Views

the properties tab is used.

Exercise 11: Make the hasIngredient property transitive

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

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 4 . However, the reasoner *will* assume that if a property is transitive, its inverse property is also a transitive.

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

[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 our pizza can only have one base. There are numerous ways that this could be accomplished. However, to do this we will make the hasBase property *functional*, so that it may have

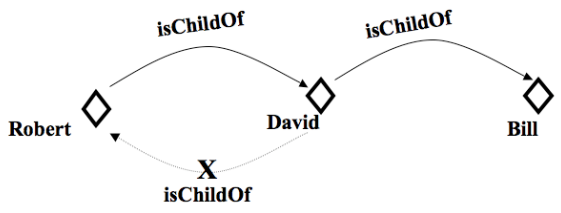


Figure 4.23: An example of the antisymmetric property hasChildOf

only one value for a given individual.

Exercise 12: Make the hasBase property functional

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

NOTE

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

4.6.5 Antisymmetric properties

If a property P is antisymmetric, and the property relates individual a to individual b then individual b cannot be related to individual a via property P. Figure 4.23 shows an example of a antisymmetric property. If the individual Robert is related to the individual David via the isChildOf property, then it can be inferred that David is not related to Robert via the isChildOf property. It is, however, reasonable to state that David could be related to another individual Bill via the isChildOf property. In other words, if Robert is a child of David, then David cannot be a child of Robert, but David can be a child of Bill.

4.6.6 Reflexive properties

A property P is said to be reflexive when the property must relate individual a to itself. In Figure 4.24 we can see an example of this: using the property knows, an individual George must have a relationship to itself using the property knows. In other words, George must know herself. However, in addition, it is possible for George to know other people; therefore the individual George can have a relationship with individual Simon along the property knows.

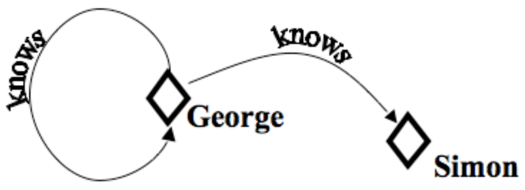


Figure 4.24: An example of a Reflexive Property: knows

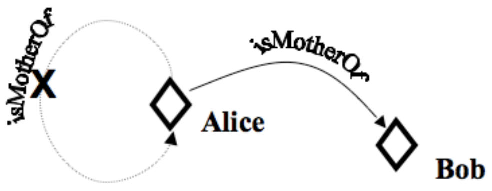


Figure 4.25: An example of a Irreflexive Property: isMotherOf

4.6.7 Irreflexive properties

If a property P is *irreflexive*, it can be described as a property that relates an individual a to individualb, where individual a and individualb are not the same. An example of this would be the property motherOf: an individual Alice can be related to individual Bob along the property motherOf, but Alice cannot be motherOf herself (Figure 4.25).

4.7 Property Domains and Ranges

Properties may have a *domain* and a *range* specified. Properties link individuals from the *domain* to individuals from the *range*. For example, in our pizza ontology, the property hasTopping would probably link individuals belonging to the class Pizza to individuals belonging to the class of PizzaTopping. In this case the *domain* of the hasTopping property is Pizza and the *range* is PizzaTopping — this is depicted in Figure 4.26.