

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.

¹The 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/#>

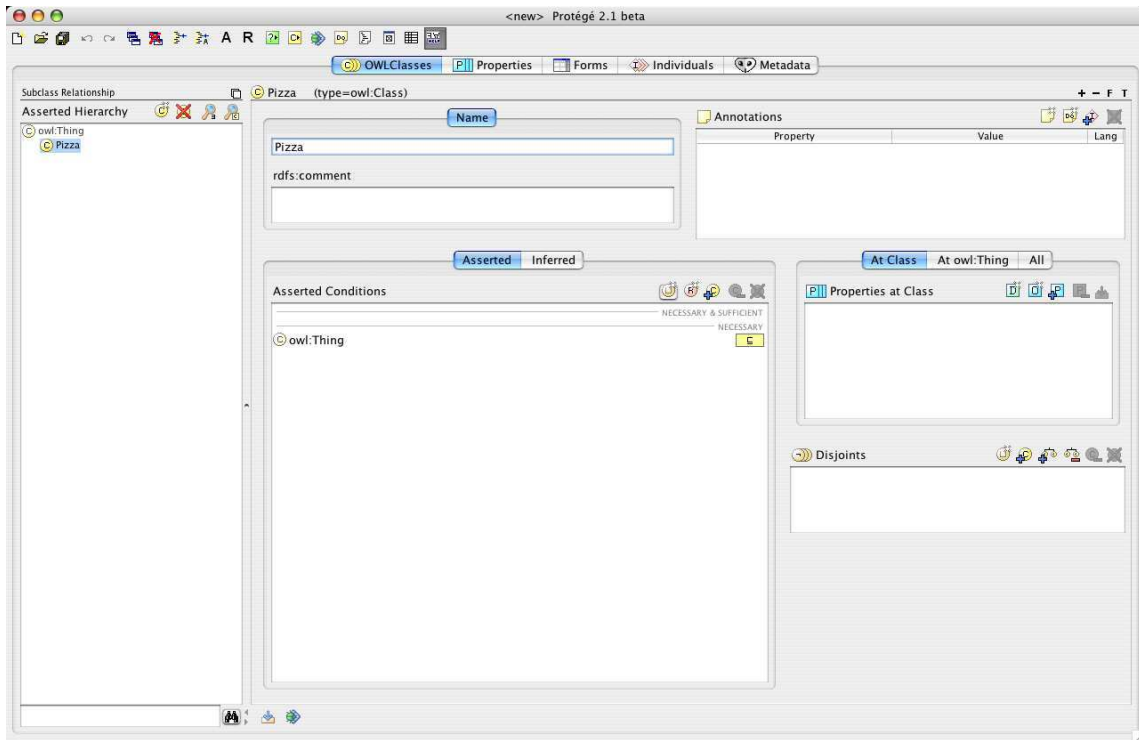


Figure 4.1: The Classes Tab

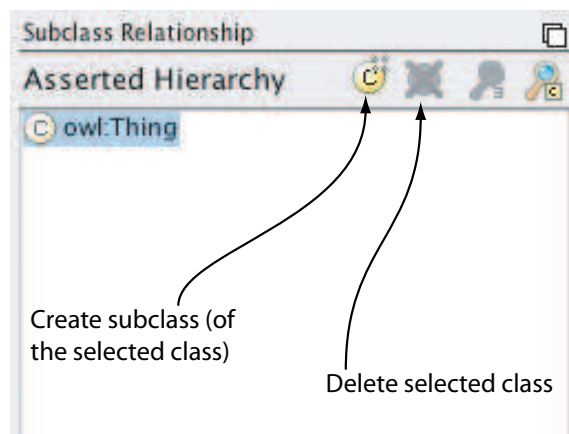


Figure 4.2: The Class Hierarchy Pane

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.

Vocabulary



A class hierarchy may also be called a taxonomy.

TIP

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

4.2 Disjoint Classes

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

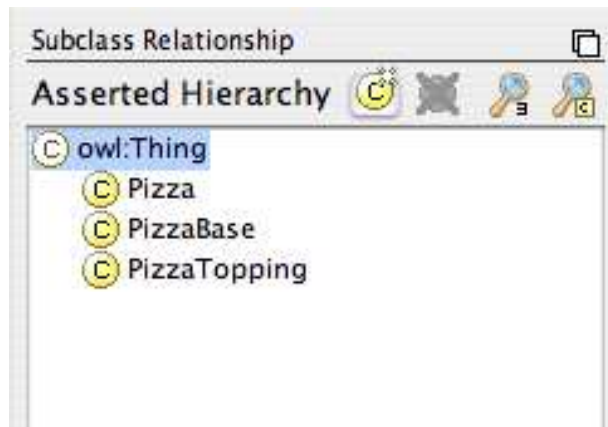


Figure 4.4: The Initial Class Hierarchy

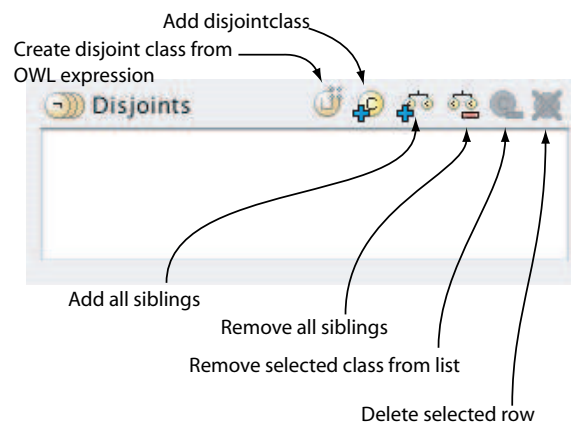


Figure 4.5: The Disjoint Classes Widget

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 **PizzaBase** 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**.

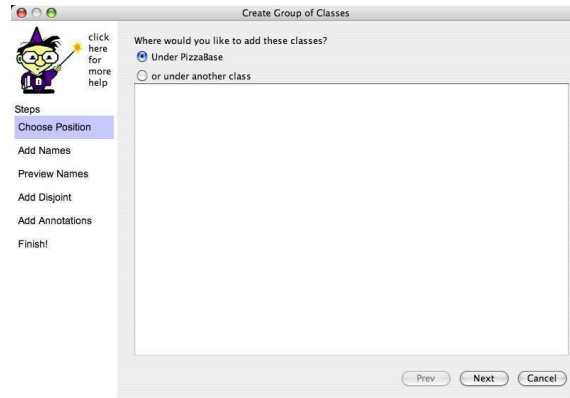


Figure 4.6: Add Group Of Classes Wizard: Select class page

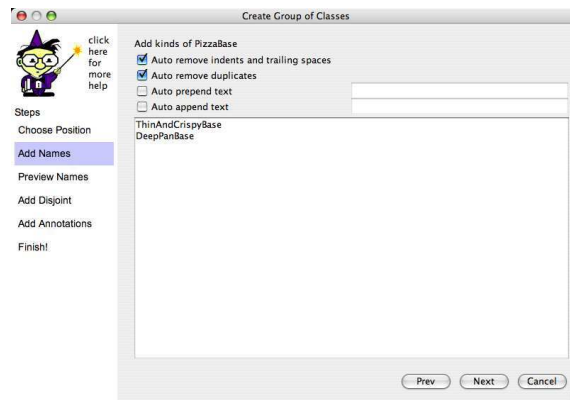


Figure 4.7: Add Group Of Classes Wizard: Enter classes page

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 is 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 hierarchy.
 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 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.
-

TIP

If we had imported the Dublin Core ontology (see section 7.2.5) then the Dublin Core annotation properties would have been available to annotate our classes in step 7 Exercise 5 . Dublin Core is a set of metadata elements that, in our case, can be used to annotate various elements of an ontology with information such as ‘creator’, ‘date’, ‘language’ etc. For more information see <http://dublincore.org/>^a.

^aAn ontology will be put into OWL-Full if the ontologies that are available on the Dublin Core website are imported. We recommend that an OWL-DL version of the Dublin Core ontology which is located in the Protégé ontology library is imported — details of this can be found in section 7.2.5

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 `MeatTopping`: `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 subclasses 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

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



Figure 4.8: Class Hierarchy

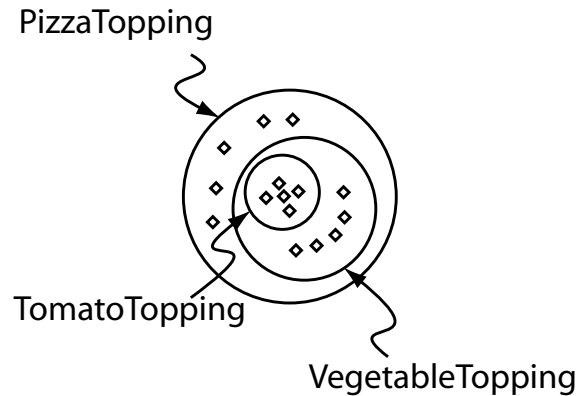


Figure 4.9: The Meaning Of Subclass — *All* individuals that are members of the class **TomatoTopping** are members of the class **VegetableTopping** and members of the class **PizzaTopping** as we have stated that **TomatoTopping** is a subclass of **VegetableTopping** which is a subclass of **PizzaTopping**

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**.

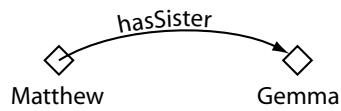
Exercise 7: Create an object property called hasIngredient

1. Switch to the ‘**Properties**’ 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**’).
-

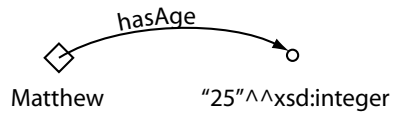
³See <http://www.w3.org/TR/xmlschema-2/> for more information on XML Schema Datatypes

⁴RDF = Resource Description Framework. See <http://www.w3.org/TR/rdf-primer/> for an excellent introduction to RDF.

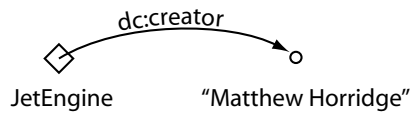
⁵Object properties and Datatype properties may be marked as Annotation properties



An object property linking the individual Matthew to the individual Gemma



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



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

Figure 4.10: The Different types of OWL Properties

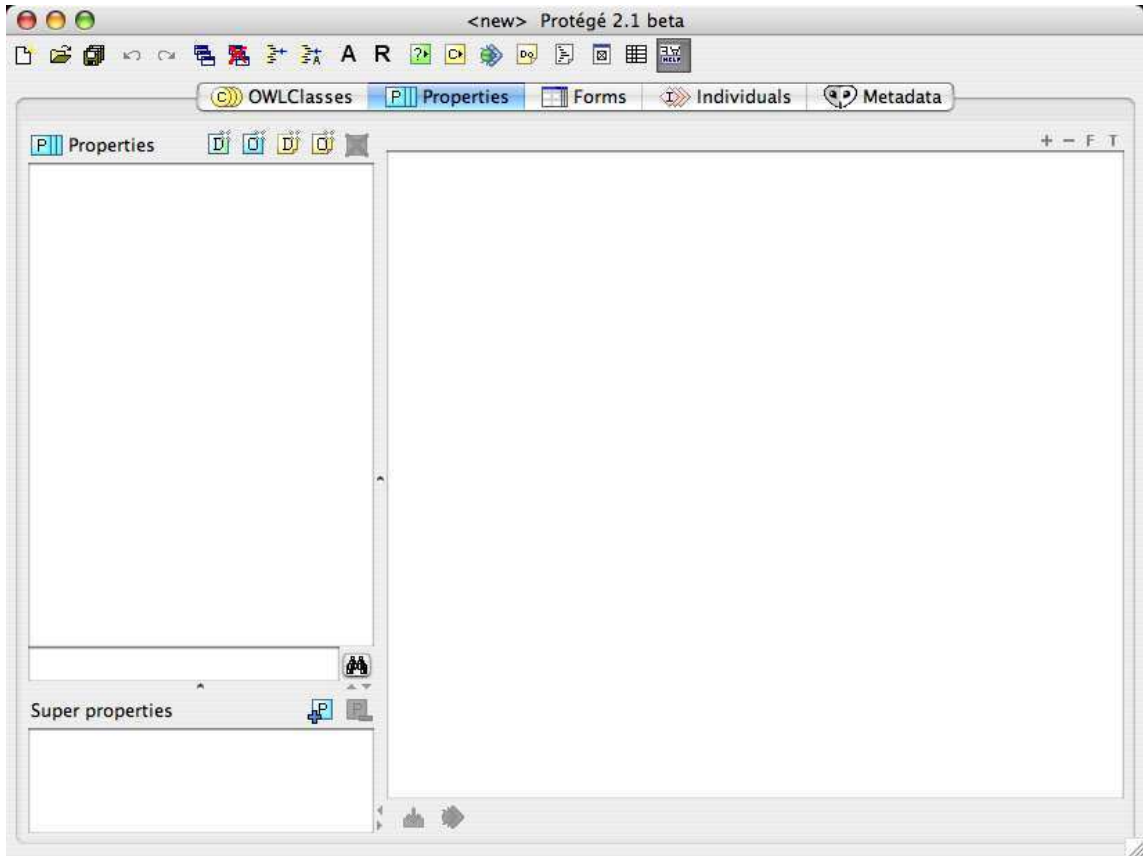


Figure 4.11: The PropertiesTab

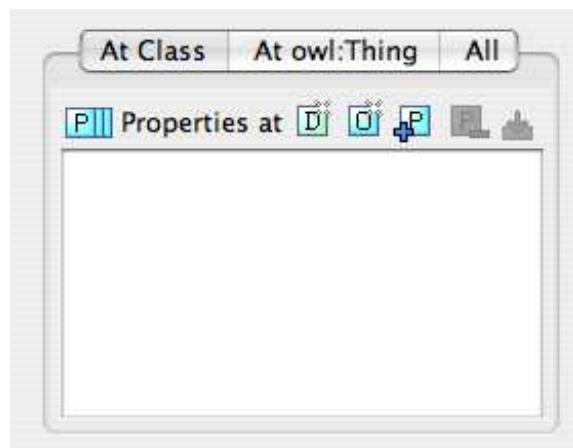


Figure 4.12: The Properties Widget

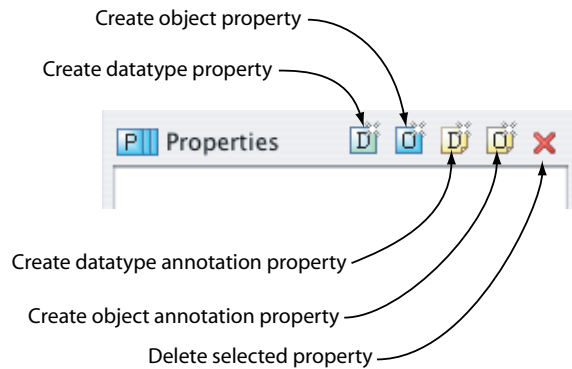


Figure 4.13: Property Creation Buttons — located on the Properties Tab above the property list/tree

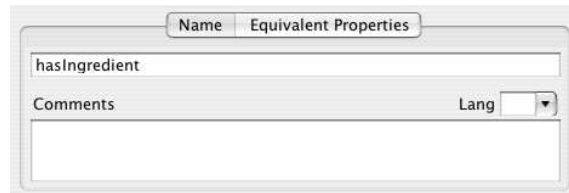


Figure 4.14: Property Name Widget

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.

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

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

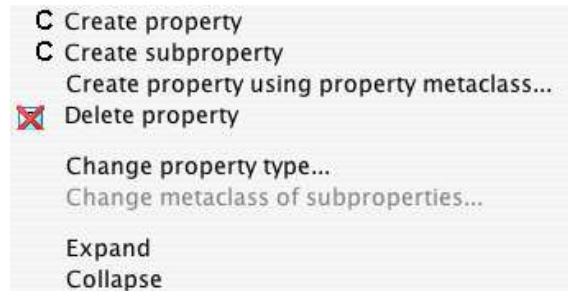


Figure 4.15: PropertyHierarchyMenu

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

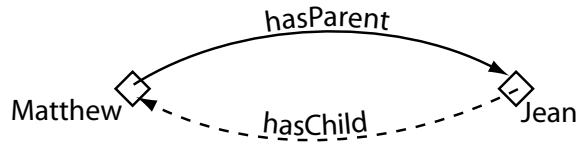


Figure 4.16: An Example Of An Inverse Property: `hasParent` has an inverse property that is `hasChild`

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 then close the dialog window (using the operating system close window button on the title bar). Notice that the `isBaseOf` property has been created as a sub property of the `isIngredientOf` property. This corresponds to the fact that `hasBase` is a sub property of `hasIngredient`, and `isIngredientOf` is the inverse property of `hasIngredient`.
 5. Select the `hasTopping` property.
 6. Press the **‘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:

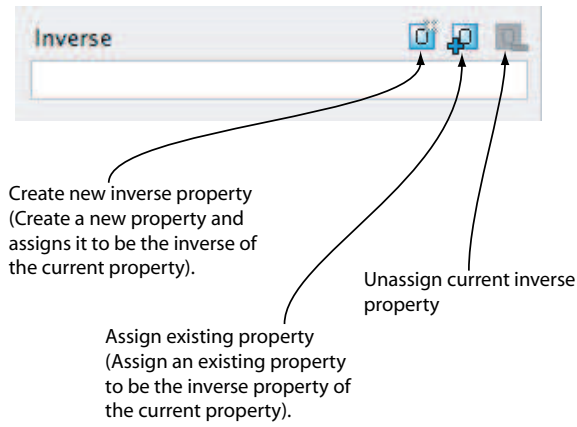


Figure 4.17: The Inverse Property Widget



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.

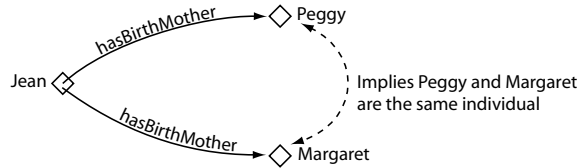


Figure 4.19: An Example Of A Functional Property: `hasBirthMother`



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.

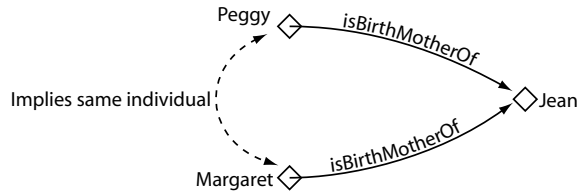


Figure 4.20: An Example Of An Inverse Functional Property: `isBirthMotherOf`

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

⁶The 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.

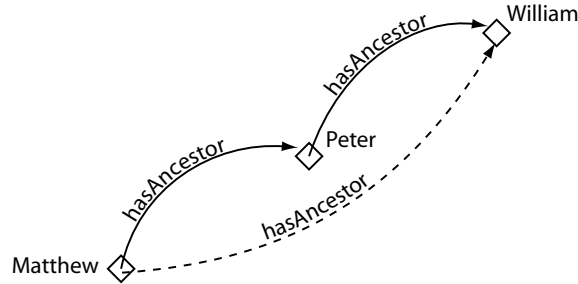


Figure 4.21: An Example Of A Transitive Property: `hasAncestor`

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.

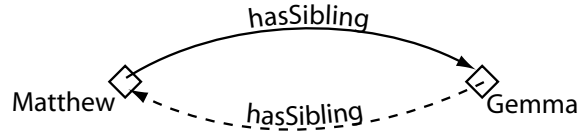


Figure 4.22: An Example Of A Symmetric Property: `hasSibling`

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.

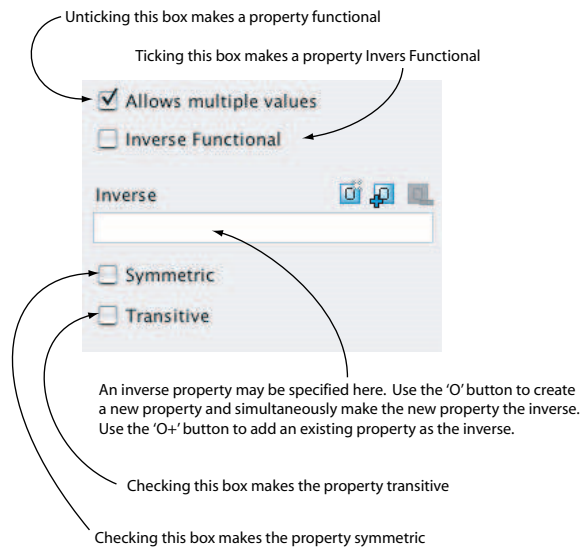


Figure 4.23: Property Characteristics Widgets



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

^aAt the time of writing this must be done manually in Protégé-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

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

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

Exercise 11: Make the `hasBase` property functional

1. Select the `hasBase` property.
2. Click the ‘**Allows multiple values**’ tick box on the ‘**Property Characteristics Widget**’ so that it is unticked.