MANCHESTER
1824

The University of Manchester

A Practical Introduction to Protégé OWL

Session 1: Primitive Classes

Nick Drummond, Matthew Horridge,
Olivier Dameron, Alan Rector, Hai Wang

Overview (morning)

- Tutorial Aims
- OWL Language Overview
  - Language constructs
- Primitive Pizzas
  - Creating a class hierarchy
  - Basic relations
- Basic Reasoning
  - Class consistency
  - Your first defined class
- Q&A

The University of Manchester

MANCHESTER 1824

Overview (afternoon)

- Formal Semantics of OWL
  - Harder but more fun
- Advanced Reasoning
  - Defined Classes
  - Using a reasoner for computing a classification
- Common Mistakes
- Q&A

The University of Manchester

MANCHESTER 1824

Aims of this morning

- Make OWL (DL) more approachable
- Get you used to the tool
- Give you a taste for the afternoon session

Exposing OWL

The University of Manchester

MANCHESTER 1824

What is OWL?

- OWL is the **W**eb **O**ntology **L**anguage
- It's part of the W3C World Wide Web Consortium **Semantic Web** framework
- It's a W3C standard

OWL

OWL has explicit formal semantics

Can therefore be used to capture knowledge in a **machine interpretable** way

MANCHESTER 1824

The University of Manchester

OWL helps us...

- Describe something, rather than just name it
- Class (BlueThing) does not mean anything
- Class (BlueThing complete

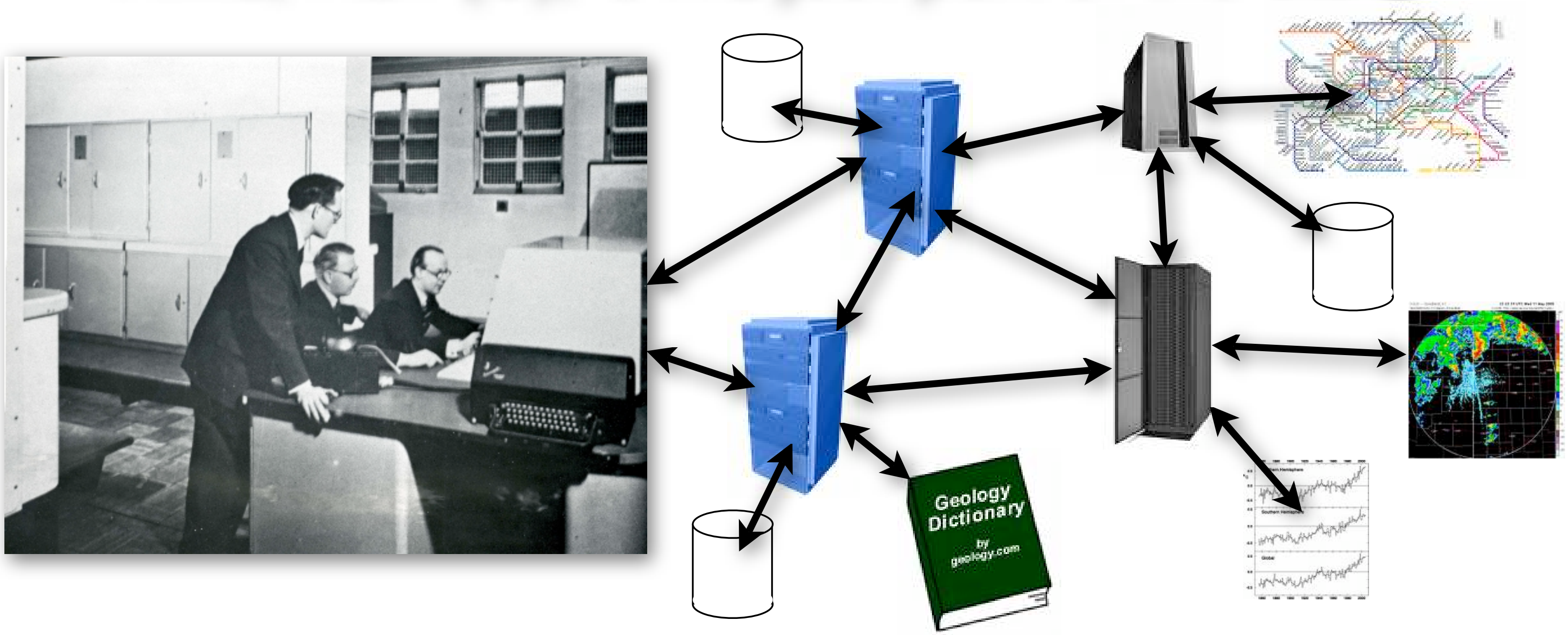
    owl:Thing

    restriction (hasColour someValuesFrom (Blue)))

  has an agreed meaning to any program accepting OWL semantics

What is the Semantic Web?

- A vision of a computer-understandable web
- Distributed knowledge and data in reusable form
- XML, RDF(S), OWL just part of the story



MANCHESTER 1824

The University of Manchester

What is the Semantic Web?

Scientific American 2001:

OWL and the Semantic Web

- A little semantics goes a long way
- Start small
- OWL is not an everything or nothing language
- Much can be gained from using the simplest of constructs and expanding on this later
- KISS

OWL and XML

- XML is a syntax
- E**X**tensible **M**arkup **L**anguage
- XML describes a tree structure
- XML was designed to improve interoperability by standardising syntax

OWL and RDF

- Another Semantic Web language
- **R**esource **D**escription **F**ramework
- RDF describes a graph of nodes and arcs, each normally identified by a URI
- RDF statements are **triples**
  - **subject** → **predicate** → **object**
  - **myhouse - islocatedIn - Manchester**
- Semantics are limited and use is unconstrained compared to OWL

The University
of Manchester

MANCHESTER
1824

OWL and RDFS

- **RDF S**chema
- Adds the notion of classes to RDF
- Allows hierarchies of classes and properties
- Allows simple constraints on properties
- OWL has the same interpretation of some RDFS statements (subsumption, domain and range)

OWL

The University of Manchester

MANCHESTER 1824

OWL and Frames

- 2 different modelling paradigms
  - Frames is object-oriented
  - OWL is based on set theory
- Both languages supported by Protégé
  - Native language is Frames
  - Only basic import/export between them
- Differences between them big subject
  - Overview talk by Hai Wang on Tuesday

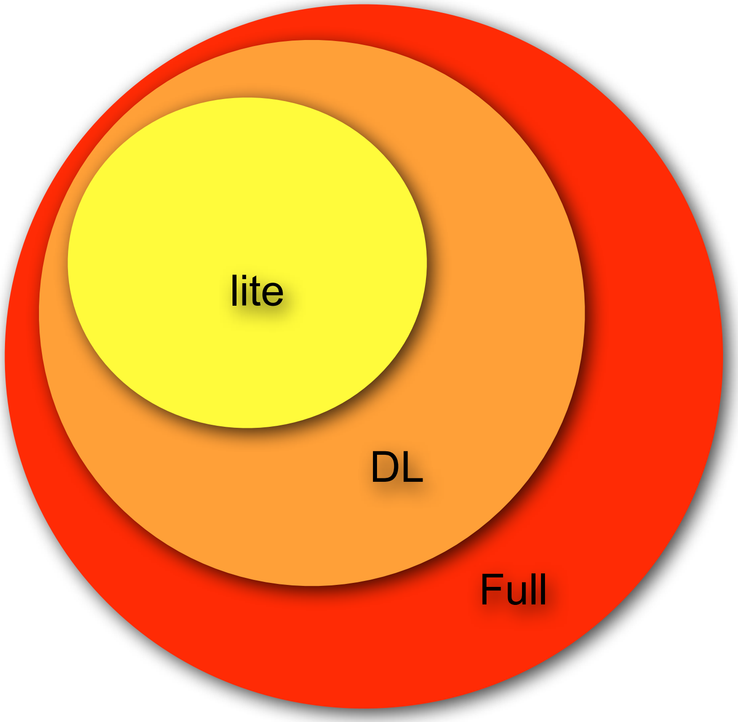
The University of Manchester

MANCHESTER 1824

OWL and Databases

- Databases are about **how data is stored**
- OWL is for describing domain knowledge
- Databases are **closed world**, whereas OWL is **open world** (more about this this afternoon)
- Triple stores are databases optimised for storing RDF/OWL statements

OWL comes in 3 Flavours



- **Lite** - partially restricted to aid learning curve
- **DL** = Description Logic
  Description Logics are a fragment of First Order Logic (FOL) that are decidable - this allows us to use DL reasoners (more later)
- **Full**
  unrestricted use of OWL constructs, but cannot perform DL reasoning

Syntax

- OWL is often thought of as an extension to RDF which is not strictly true
- OWL is a syntax independent language that has several common representations
- Many tools try to completely abstract away from the syntax

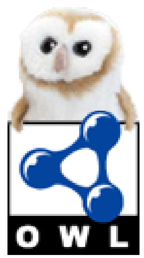
MANCHESTER 1824

The University of Manchester

OWL Syntax: abstract syntax

- One of the clearer human-readable syntaxes

```
Class(SpicyPizza complete
   annotation(rdfs:label "PizzaTemperada"@pt)
   annotation(rdfs:comment "Any pizza that has a spicy topping
                              is a SpicyPizza"@en)
   Pizza
   restriction(hasTopping someValuesFrom(SpicyTopping))
)
```



OWL Syntax: N3

- Recommended for human-readable fragments

```
default:SpicyPizza
      a owl:Class ;
      rdfs:comment "Any pizza that has a spicy topping is a
                                         SpicyPizza"@en ;
      rdfs:label "PizzaTemperada"@pt ;
      owl:equivalentClass
              [ a owl:Class ;
                owl:intersectionOf (default:Pizza [ a owl:Restriction ;
                            owl:onProperty default:hasTopping ;
                            owl:someValuesFrom default:SpicyTopping
                          ])
              ] .
```

MANCHESTER 1824

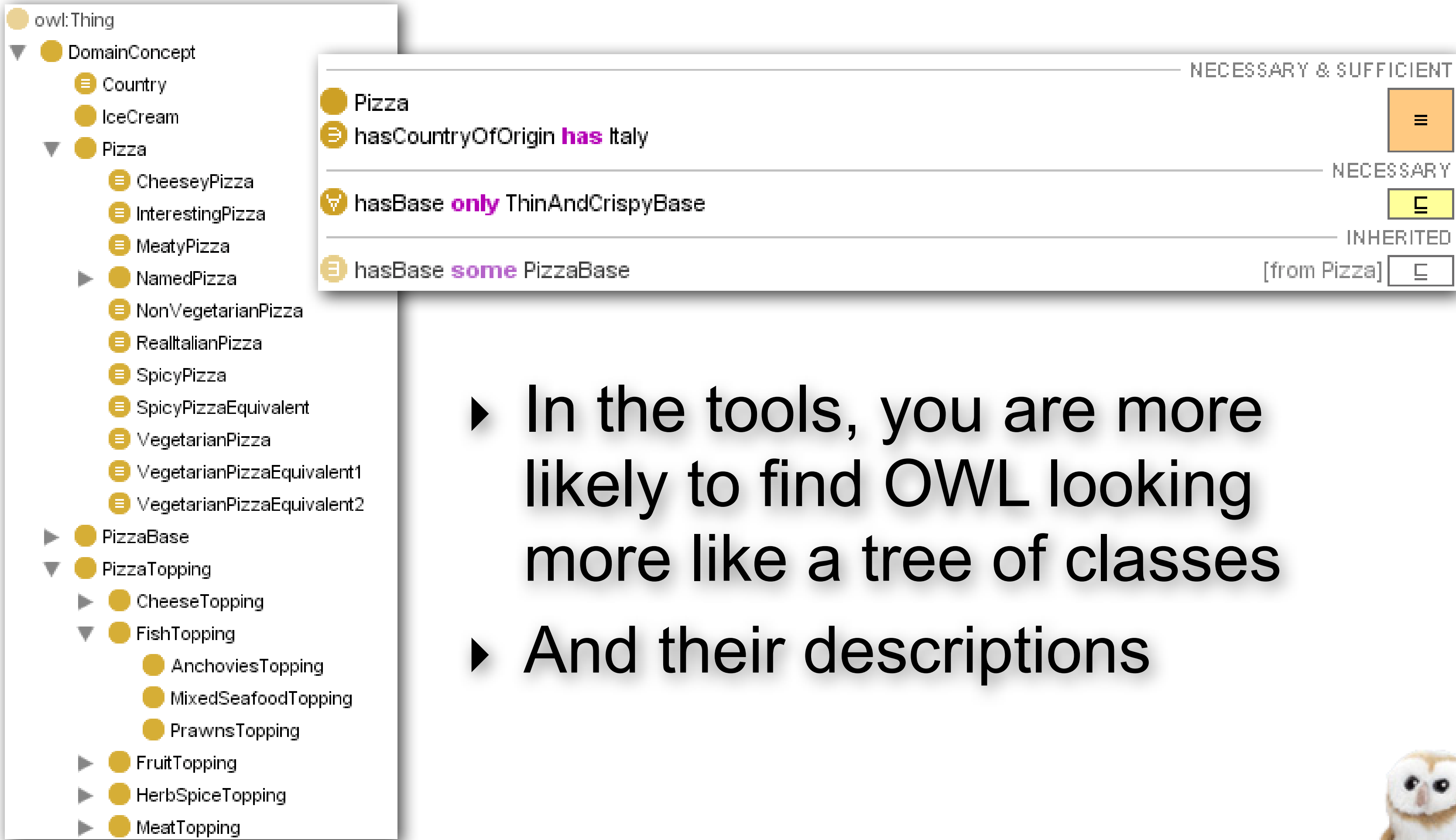
The University of Manchester

OWL Syntax: RDF/XML

- Recommended for serialisation

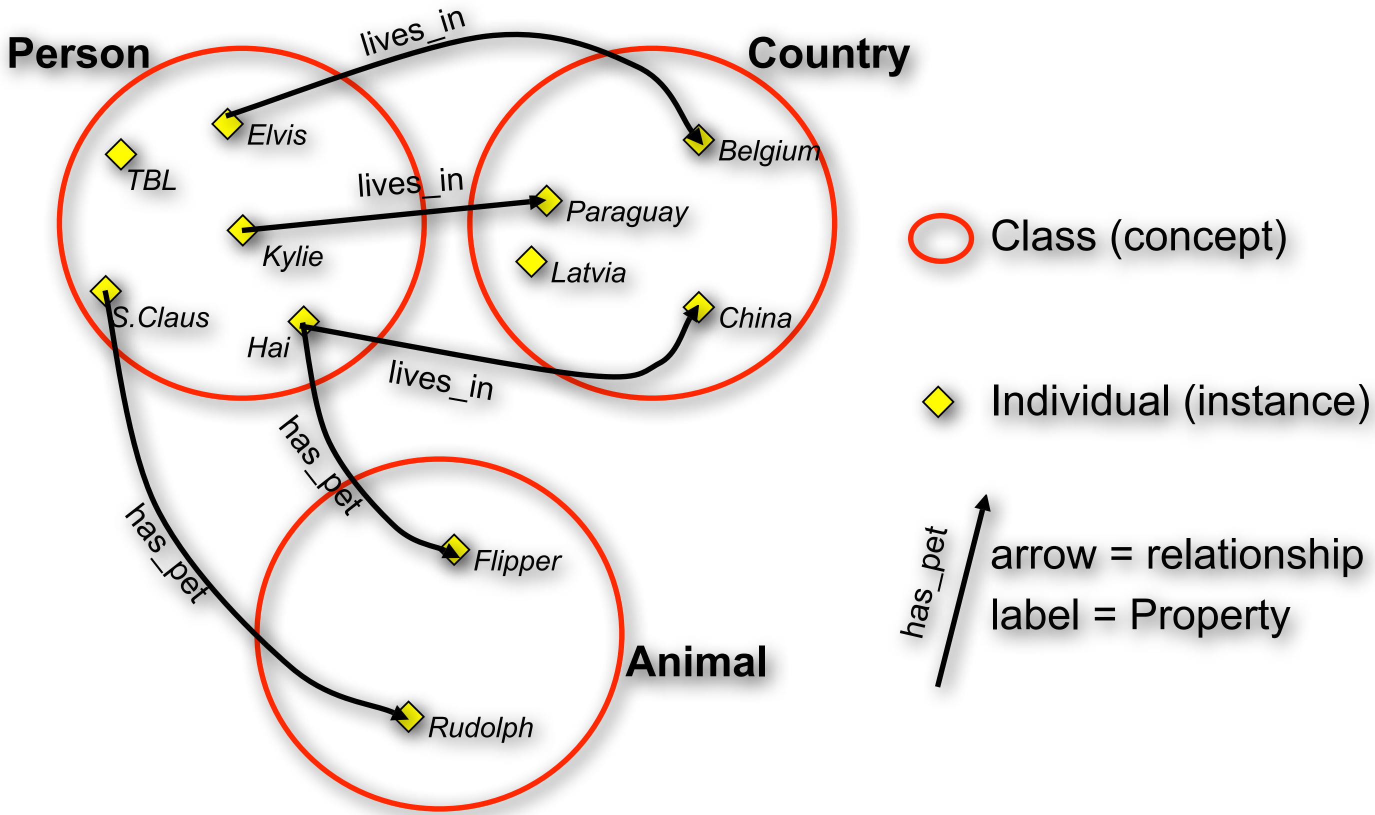
```
<owl:Class rdf:ID="SpicyPizza">
   <rdfs:label xml:lang="pt">PizzaTemperada</rdfs:label>
   <rdfs:comment xml:lang="en">Any pizza that has a spicy topping is a SpicyPizza</
   rdfs:comment>
   <owl:equivalentClass>
     <owl:Class>
       <owl:intersectionOf rdf:parseType="Collection">
         <owl:Class rdf:about="#Pizza"/>
         <owl:Restriction>
           <owl:onProperty>
             <owl:ObjectProperty rdf:about="#hasTopping"/>
           </owl:onProperty>
           <owl:someValuesFrom rdf:resource="#SpicyTopping"/>
         </owl:Restriction>
       </owl:intersectionOf>
     </owl:Class>
   </owl:equivalentClass>
 </owl:Class>
```

Tools "Hiding the Syntax"



- In the tools, you are more likely to find OWL looking more like a tree of classes
- And their descriptions

OWL Constructs Overview



The University of Manchester

MANCHESTER 1824

OWL Constructs: Classes

- **Eg Mammal, Tree, Person, Building, Fluid**
- Classes are **sets** of Individuals
- aka “Type”, “Concept”, “Category”, “Kind”
- Membership of a Class is dependent on its **logical description**, not its name
- Classes do not have to be named – they can be logical expressions – eg things that have colour Blue
- A Class should be described such that it is possible for it to contain Individuals (unless the intention is to represent the empty class)

The University of Manchester

MANCHESTER 1824

OWL Constructs: Individuals

- **Eg me, you, this tutorial, this room**
- Individuals are the objects in the domain
- aka “Instance”, “Object”
- Individuals may be (and are likely to be) a member of multiple Classes

OWL Constructs: Properties

- **Eg hasPart, isInhabitedBy, isNextTo, occursBefore**
- Object Properties are used to **relate Individuals**
- Datatype Properties relate Individuals to data values
- We generally state that "Individuals are related **along** a given property"
- Relationships in OWL are **binary** and can be represented in **triples**:
  - **subject** → **predicate** → **object**
  - nick → worksWith → matthew

The University of Manchester

MANCHESTER 1824

A note on naming

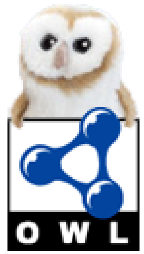
- Named things (classes, properties and individuals) have **unique identifiers**
- In Semantic Web languages these are **URIs**
- Something with the same URI is the same object
- This is so we can refer to things in someone else's ontology

- Full URIs are hidden in most tools:

  http://www.co-ode.org/ontologies/pizza/2006/07/18/pizza.owl#PizzaTopping

  is a bit harder to read than:

  PizzaTopping
- **URIs do not have to be URLs**



What can be said in OWL?

- “All pizzas are a kind of food”
- “No kinds of meat are vegetables”
- “All pizzas must have only one base but at least one topping”
- “Ingredients must be some kind of food”
- “Any pizza that has no meat or fish on it must be vegetarian”
- “Interesting pizzas have at least 4 toppings”
- “Spicy pizzas are pizzas that have at least one ingredient that is spicy”

MANCHESTER 1824

The University of Manchester

The Pizza Ontology

proudly brought to you by

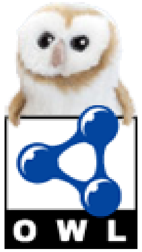
The Manchester Pizza Finder

MANCHESTER 1824

The University of Manchester

Our Domain

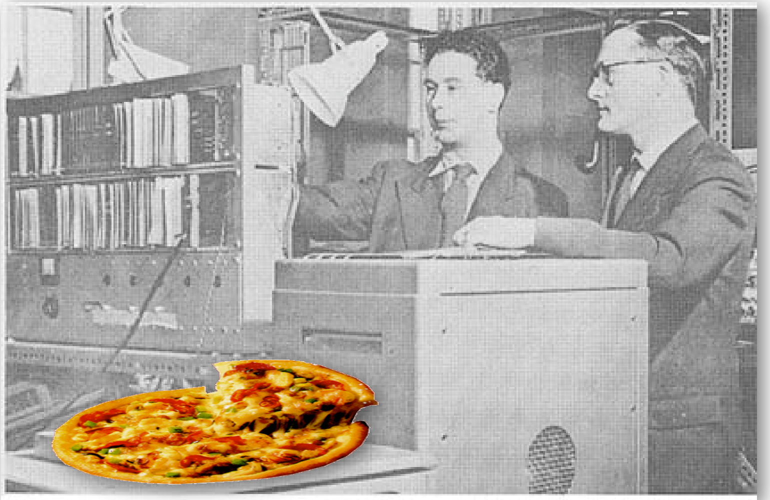
- Pizzas have been used in Manchester tutorials for years



Pizzas…

- Tutorial developed by BioHealth Informatics Group in Manchester (in alphabetical order)
Mike Bada, Sean Bechhofer, Carole Goble, Matthew Horridge, Ian Horrocks, Alan Rector, Jeremy Rogers, Robert Stevens, Chris Wroe

Pizzas...



- are fun
- are internationally known
- are highly compositional
- are limited in scope
- are fairly uncontroversial
  - Although arguments still break out over representation
  - ARGUING IS NOT BAD!

You are the Expert

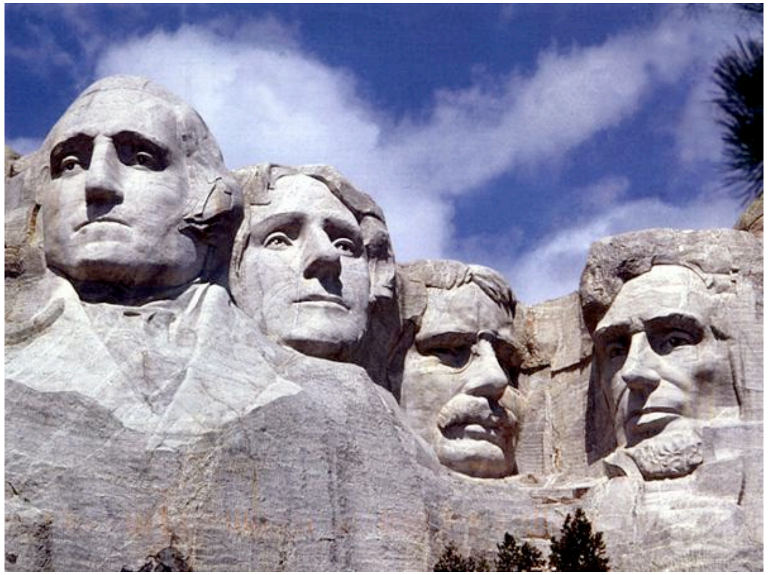
- Most often it is not the domain expert that formalises their knowledge
- Because of the complexity of the modelling task it is normally a specialist “knowledge engineer”
  Hopefully, as tools get easier to use, this will change
- Having access to experts is critical for most domains
- Luckily, we are all experts in Pizzas, so we just need some material to verify our knowledge…

Reference Materials

- Having references to validate decisions, and act as provenance can be useful for maintaining an ontology
- Mistakes, omissions and intentions can be more easily traced if a reference can be made
  - When building, we highly recommend documenting your model as you go – keeping provenance information is a good way of doing this
- We have pizza menus available for inspiration

Our Ontology

- Some things get built just to impress

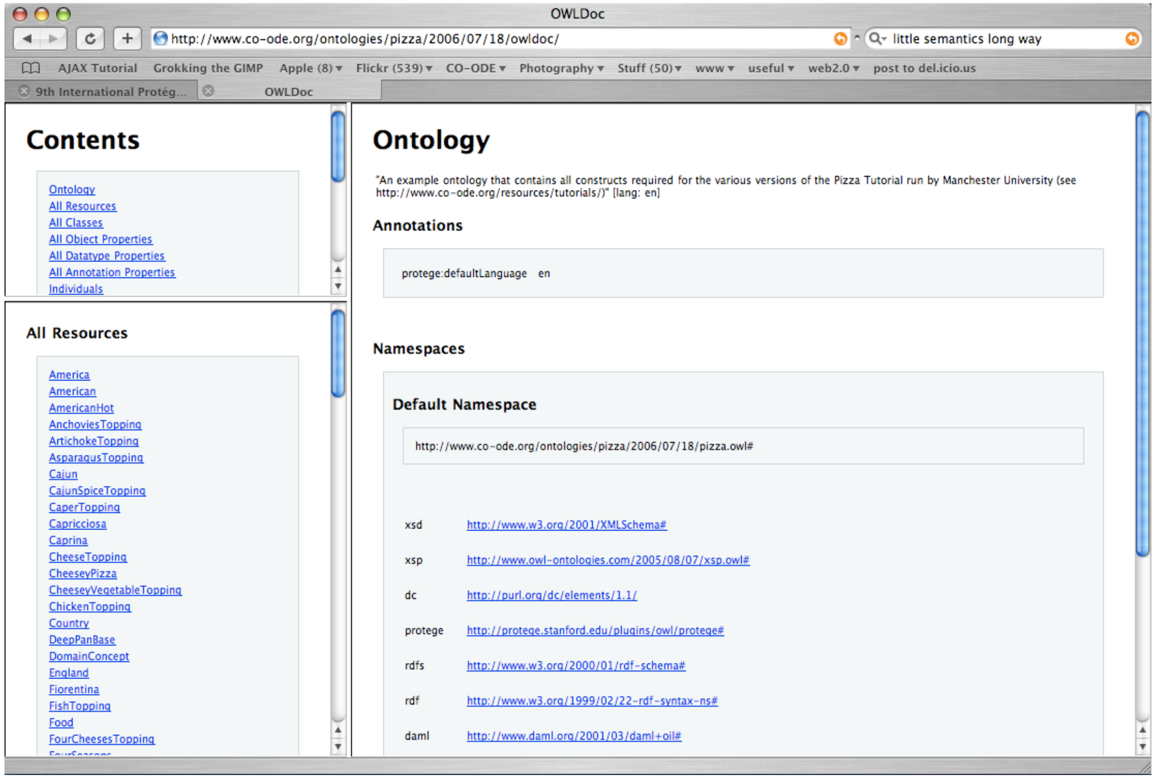


- Ontologies are not just there to look pretty
- Have an application in mind before starting

Demo Ontology

Our Pizza Ontology is available from:

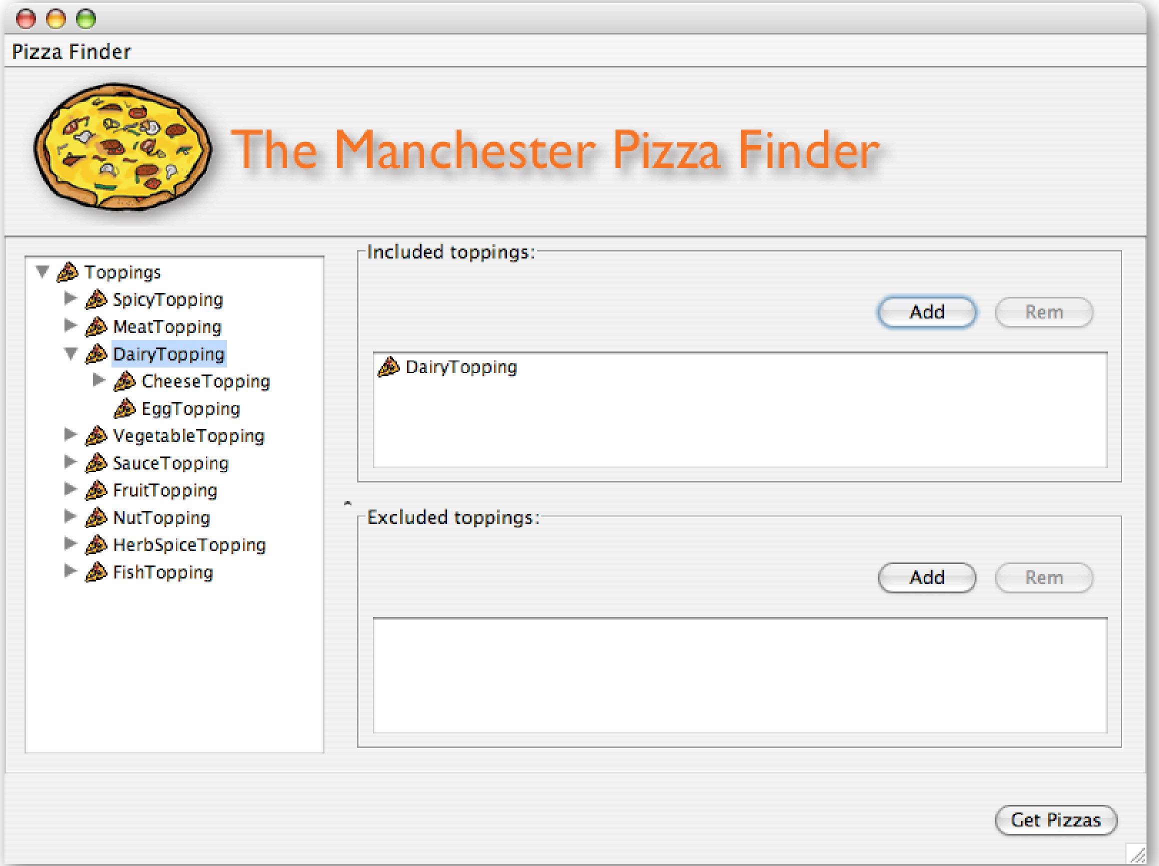
www.co-ode.org/ontologies/pizza/



Classes vs Instances

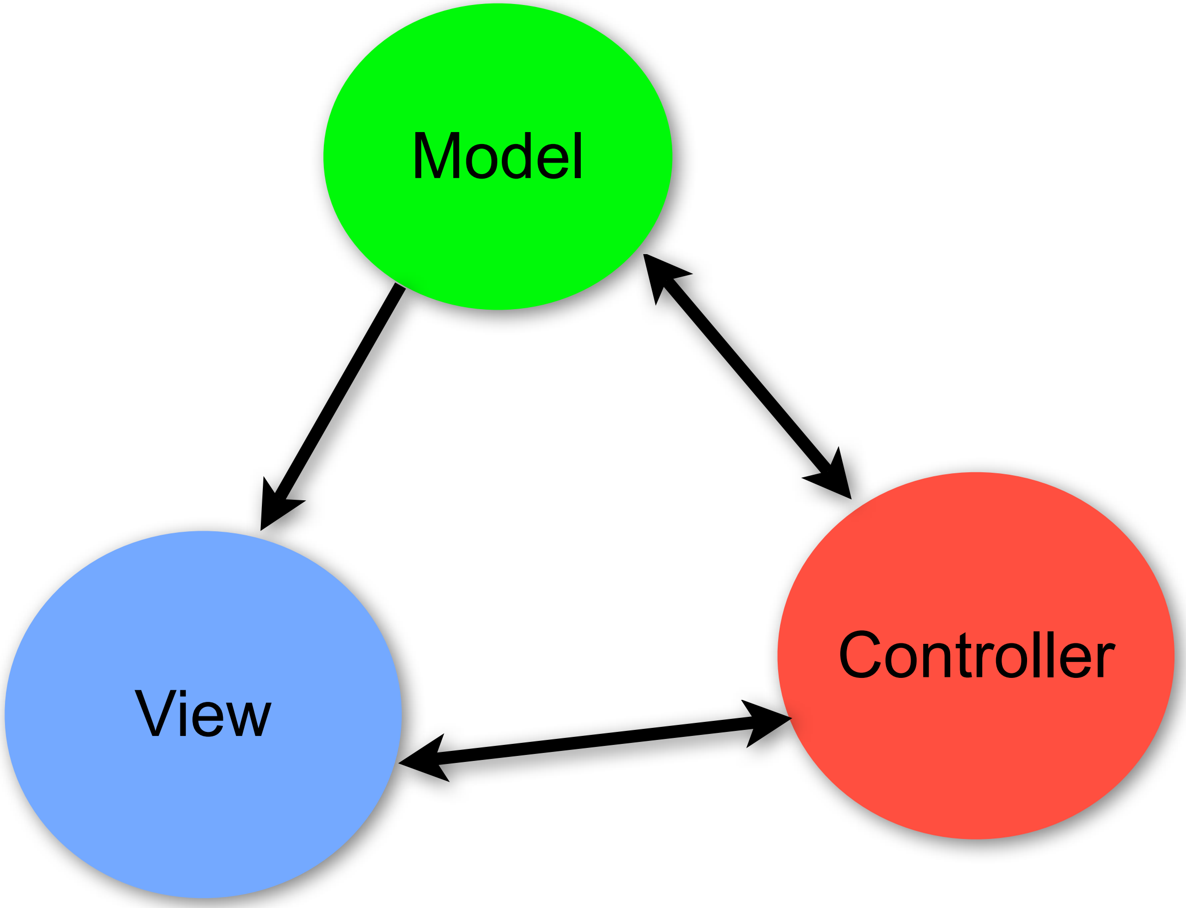
- You may note that the ontology consists almost completely of Classes
- Ontologies are about knowledge, so we only use individuals when necessary to describe a class
- Be careful adding Individuals to your ontology as they can restrict its reusability
  - eg you cannot create a new kind of Cheese if Cheese is an individual

Our Application

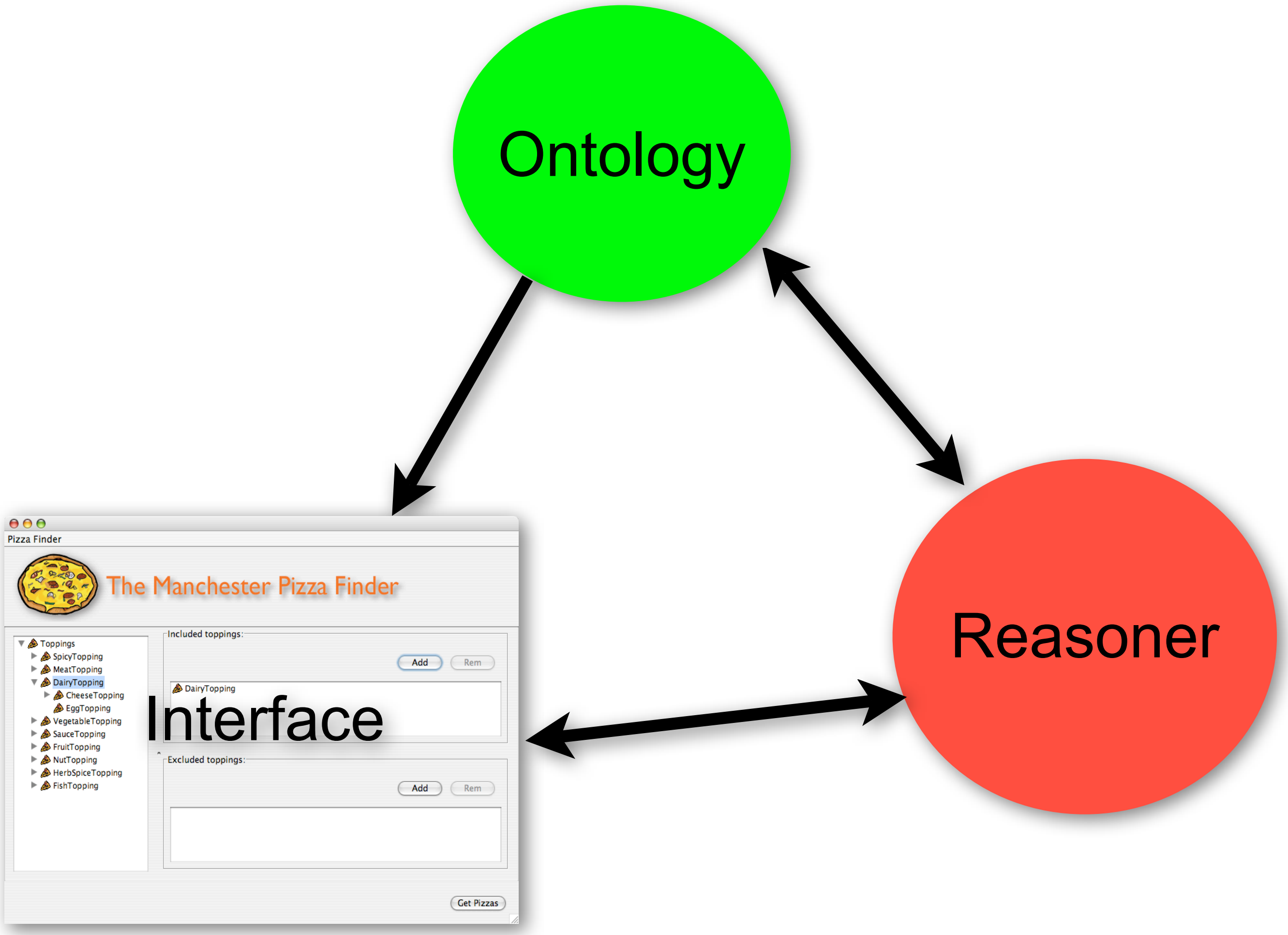


www.co-ode.org/downloads/pizzafinder/

Pizza Finder Architecture

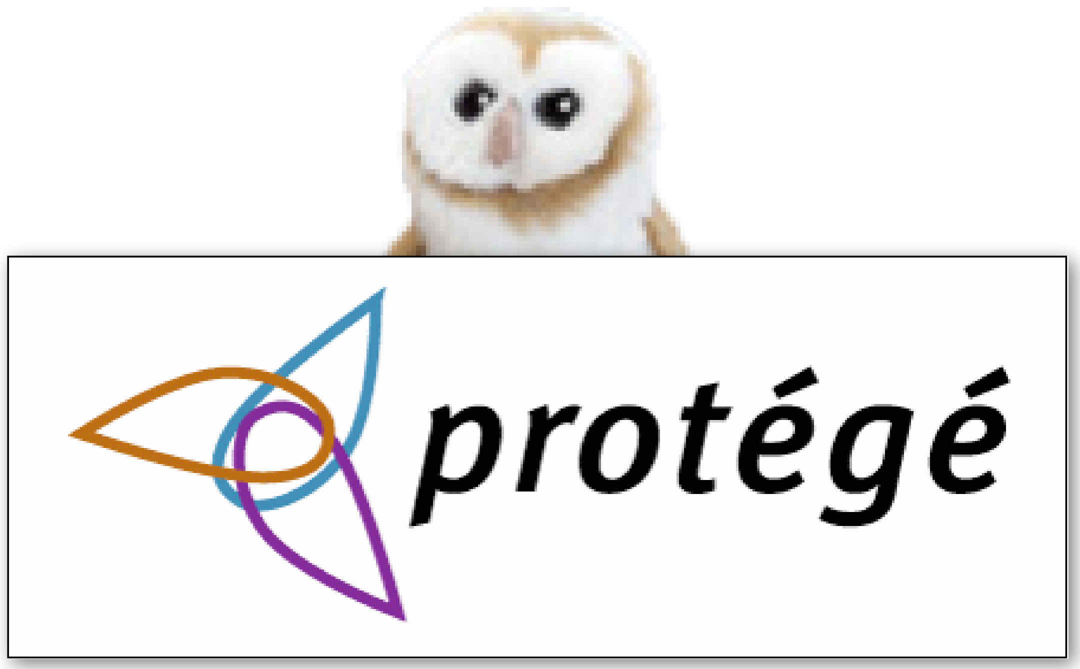


Pizza Finder Architecture



Plug a Pizza Ontology

- The PizzaFinder application has been developed so that you can create your own pizza ontology and plug it in to see it in action
- At the end of the day, let us know if you want to try this


protégé

The University of Manchester

Protégé-OWL = Protégé + OWL

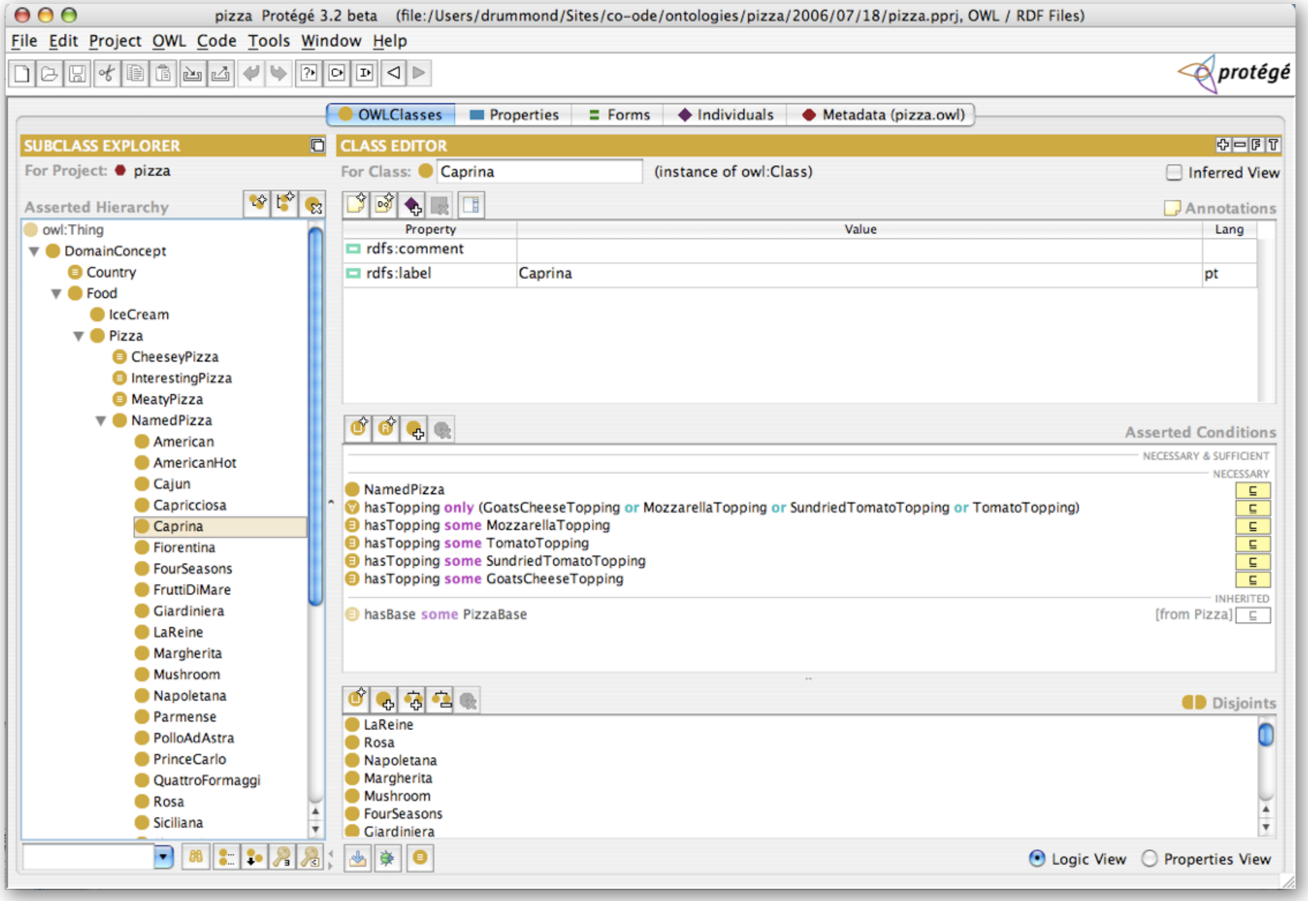
- core is based on Frames (object oriented) modelling
- has an open architecture that allows other modelling languages to be built on top
- supports development of plugins to allow backend / interface extensions
- supports OWL through the Protégé-OWL plugin

So let's have a look…

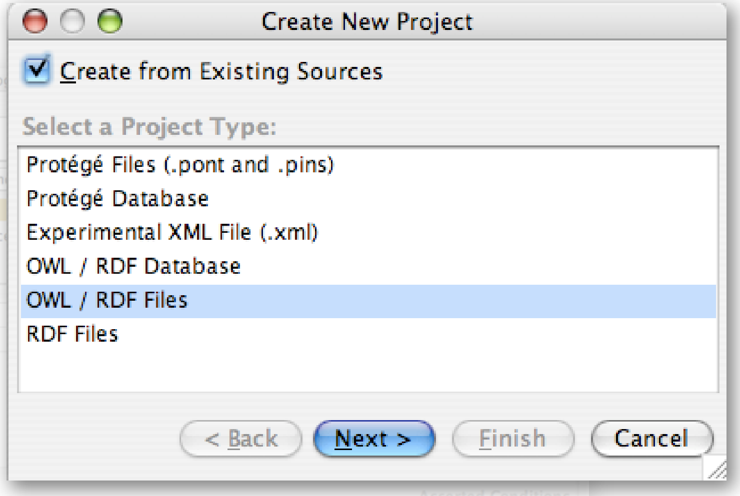
MANCHESTER
1824

The University
of Manchester

Protégé-OWL



Loading OWL files

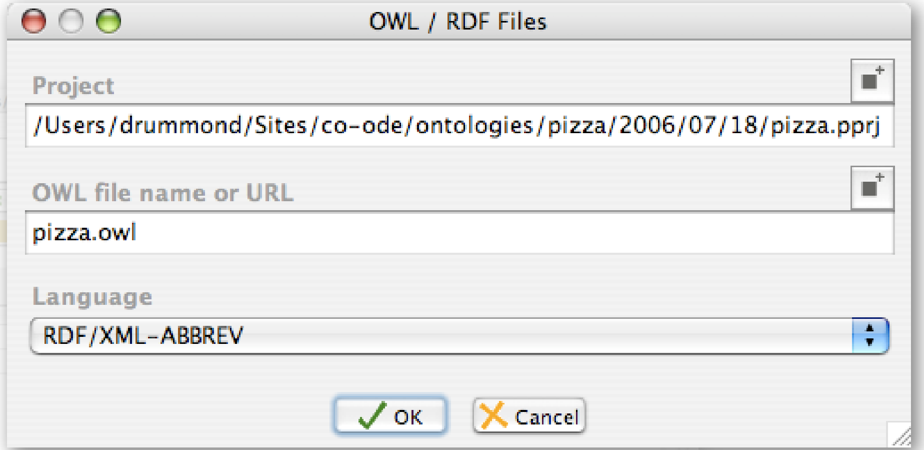


- If you only have an OWL file:
  - File → New Project
  - Select OWL Files as the type
  - Tick Create from existing sources
  - Next to select the .owl file
- If you've got a valid project file:
  - File → Open Project
  - select the .pprj file

MANCHESTER
1824

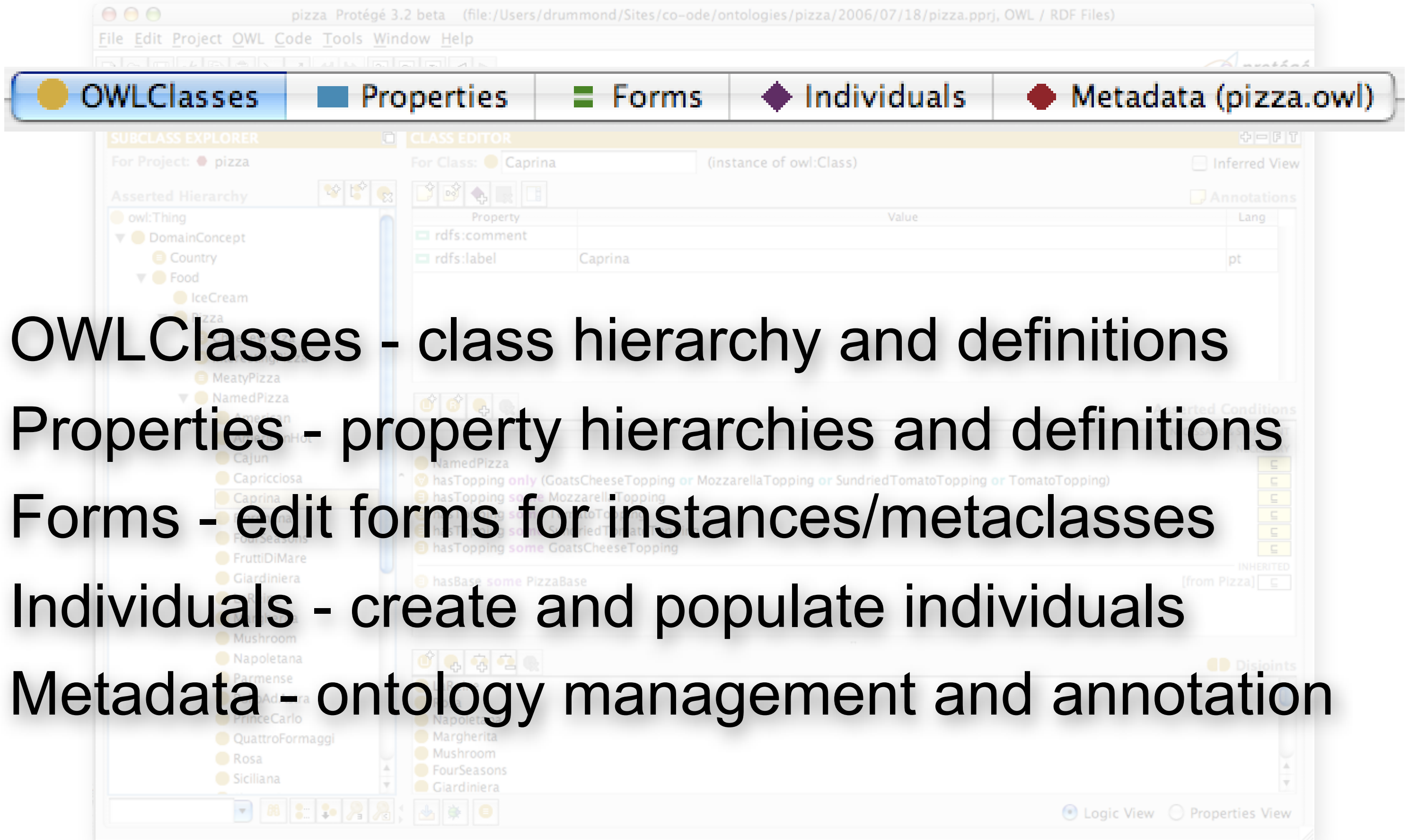
The University of Manchester

Saving OWL Files



- Select File → Save Project As
  A dialog (as shown) will pop up
- Select a file directly by clicking the button on the top right
  You will notice that 2 files are created
  .pprj – the project file
    this just stores information about the GUI and the workspace
  .owl – the OWL file
    this is where your ontology is stored in RDF/OWL format

Protégé-OWL Tabs



- OWLClasses - class hierarchy and definitions
- Properties - property hierarchies and definitions
- Forms - edit forms for instances/metaclasses
- Individuals - create and populate individuals
- Metadata - ontology management and annotation

The University of Manchester

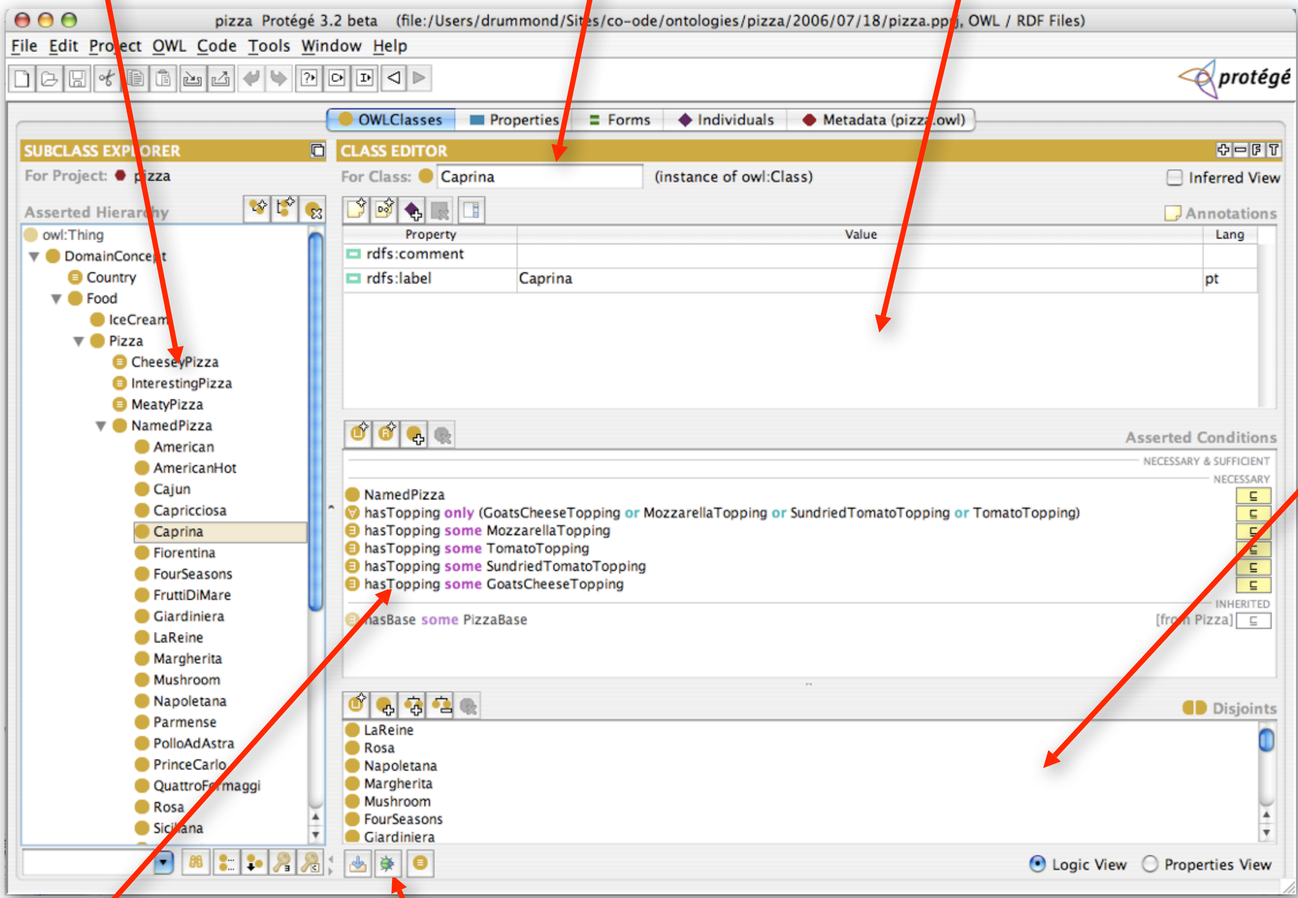
MANCHESTER 1824

OWL Classes Tab

Asserted Class hierarchy

Class name

Class annotations (for class metadata and documentation)



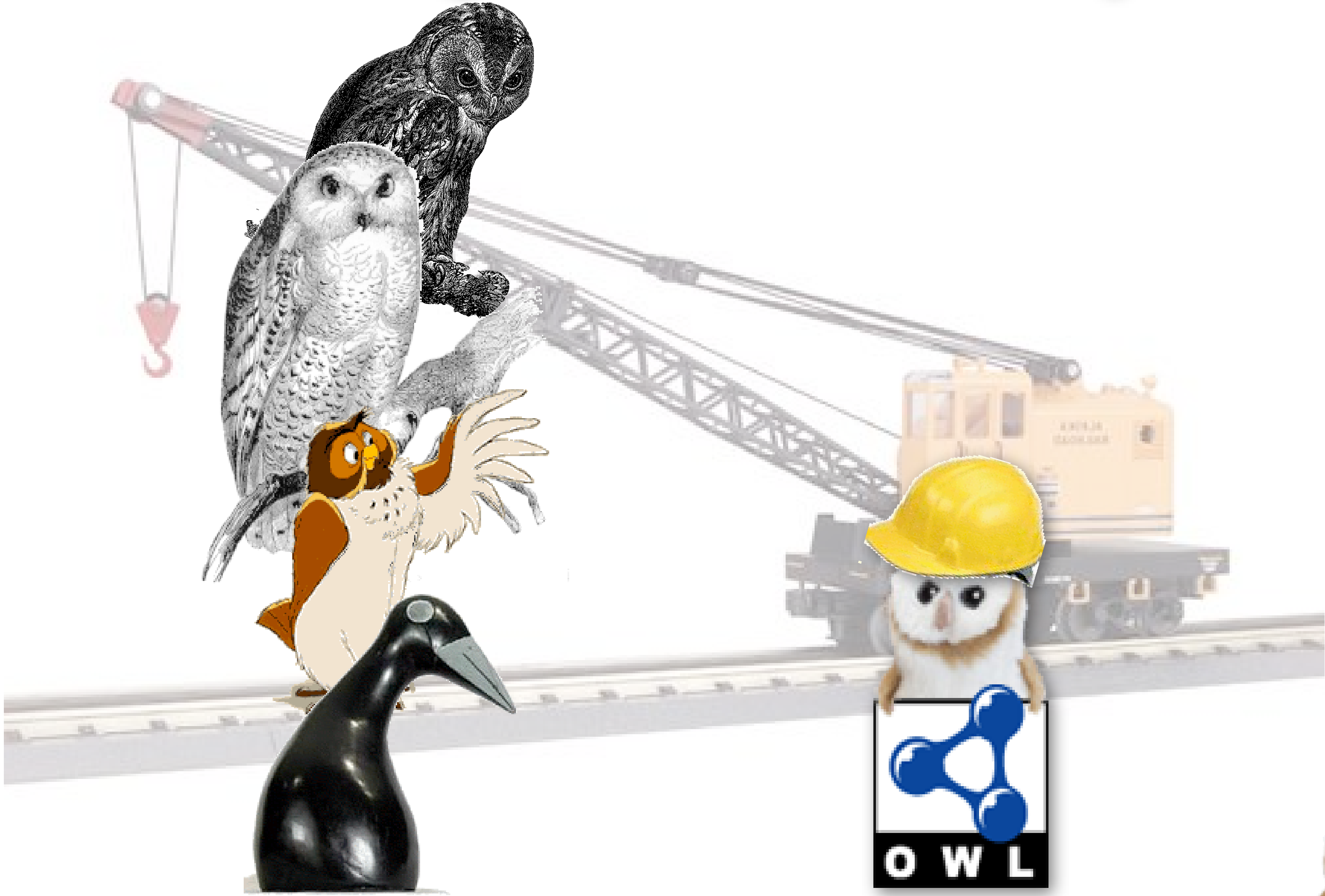
Disjoints widget

Conditions Widget

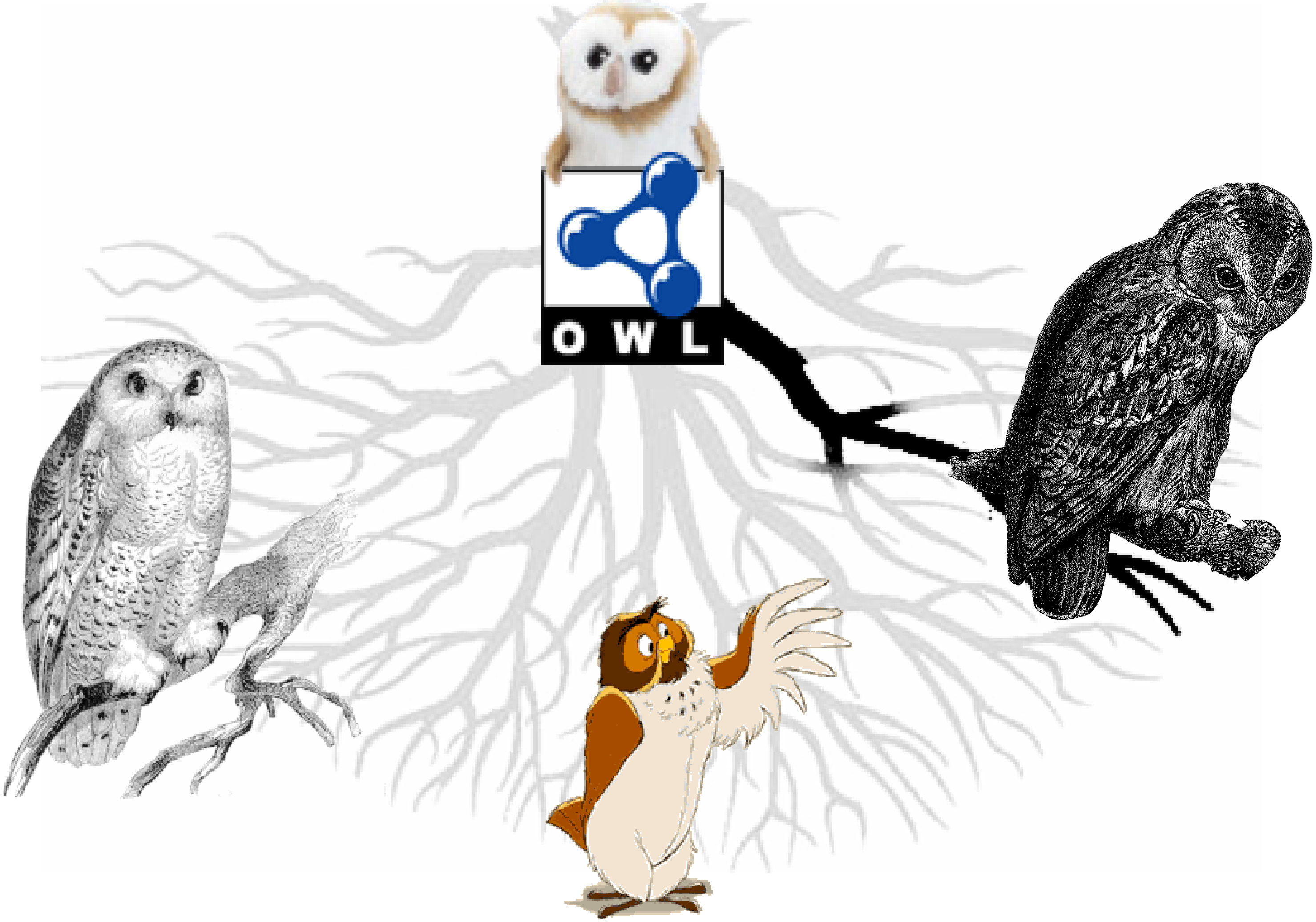
Class-specific tools (find usage etc)

MANCHESTER
1824
The University
of Manchester

Building a Class Hierarchy

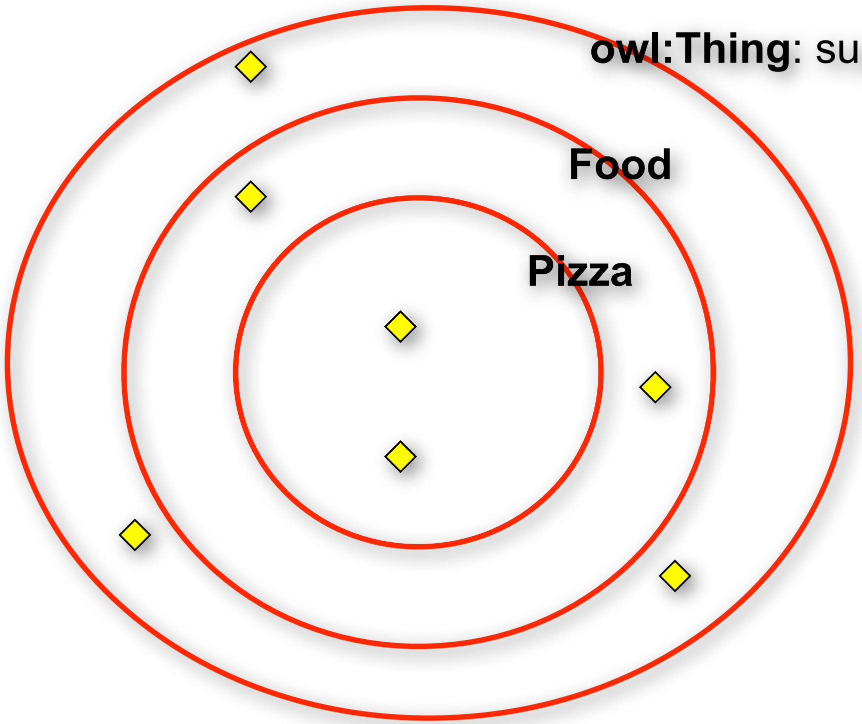


Subsumption



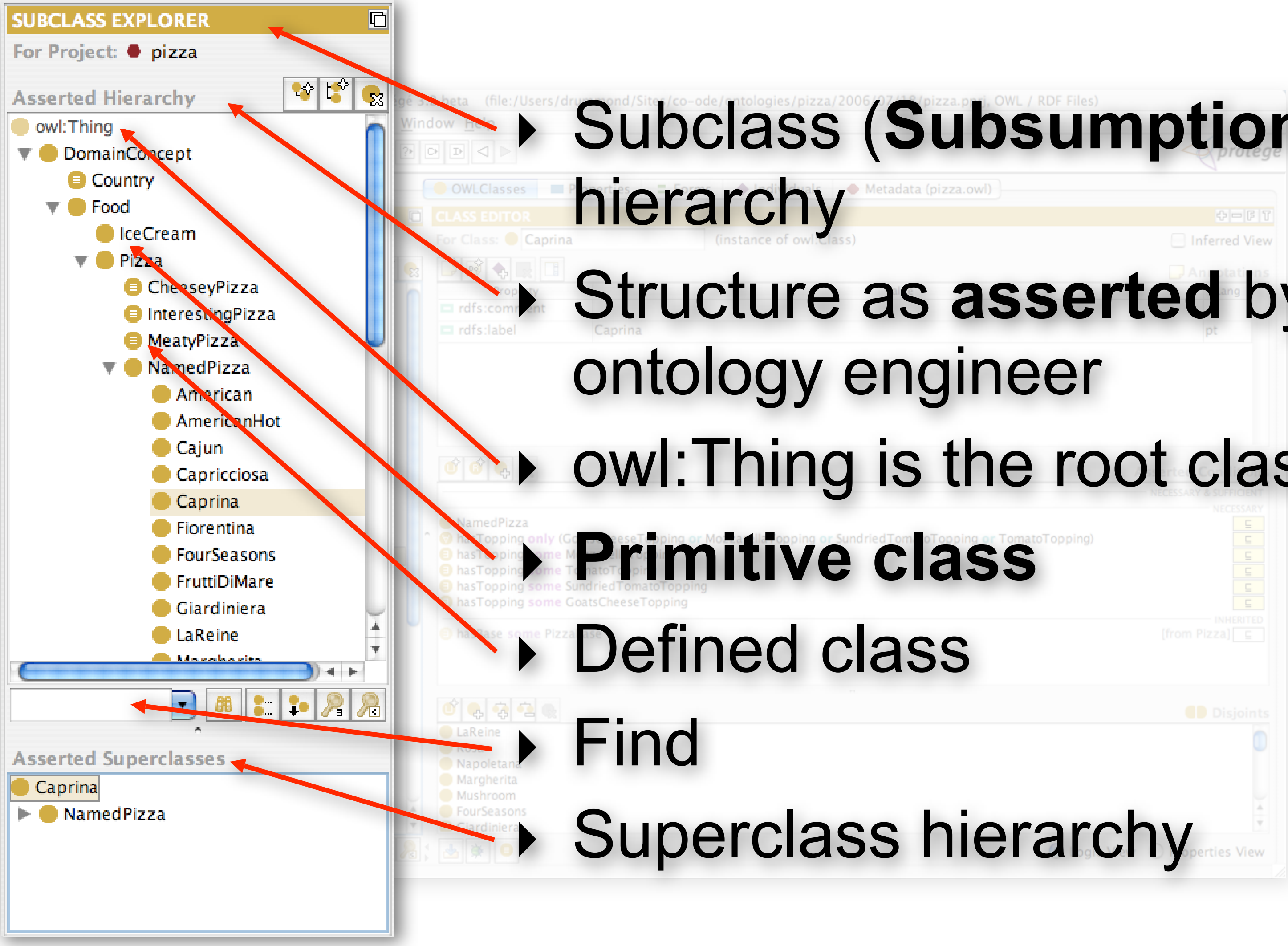
What is Subsumption?

- Superclass/subclass relationship, “isa”
- **All** members of a subclass are members of its superclasses



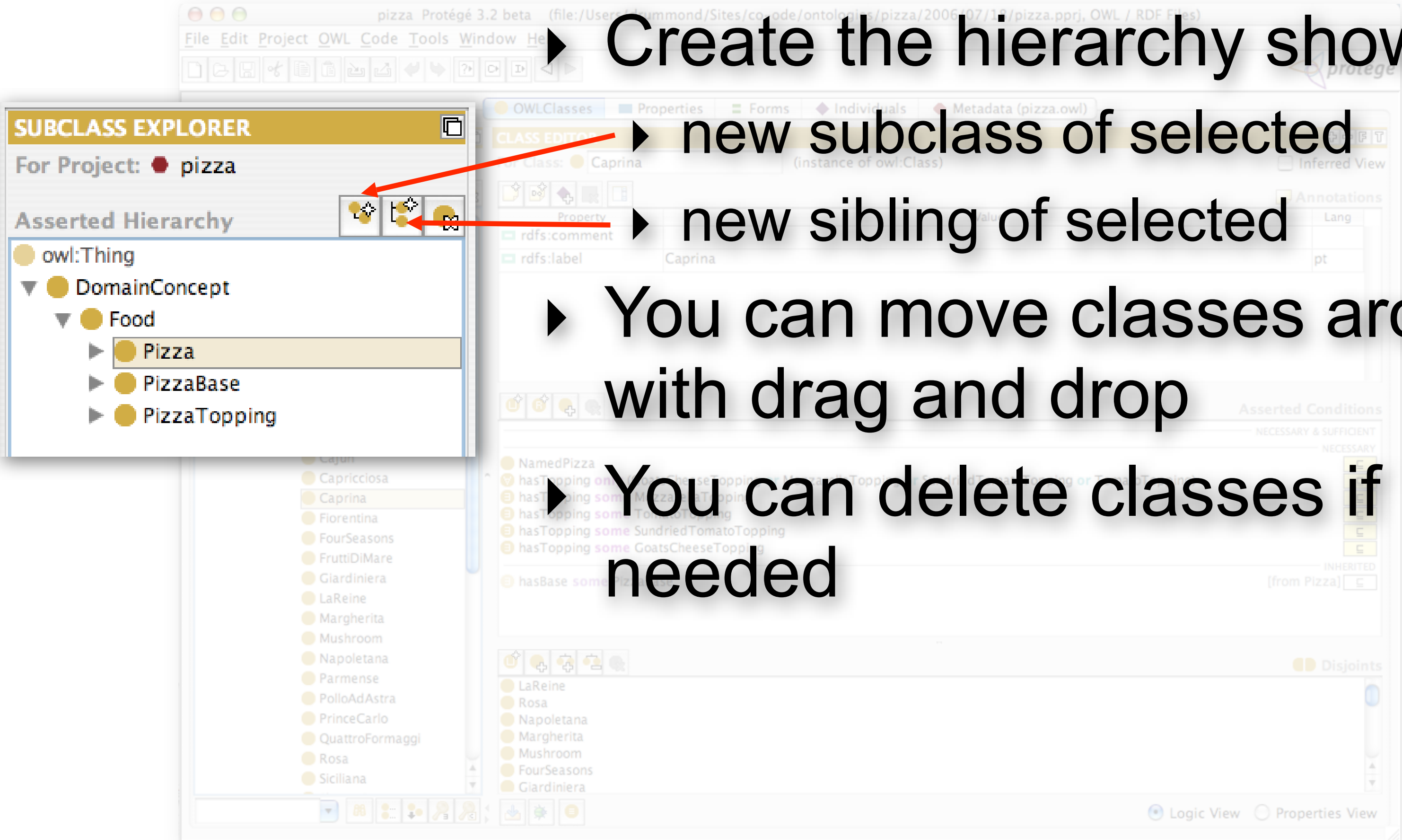
- **Food** subsumes **Pizza**
- **Food** is a superclass of **Pizza**
- **Pizza** is a subclass of **Food**
- **All** members of **Pizza** are also members of **Food**
- **Everything** is a member of **owl:Thing**

Class Hierarchy



- Subclass (**Subsumption**) hierarchy
- Structure as **asserted** by the ontology engineer
- owl:Thing is the root class
- **Primitive class**
- Defined class
- Find
- Superclass hierarchy

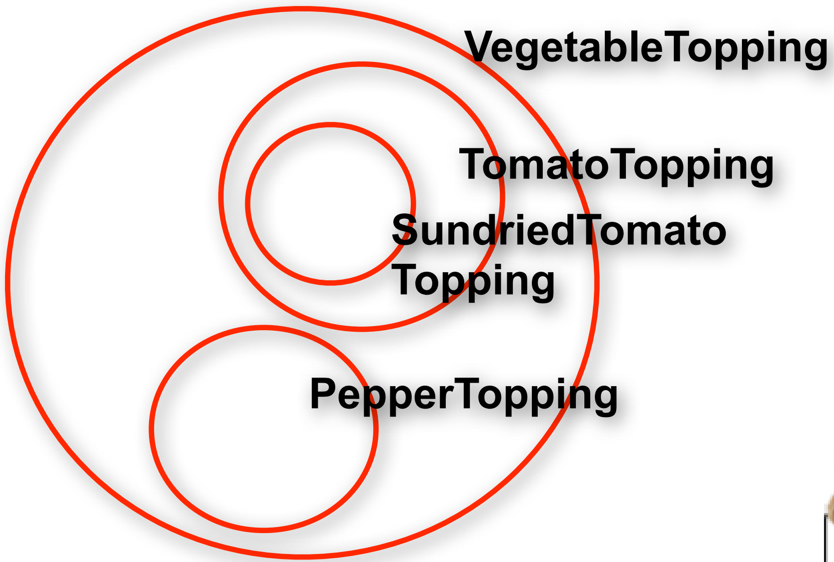
Create a Class Hierarchy



- Create the hierarchy shown
  - new subclass of selected
  - new sibling of selected
- You can move classes around with drag and drop
- You can delete classes if needed

Create a Class Hierarchy

- Create subclasses of **PizzaTopping**
- Think of some abstract classes to categorise your toppings
- Include at least the following 4:
  - **MeatTopping**
  - **CheeseTopping**
  - **MozzarellaTopping**
  - **TomatoTopping**
- More examples:

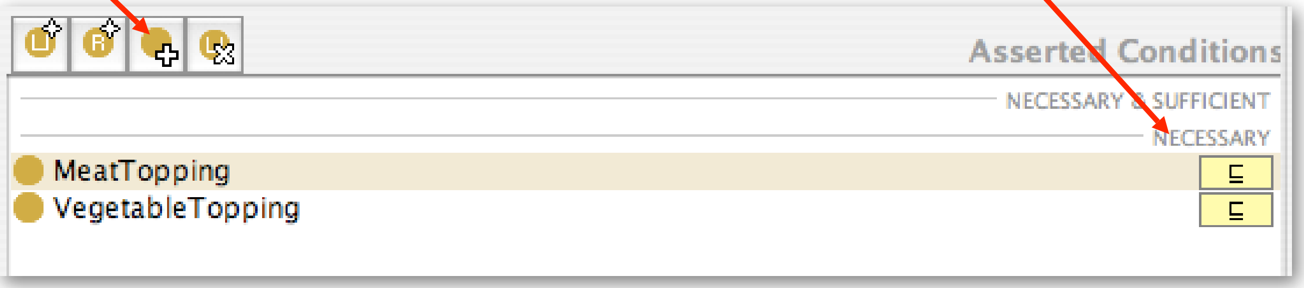


The University of Manchester

MANCHESTER 1824

Create a Class Hierarchy

- Create a **MeatyVegetableTopping**
- To add multiple superclasses to a class
  - first create the class
  - then use the conditions widget to add a new superclass
  - make sure “Necessary” is highlighted
  - select an existing class to add



MANCHESTER 1824

The University of Manchester

Create a Class Hierarchy

- You will notice that we use naming conventions for our ontology entities
- Typically, we use **CamelNotation** with a starting capital for Classes
- Use whatever conventions you like
- It is helpful to be consistent – especially when trying to find things in your ontology

MANCHESTER 1824

The University of Manchester

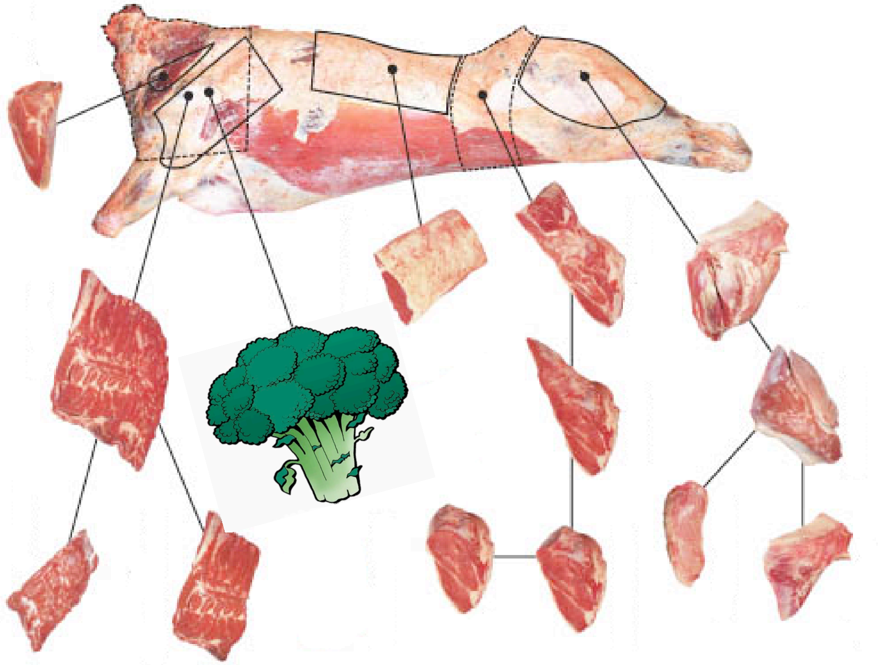
What is a MeatyVegetableTopping?

- Does it make sense?
- Can we check for mistakes like this?
- If we have a decent model, we can use a reasoner
- This is one of the main advantages of using a logic-based formalism such as OWL-DL

Checking our Model

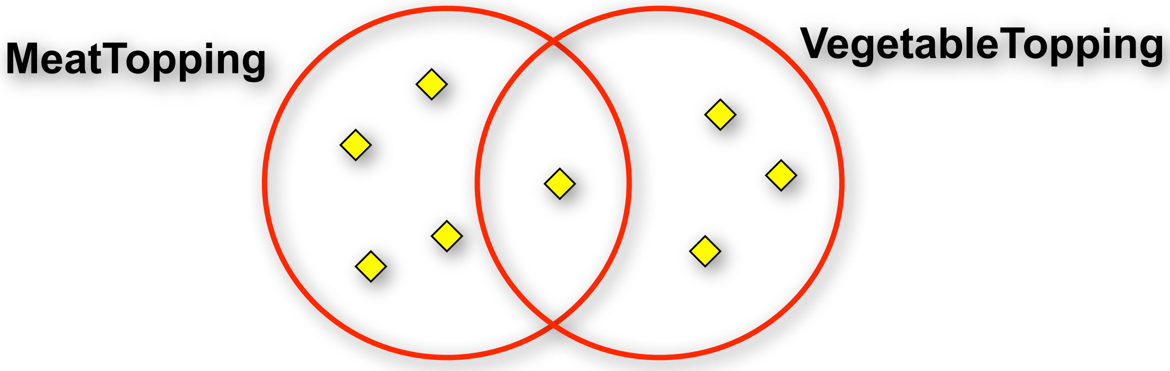
- We will explain the reasoner later
- Currently it shows us nothing
- We have something missing from the model

Disjoints



Disjoints

Regardless of where they exist in the hierarchy, OWL assumes that classes may overlap

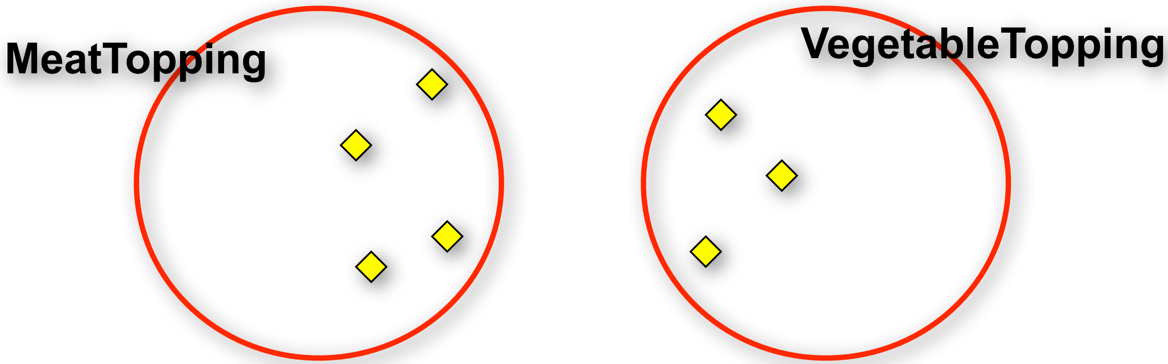


By default, an individual could be both a **MeatTopping** and a **VegetableTopping** at the same time

Disjoints

Stating that 2 classes are disjoint means

Nothing can be both a **MeatTopping** and a **VegetableTopping** at the same time

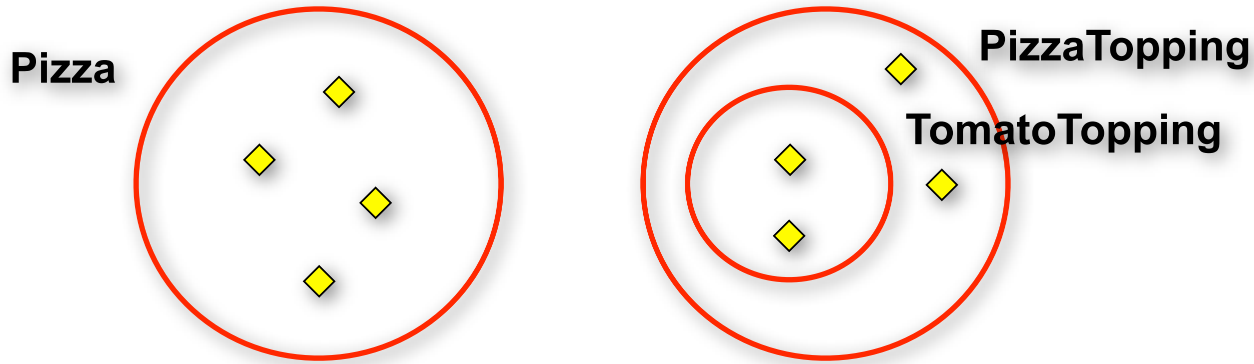


MeatTopping can never be a subclass of **VegetableTopping** (and vice-versa)

This can help us find errors

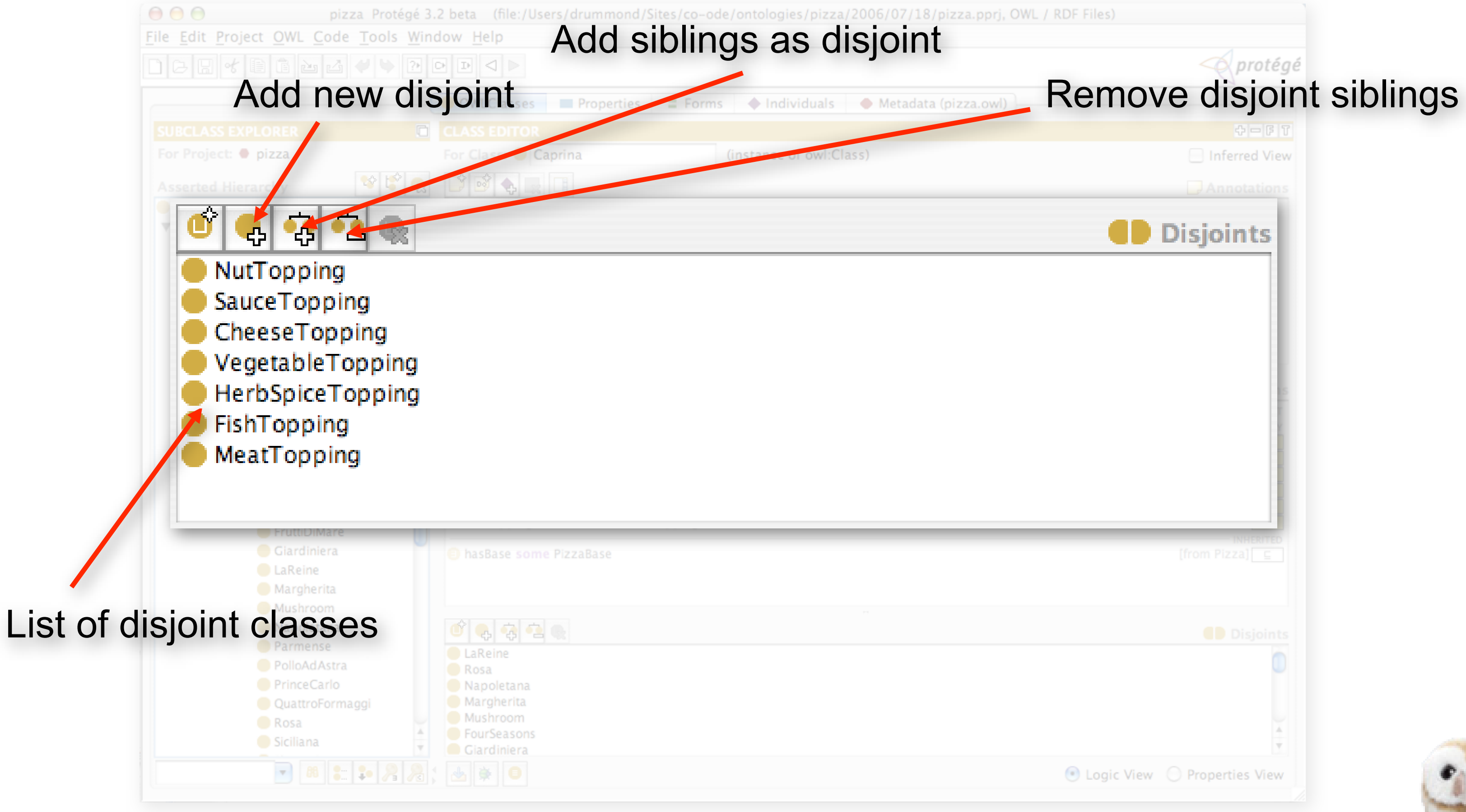
Disjoints

- Disjoints are **inherited** down the subsumption hierarchy



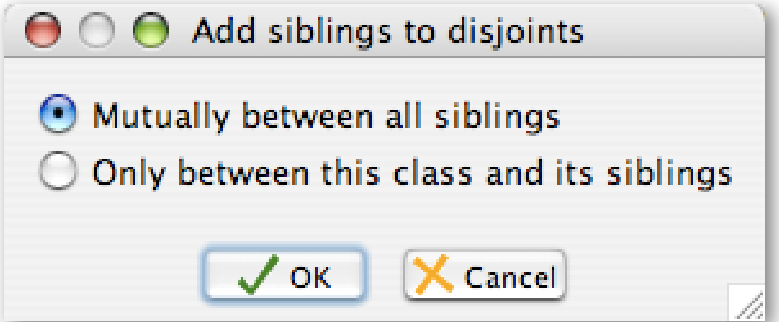
- Something that is a **TomatoTopping** cannot be a **Pizza** because its superclass, **PizzaTopping**, is disjoint from **Pizza**

ClassesTab: Disjoints Widget



Add Disjoints

- At each level in the ontology decide if the classes should be disjoint
- Use “Add all siblings” and choose “mutually” from the dialog
- You should now be able to select every class and see its siblings in the disjoints widget (if it has any)



Checking disjoints

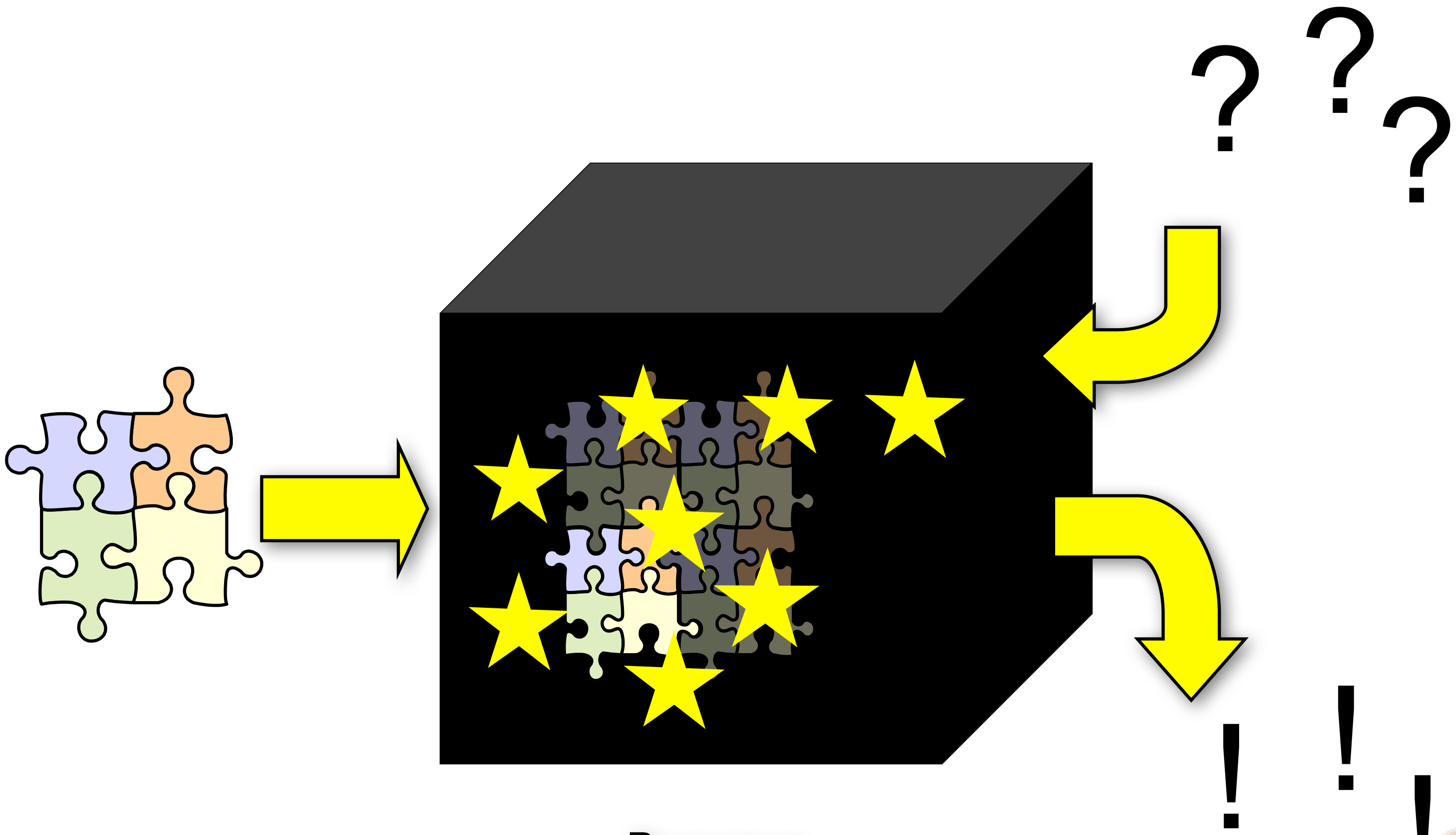
- Now that we've asserted some disjoints we have enough to start checking the consistency of our model
- Time for some magic...

Reasoners and Inference

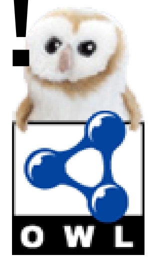
The University of Manchester

MANCHESTER 1824

Reasoners and Inference



Reasoner:
A clever (probably magic) black box designed by clever people
Best to let them worry about how they work



Reasoners and Inference: Basics

- Reasoners are used to **infer** information that is not explicitly contained within the ontology
- You may also hear them being referred to as **classifiers**
- Reasoners can be used at runtime in applications as a querying mechanism (esp useful for smaller ontologies)
- We will use one during development as an ontology "**compiler**"

Reasoners and Inference: Services

- Standard reasoner services are:
  - Consistency Checking
  - Subsumption Checking
  - Equivalence Checking
  - Instantiation Checking

Reasoners and Protégé

- Protégé-OWL supports the use of reasoners implementing the **DIG** interface
- Protégé-OWL can connect to reasoners that provide an http:// connection

FaCT++

Pellet

Racer

KAON2

MANCHESTER 1824

The University of Manchester

Connecting to a reasoner

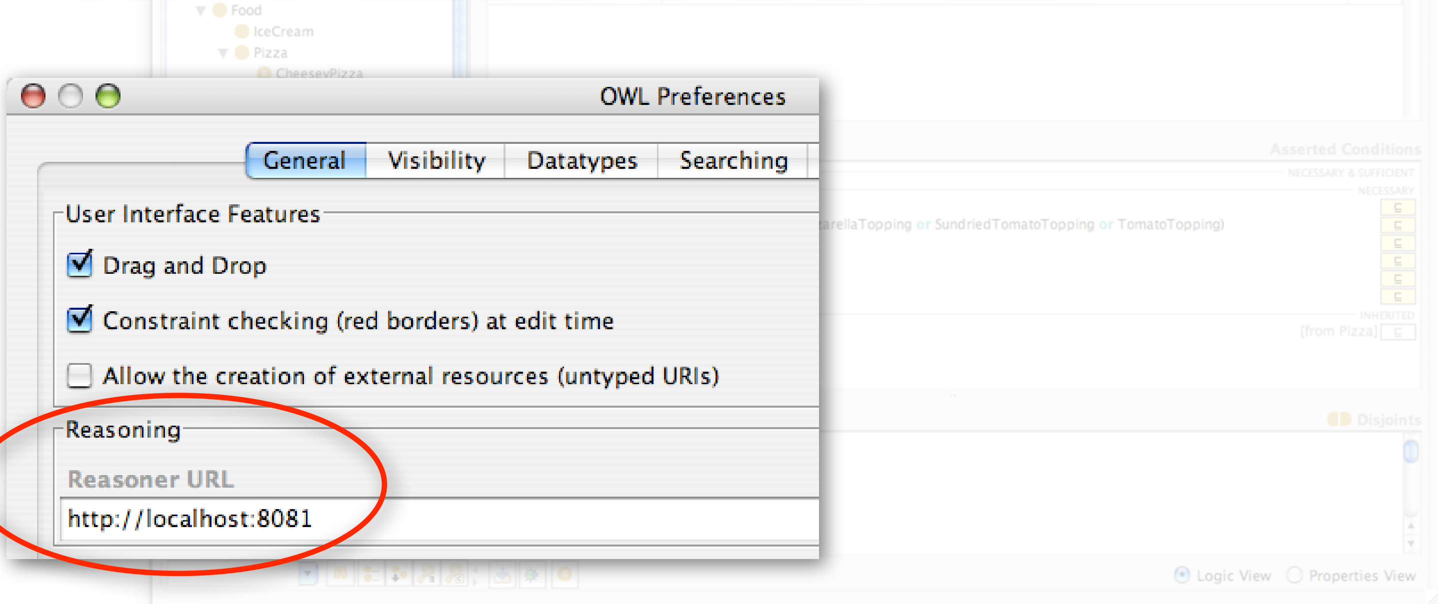
```
Terminal — java — 80x24
Last login: Thu Jul 20 15:24:45 on ttyp2
/Applications/Reasoners/pellet-1.3/pellet-dig.command; exit
Welcome to Darwin!
nick-drummonds-computer:~ drummond$ /Applications/Reasoners/pellet-1.3/pellet-di
g.command; exit

PelletDIGServer Version nightly-build-20060719 (July 19 2006)
Port: 8081
```

- Run a reasoner locally (or on a server)
- Note the address
- local typically http://localhost:<port_number>

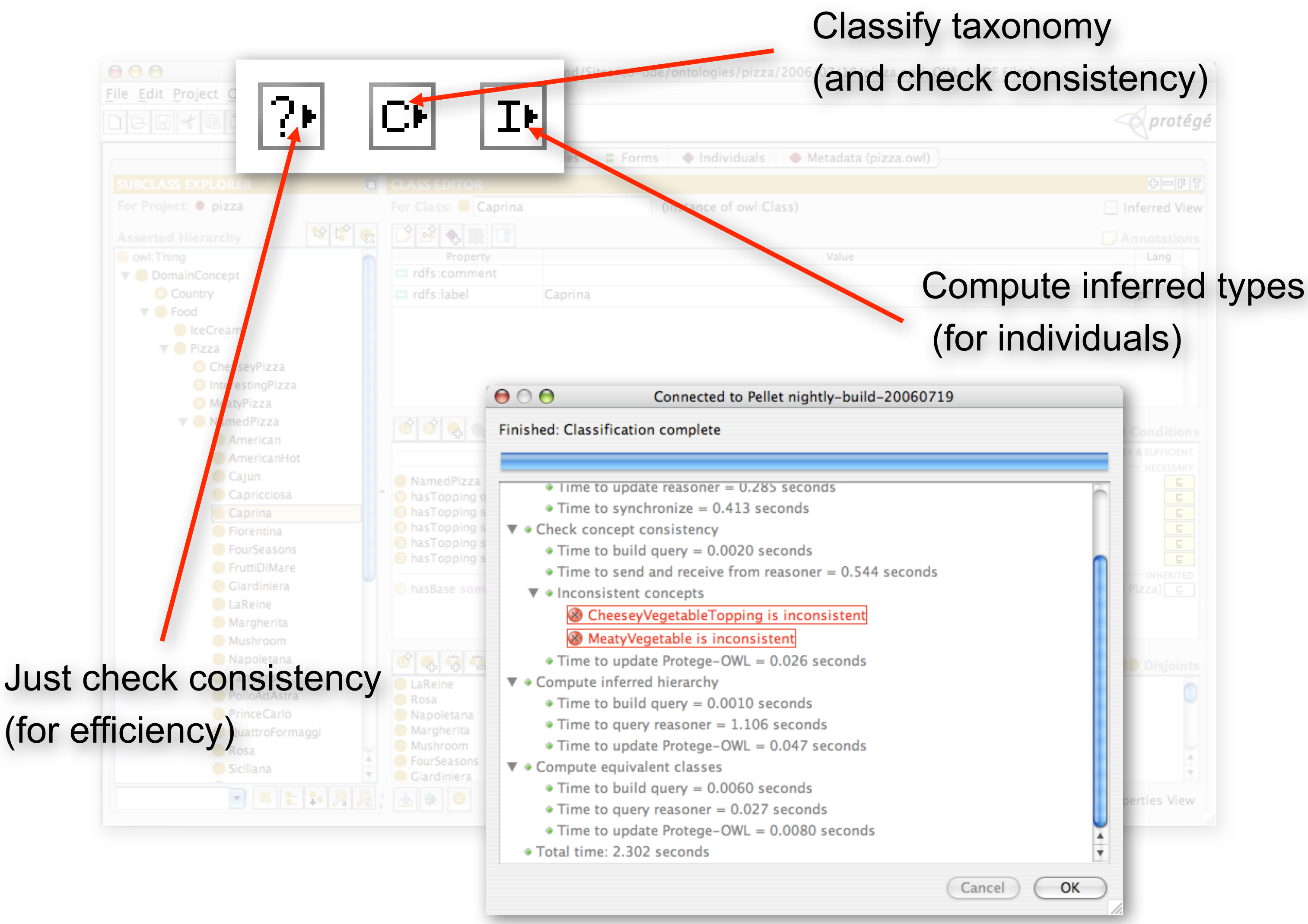
O W L

Connecting a Reasoner

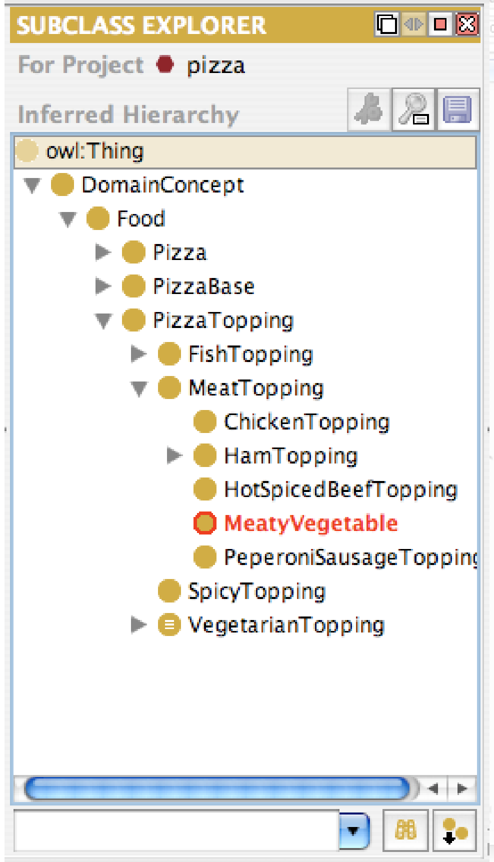
- In Protégé menu, go to:
  OWL → Preferences
- Set the reasoner URL to match



Accessing the Reasoner



Reasoning about our Pizzas



- When the reasoner has finished, you will see the **inferred hierarchy**
- Inferences are reported in the reasoner dialog and in a separate results window
- **inconsistent** classes turn red
- **moved** classes turn blue
- close this window

Why is MeatyVegetableTopping Inconsistent?

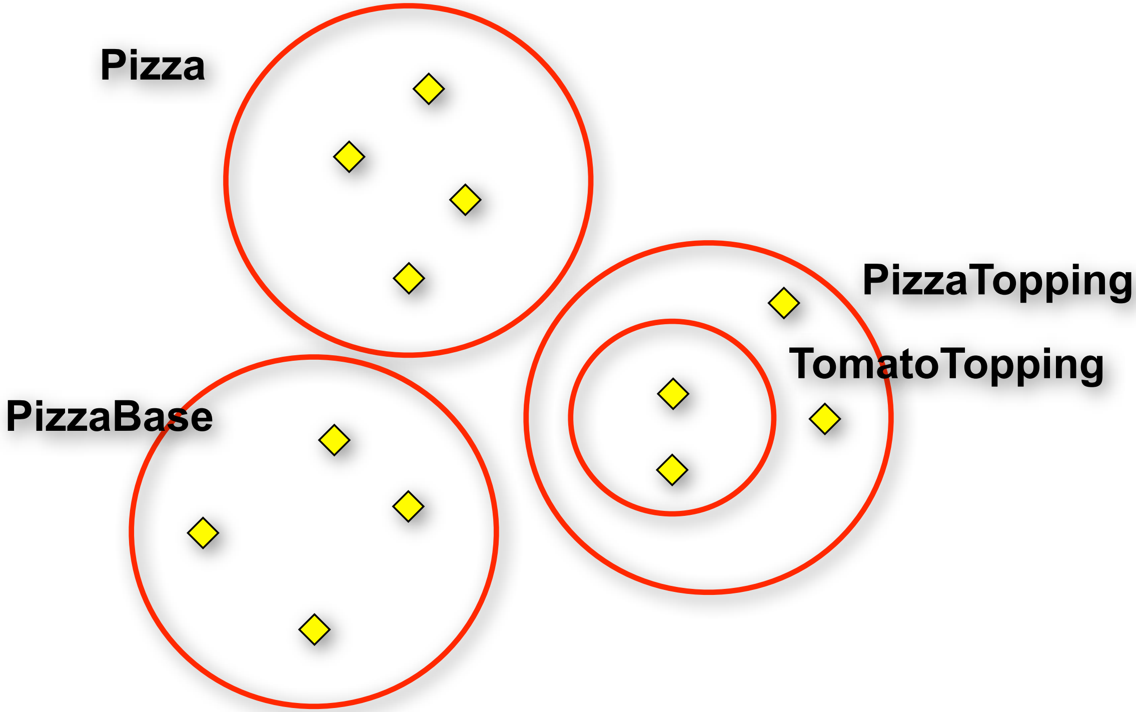
- **MeatyVegetableTopping** is a subclass of two classes we have stated are disjoint
- The disjoint means nothing can be a **MeatTopping** and a **VegetableTopping** at the same time
- This means that **MeatyVegetableTopping can never contain any individuals**
- The class is therefore **inconsistent**
- **This is what we expect!**
- It can be useful to create “probe” classes we expect to be inconsistent to “test” your model

In a tangle?

- You might have several inconsistent classes with multiple asserted parents
- We call this a **tangle**
- As we have seen, a class cannot have 2 disjoint parents – it will be inconsistent
- Removing disjoints between multiple parents by hand is tricky
- We will later show you some better ways to manage your tangle

What have we got?

- We’ve created a tangled graph of mostly disjoint classes

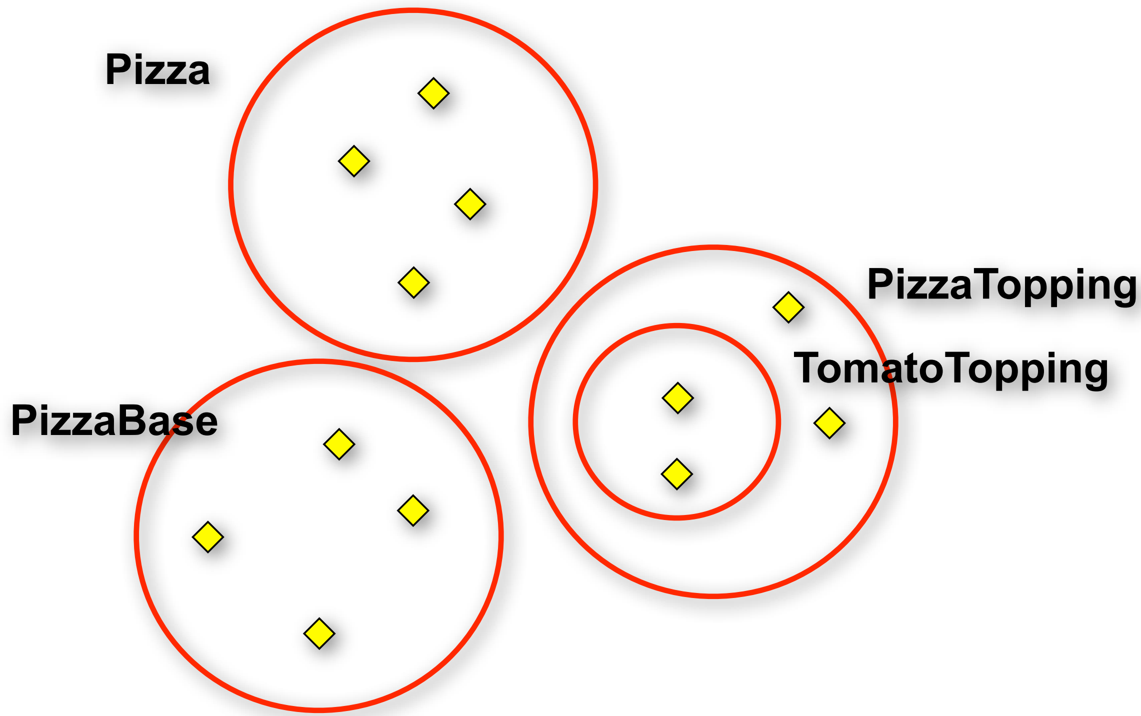


The University of Manchester

MANCHESTER 1824

What have we got?

- Although this could be very useful, its not massively exciting is it?

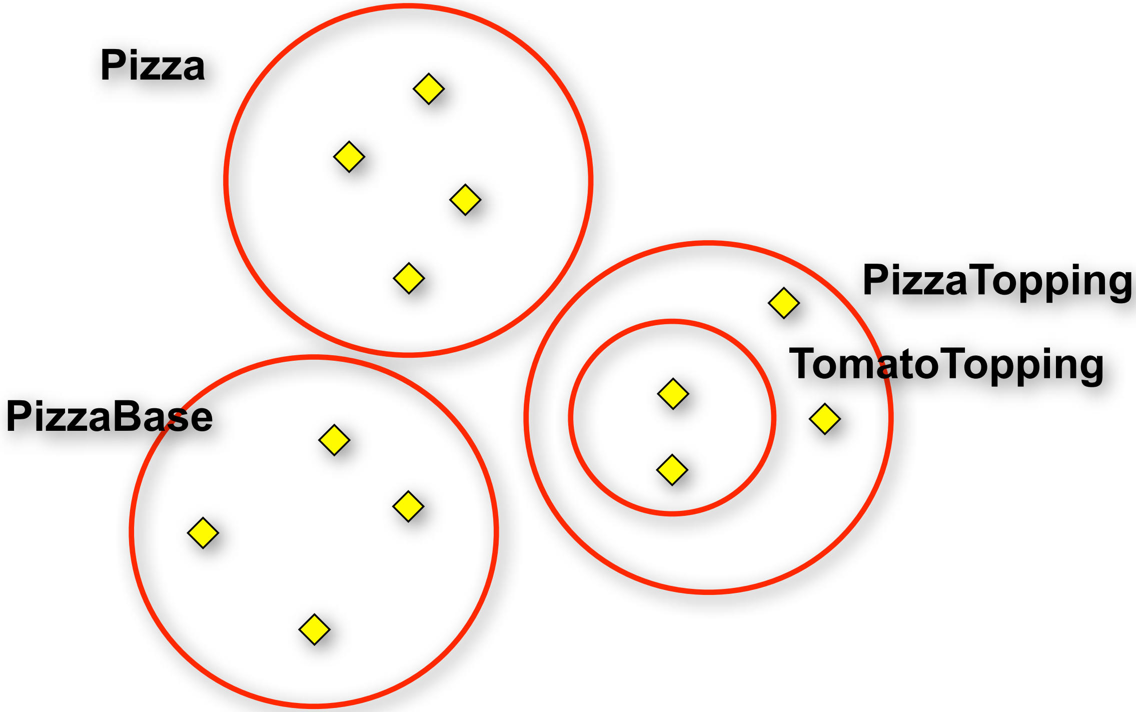


MANCHESTER 1824

The University of Manchester

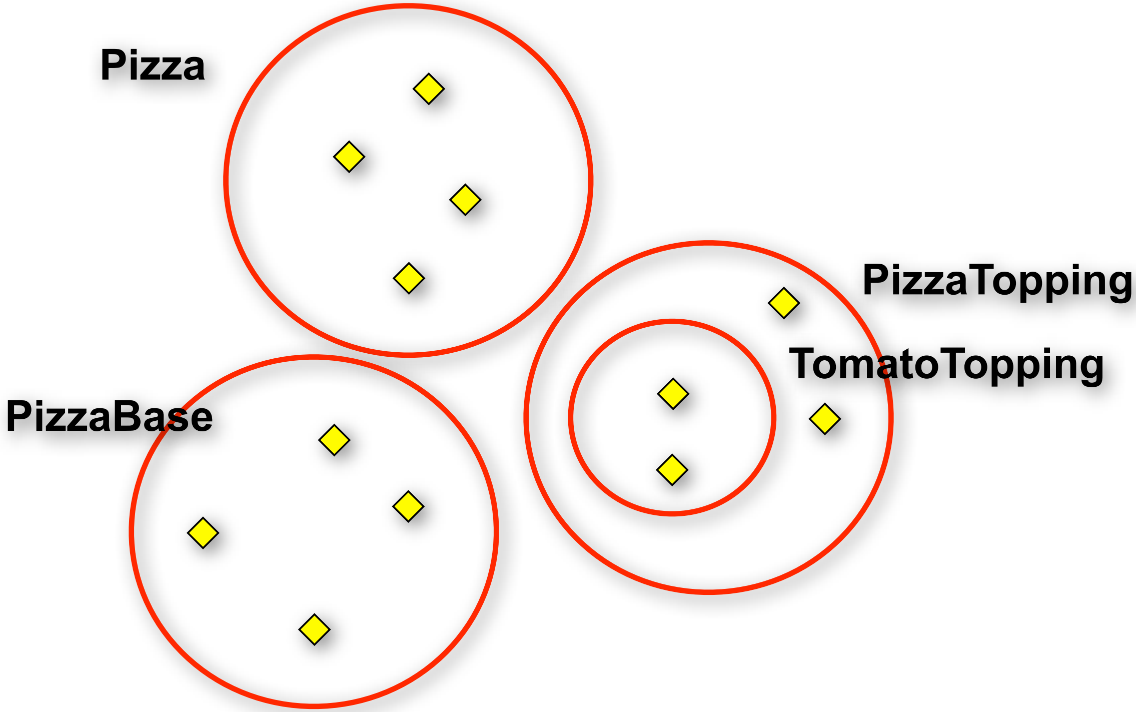
What have we got?

- Apart from “is kind of” (subsumption) and “is not kind of” (disjoint), we currently don’t have any other information of interest



What have we got?

- We want to say more about **Pizzas**
- eg All Pizzas must have a PizzaBase



Relationships in OWL

MANCHESTER 1824

The University of Manchester

Relationships in OWL

- In OWL-DL, relationships can only be formed **between Individuals** or between an Individual and a data value
  (In OWL-Full, Classes can be related, but this cannot be reasoned with)
- Relationships are formed **along Properties**
- We can restrict how these Properties are used:
  - **Globally** – by stating things about the Property itself
  - Or **locally** – by restricting their use for a given Class

OWL Properties

isFromSpecies

hasLimbs

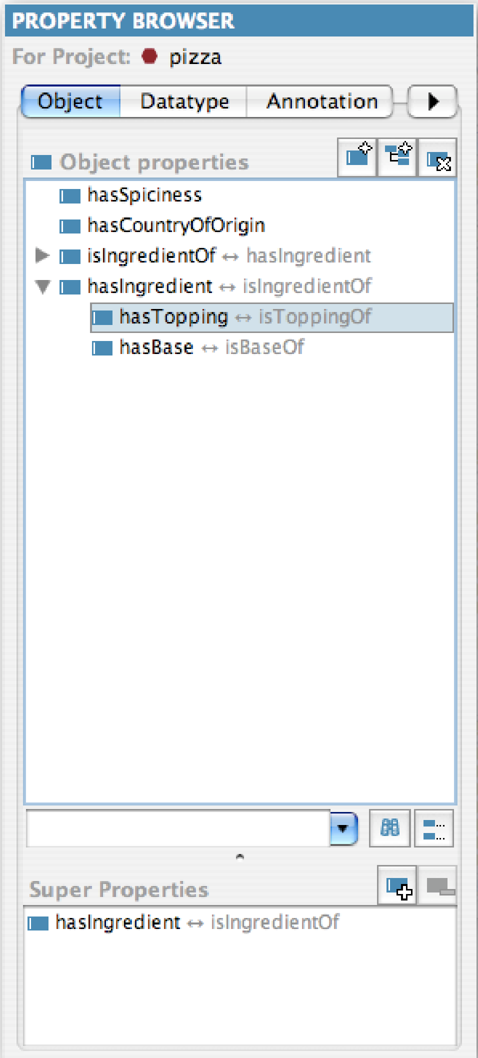
isCoveredWith

hasCuteness

MANCHESTER 1824

The University of Manchester

Property Browser

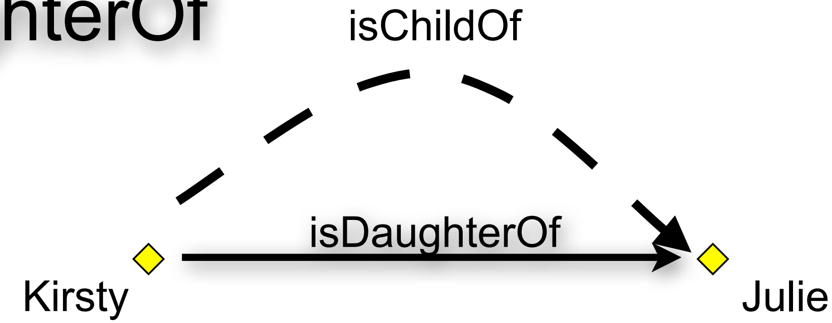


- Object Property – relate Individuals
- Datatype Property – relate Individuals to data (int, string, float etc)
- Annotation Property – for attaching metadata to classes, individuals or properties
- Note that Properties can be in a hierarchy

Subproperties

- What does subproperty mean?

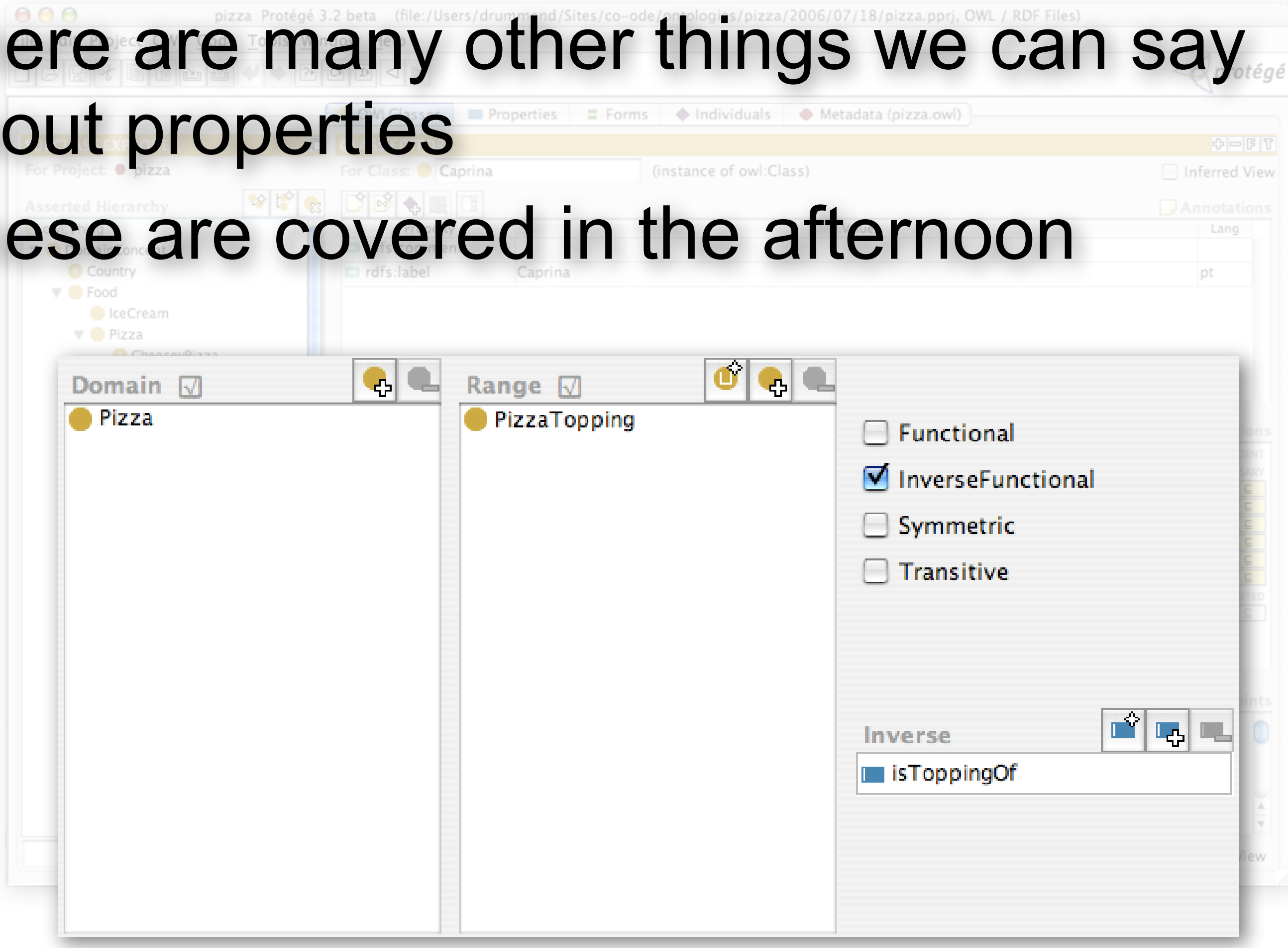
  - isChildOf
    - isDaughterOf



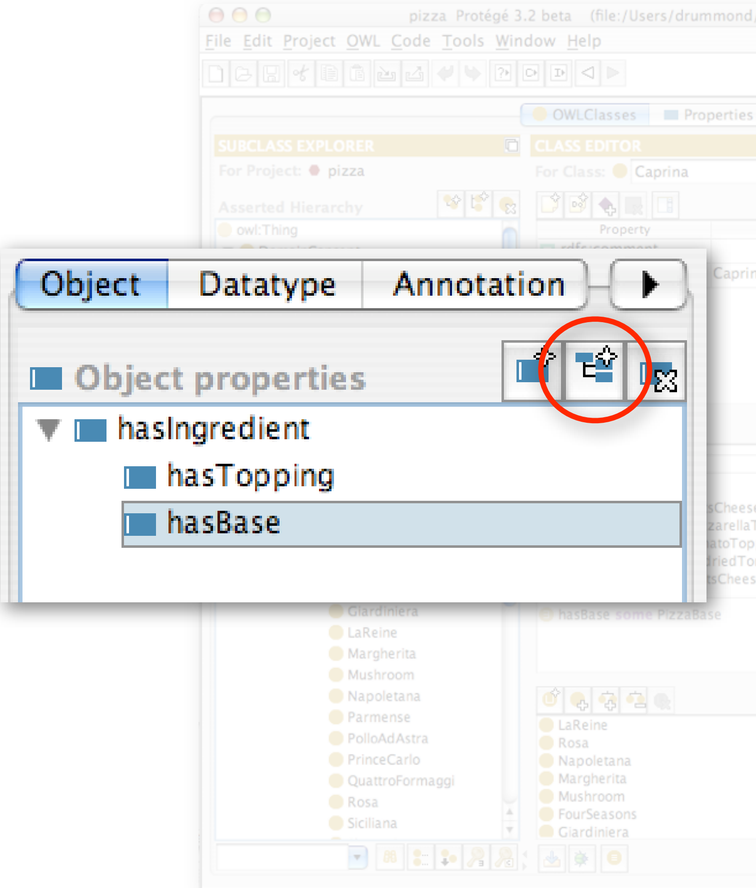
- You cannot mix property types in the tree
  ie Object properties cannot be subproperties of Datatype properties and vice-versa

Property Features

- There are many other things we can say about properties
- These are covered in the afternoon



Creating Object Properties



- Switch to the properties Tab
- Make sure the object property hierarchy is showing
- Create the property hierarchy shown
- We will normally use the subproperties and infer the superproperties

Using Properties

- We now have some properties we want to use to describe **Pizza**s
- We can just use properties directly to relate individual pizzas
- But, we're not creating individuals
- Instead, we are going to make statements about all members of the **Pizza** Class

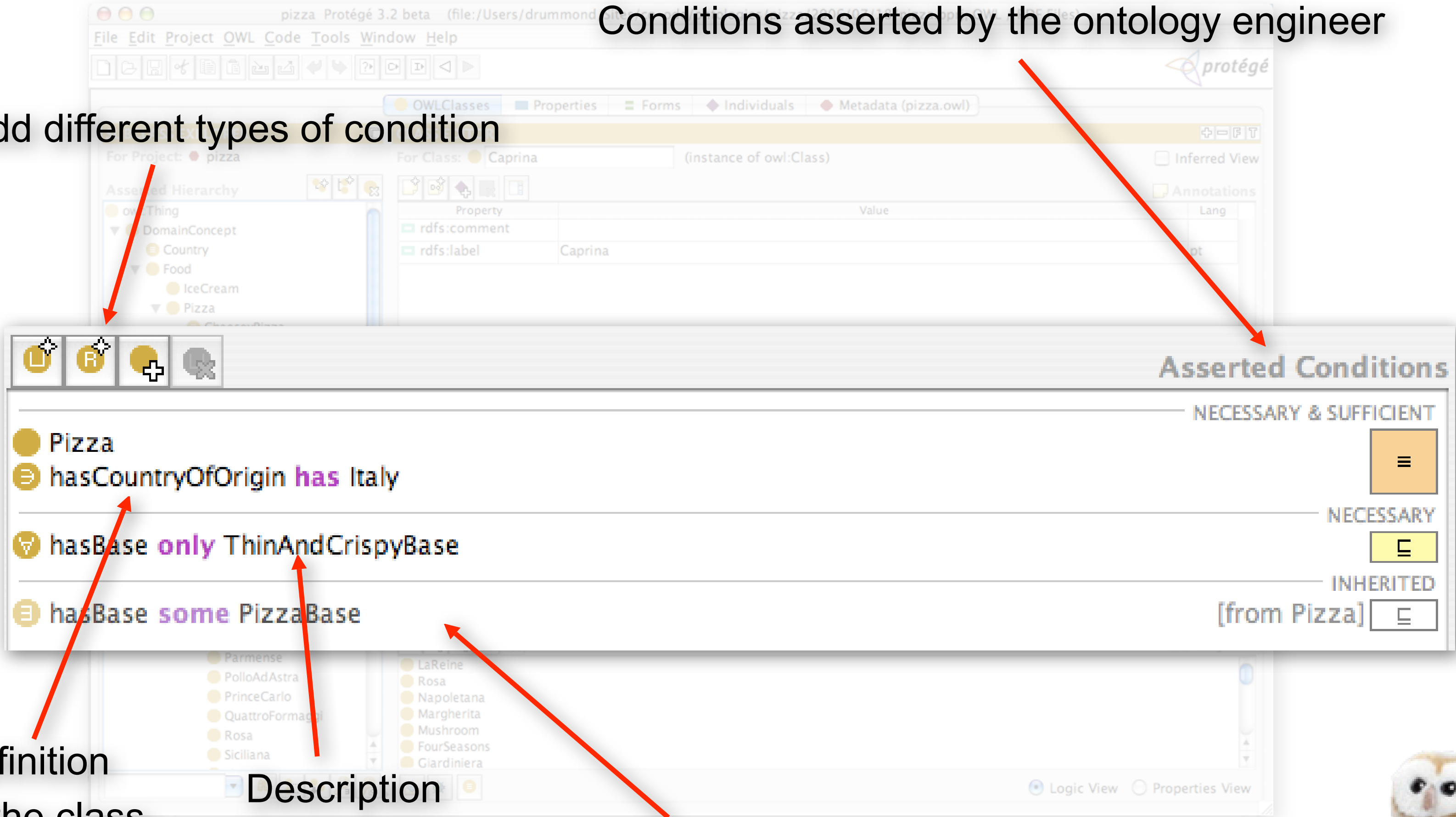
Using Properties with Classes

- To do this, we must go back to the **Pizza** class and add some further information
- This comes in the form of **Restrictions**
- We create Restrictions in the Conditions widget
- Conditions can be any kind of Class – you have already added Named superclasses in the Conditions Widget. **Restrictions are a type of Anonymous Class**

Conditions Widget

Conditions asserted by the ontology engineer

Add different types of condition



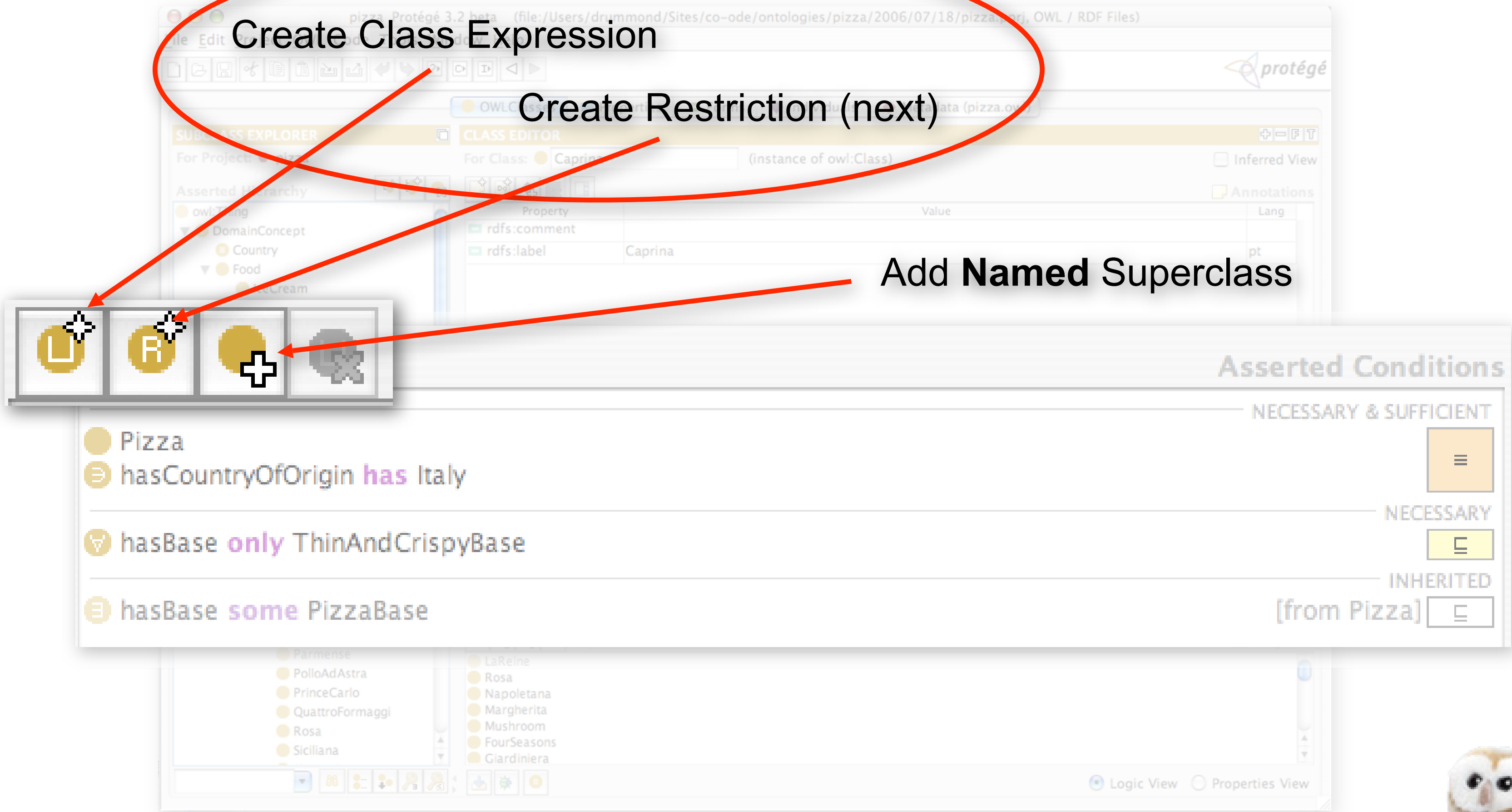
Definition of the class (later)

Description of the class

Conditions inherited from superclasses

Conditions Widget

Logical (Anonymous) Classes



Creating Restrictions

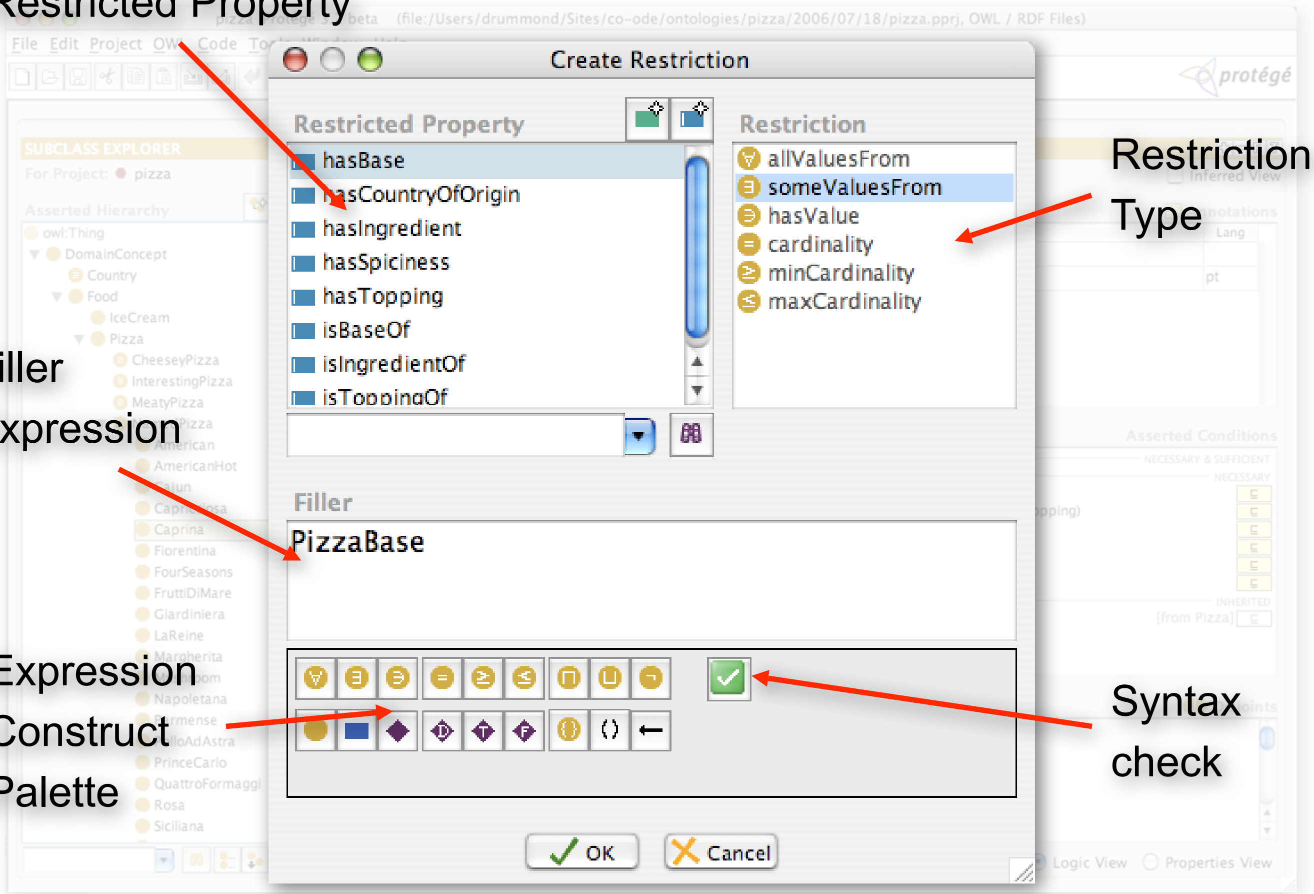
Restricted Property

Restriction Type

Filler Expression

Expression Construct Palette

Syntax check

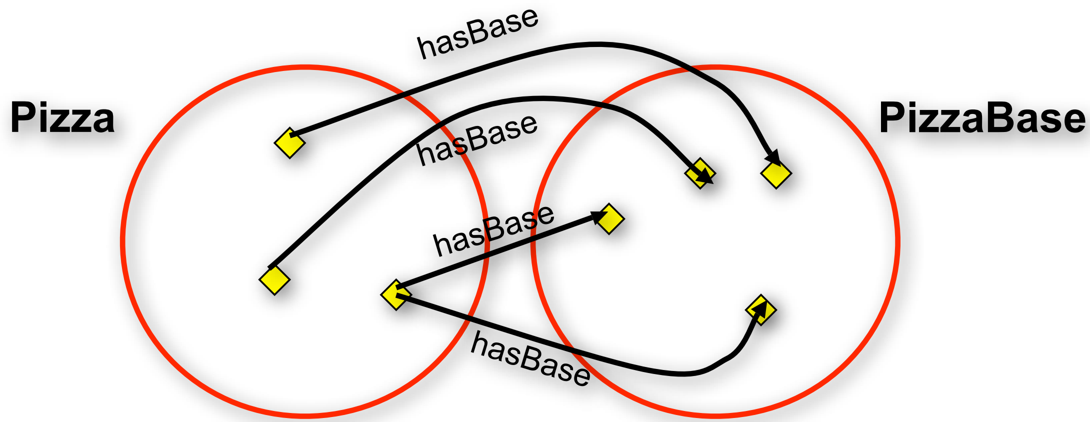


MANCHESTER 1824

The University of Manchester

What does this mean?

- restriction: **hasBase some PizzaBase** on Class **Pizza** as a necessary condition



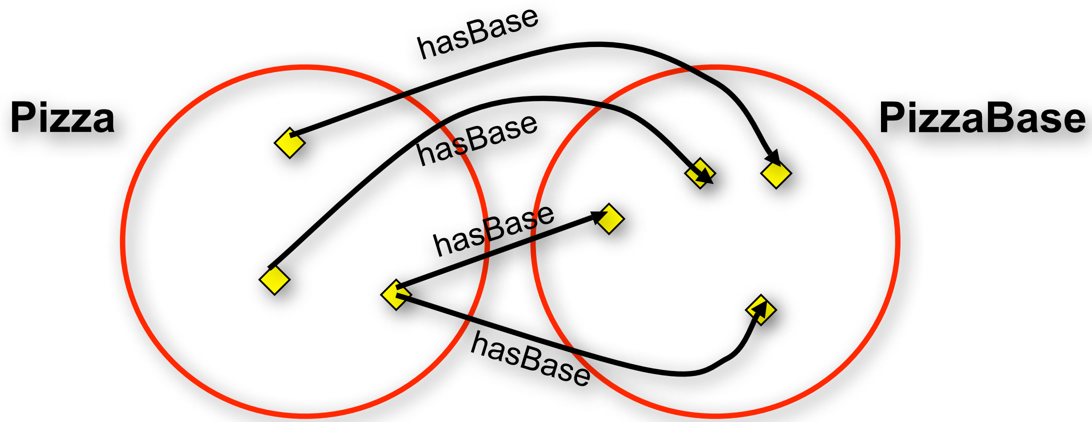
- "If an individual is a member of this class, it is necessary that it has at least one hasBase relationship with an individual from the class **PizzaBase**"

MANCHESTER 1824

The University of Manchester

What does this mean?

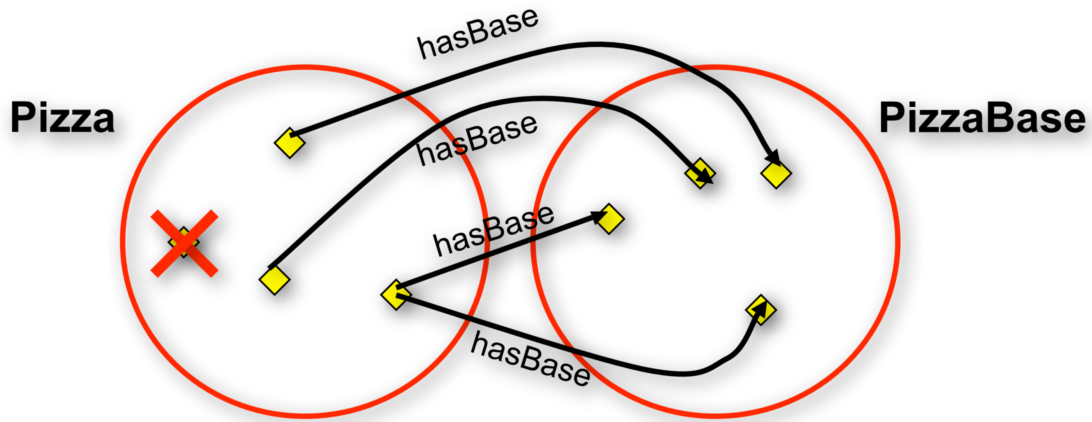
► restriction: **hasBase some PizzaBase** on Class **Pizza** as a necessary condition



► "Every individual of the **Pizza** class must have at least one base from the class **PizzaBase**"

What does this mean?

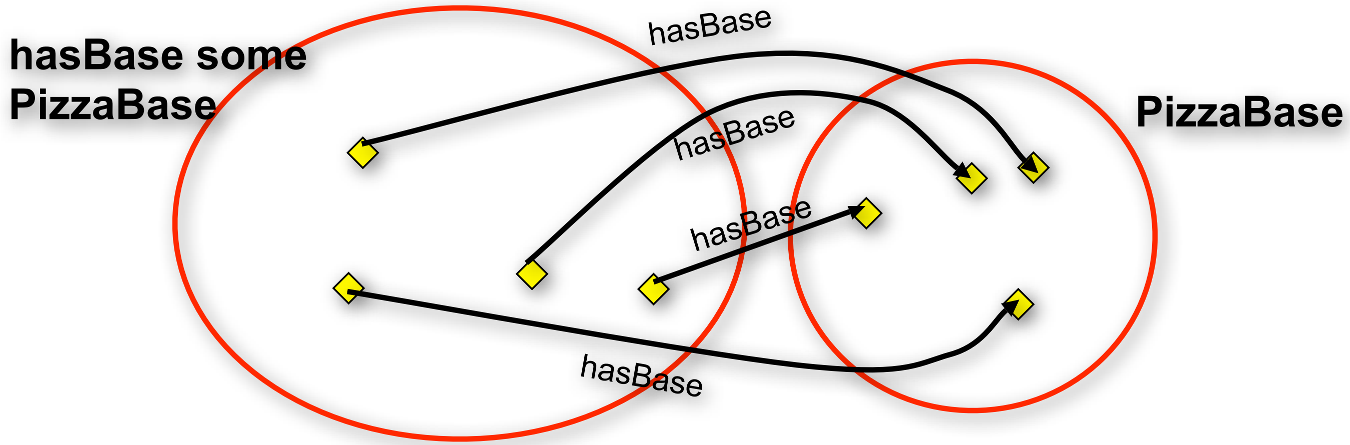
- restriction: **hasBase some PizzaBase** on Class **Pizza** as a necessary condition



- "There can be **no individual**, that is a member of this class, that **does not have at least one** hasBase relationship with an individual from the class **PizzaBase**"

Why? Restrictions are Classes

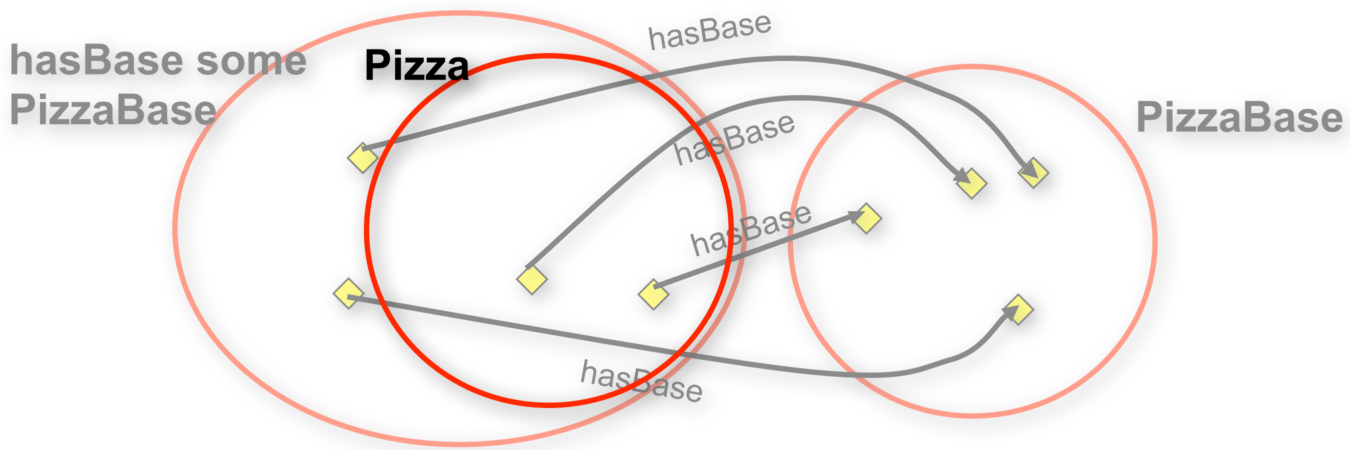
- restriction: **hasBase some PizzaBase** on Class **Pizza** as a necessary condition



- Restrictions and Class Expressions are anonymous **classes**
- they contain the set of all individuals that satisfy the condition

Why? Necessary Conditions are Superclasses

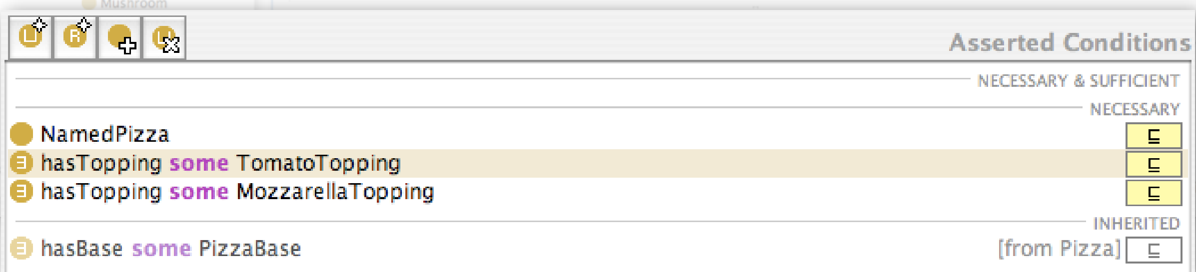
- ▶ restriction: **hasBase some PizzaBase** on Class **Pizza** as a necessary condition



- ▶ Each necessary condition is a superclass
- ▶ Pizza is a subclass of all the things that have a pizza base
- ▶ All pizzas must have a pizza base

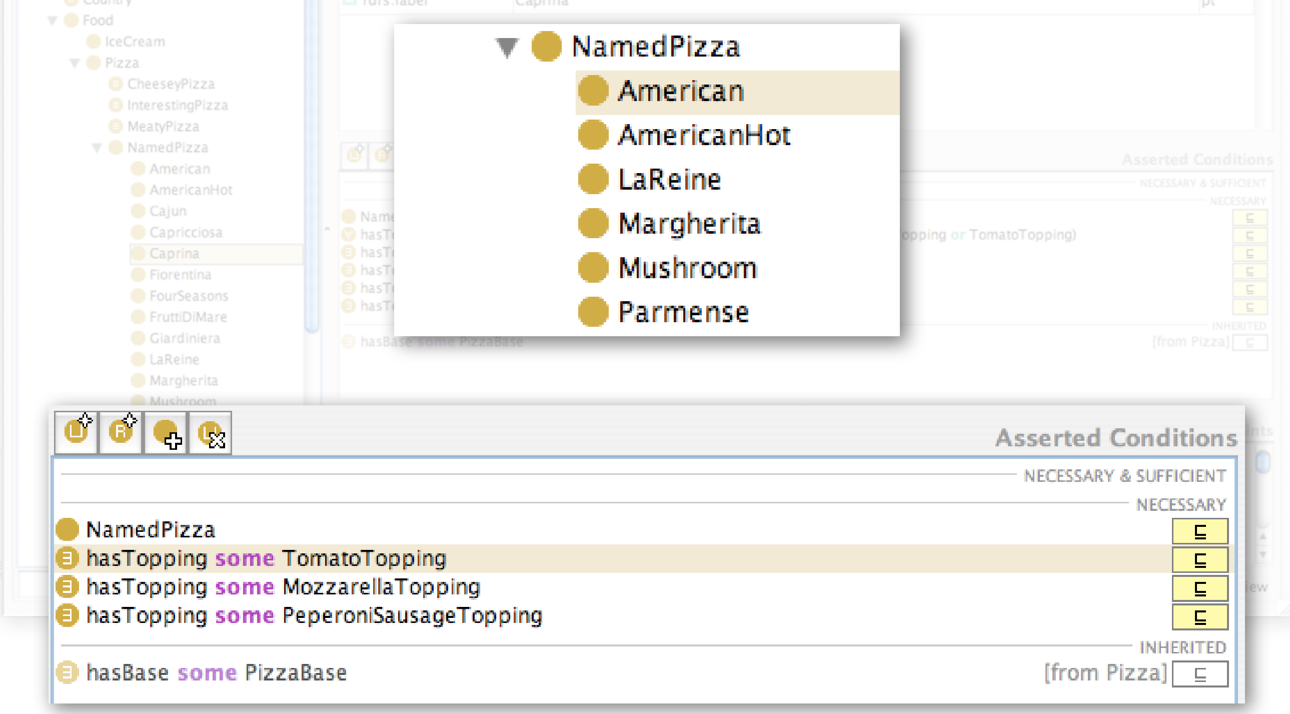
Create your first pizza

- Create a subclass of Pizza, **NamedPizza**
- Create a named pizza, **Margherita**
- Add 2 restrictions on Margherita:
  hasTopping some MozzarellaTopping
  hasTopping some TomatoTopping
  "All Margheritas have at least one topping that is Mozzarella and one that is Tomato"



Create more pizzas

- Create a couple more named pizzas and add the appropriate toppings using existential restrictions

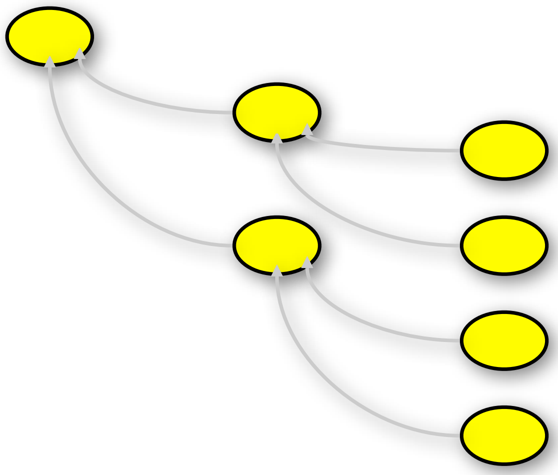


Restriction Types

$\exists$	**Existential, someValuesFrom**	**"some", "at least one"**
$\forall$	Universal, allValuesFrom	"only"
$\ni$	hasValue	"equals x"
$=$	Cardinality	"exactly n"
$\leq$	Max Cardinality	"at most n"
$\geq$	Min Cardinality	"at least n"

Single Asserted Superclasses

- All classes in our ontology so far are **Primitive**
- Primitive Class = only Necessary Conditions
- We condone building a disjoint tree of primitive classes
- This is also known as a **Primitive Skeleton**



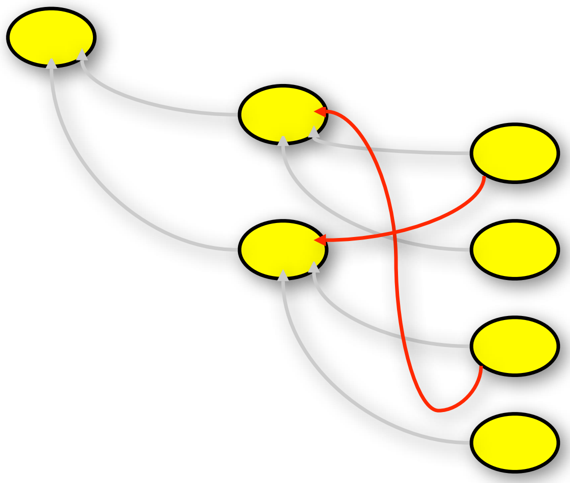
Polyhierarchies

- In the afternoon session you will create a VegetarianPizza
- Some of our existing Pizzas could be types of VegetarianPizza, SpicyPizza and/or CheeseyPizza
- We **need** to be able to give them **multiple parents** in a principled way
- We could just assert multiple parents like we did with MeatyVegetableTopping (without disjoints)

BUT…

Multiple Asserted Superclasses

- We lose some encapsulation of knowledge
  - **Why** this class is a subclass of that one
- Adding a new abstraction becomes difficult because all subclasses may need to be updated
- Extracting from a graph is harder than from a tree

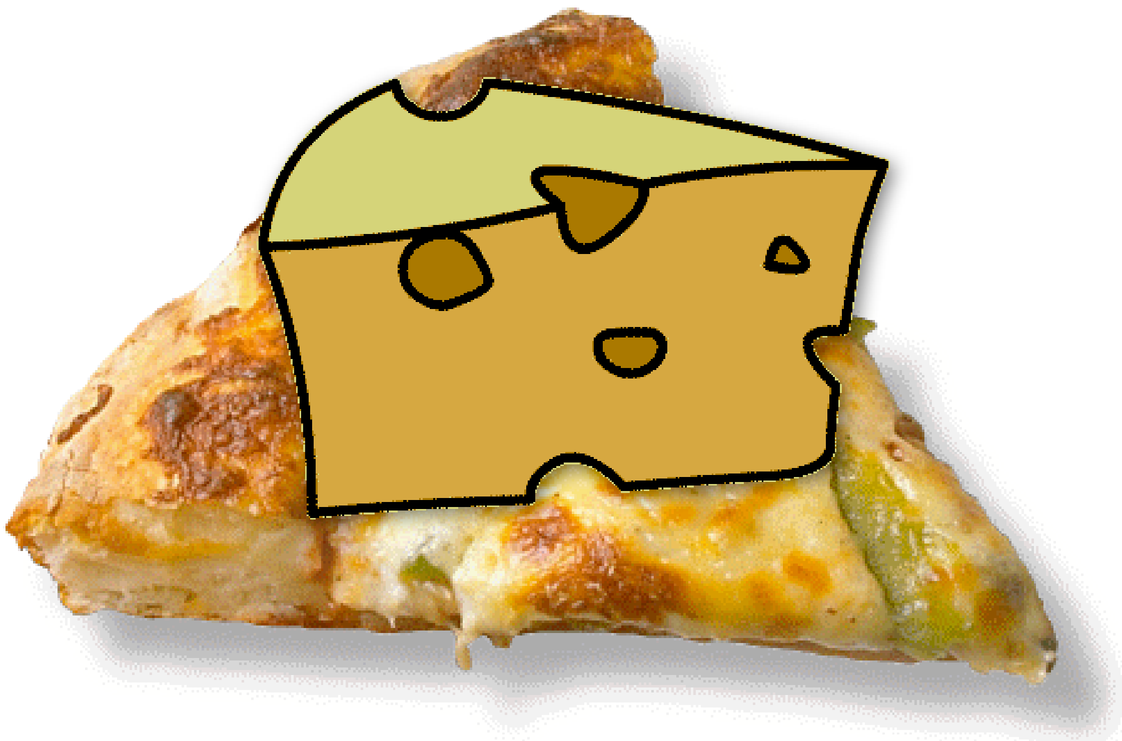


let the reasoner do it!

The University of Manchester

MANCHESTER 1824

Defined Classes



CheeseyPizza

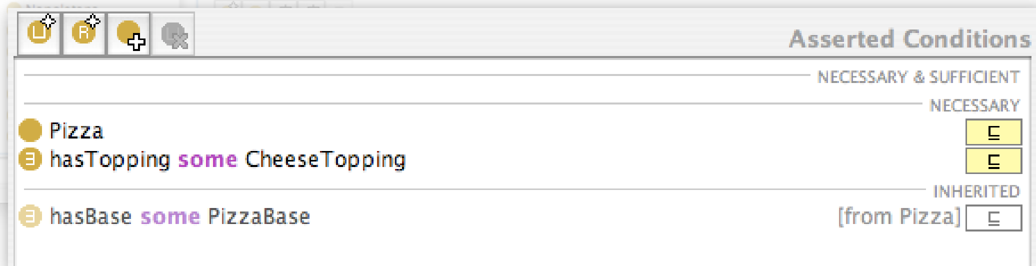
- "A CheeseyPizza is **any** pizza that has some cheese on it"
- We would expect then, that some pizzas might be **both** named pizzas and cheesey pizzas (among other things later on)
- We can use the reasoner to help us produce this polyhierarchy without having to **assert** multiple parents and so avoid a **tangle**

The University of Manchester

MANCHESTER 1824

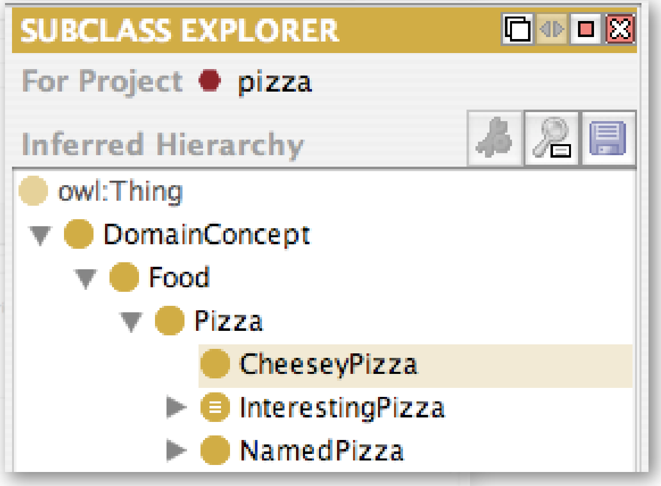
Creating a CheeseyPizza

- We often create primitive classes and then migrate them to defined classes
- All of our defined pizzas will be direct subclasses of **Pizza**
- Create a **CheesyPizza** Class (do not make it disjoint)
  - add a restriction:
    "Every CheeseyPizza must have at least one CheeseTopping"



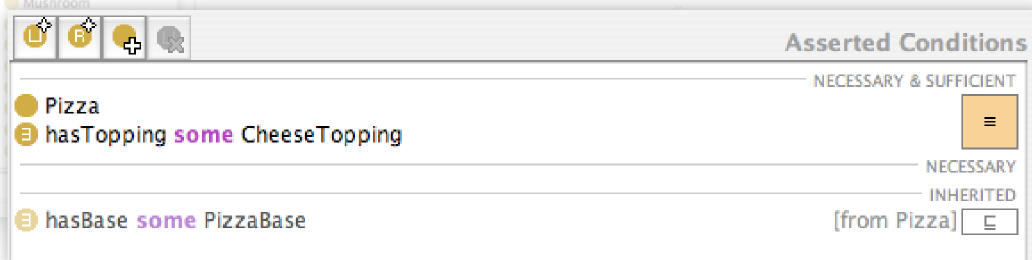
Classifying Primitive Classes

- Classifying this ontology does nothing
- Our definition is **“Every CheeseyPizza must have at least one CheeseTopping”**
- What we want is **“A CheeseyPizza is any pizza that has some cheese on it”**



Creating a Defined Class

- Lets move the conditions we've created
- There is a useful button for turning this into a defined class at the bottom of the class editor
- Notice the conditions are now in the "Necessary & Sufficient" block

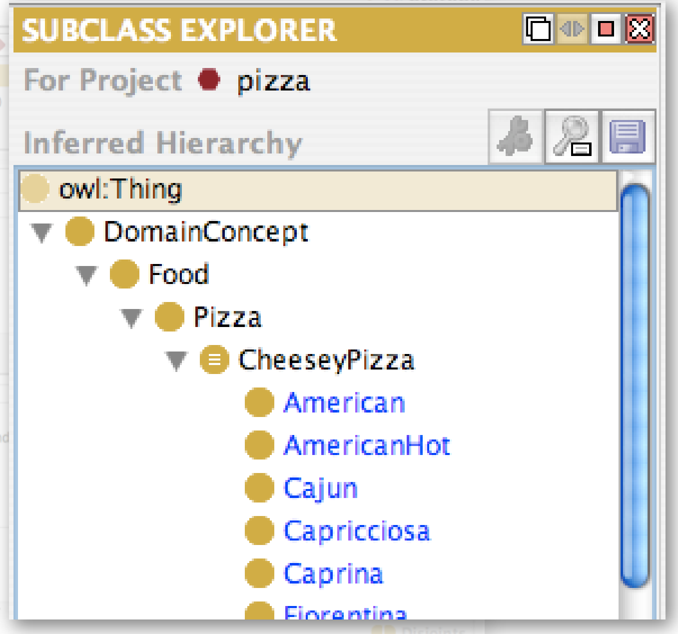


MANCHESTER 1824

The University of Manchester

Classifying a Defined Class

- The inferred hierarchy now shows many (blue) subclasses of **CheeseyPizza**
- The reasoner has been able to infer that any **Pizza** that has at least one topping from **CheeseTopping** is a **CheeseyPizza**



Why? Necessary & Sufficient Conditions

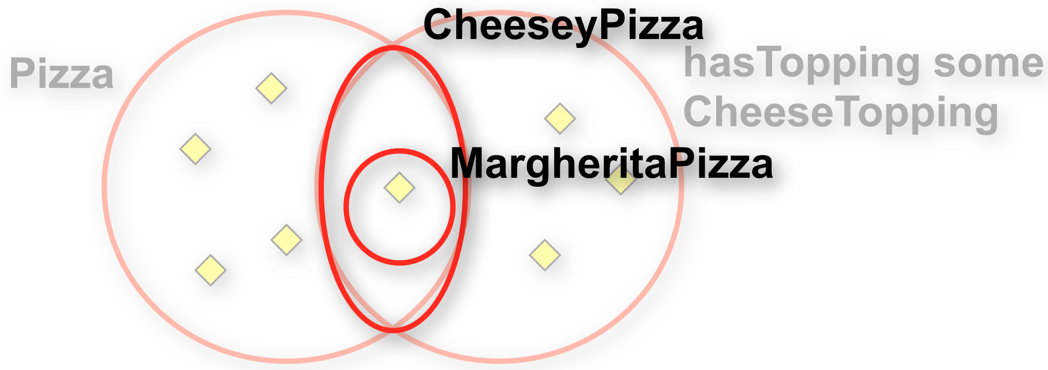
- Each set of necessary & sufficient conditions is an Equivalent Class

- **CheeseyPizza** is equivalent to the intersection of **Pizza** and **hasTopping some CheeseTopping**

The University of Manchester

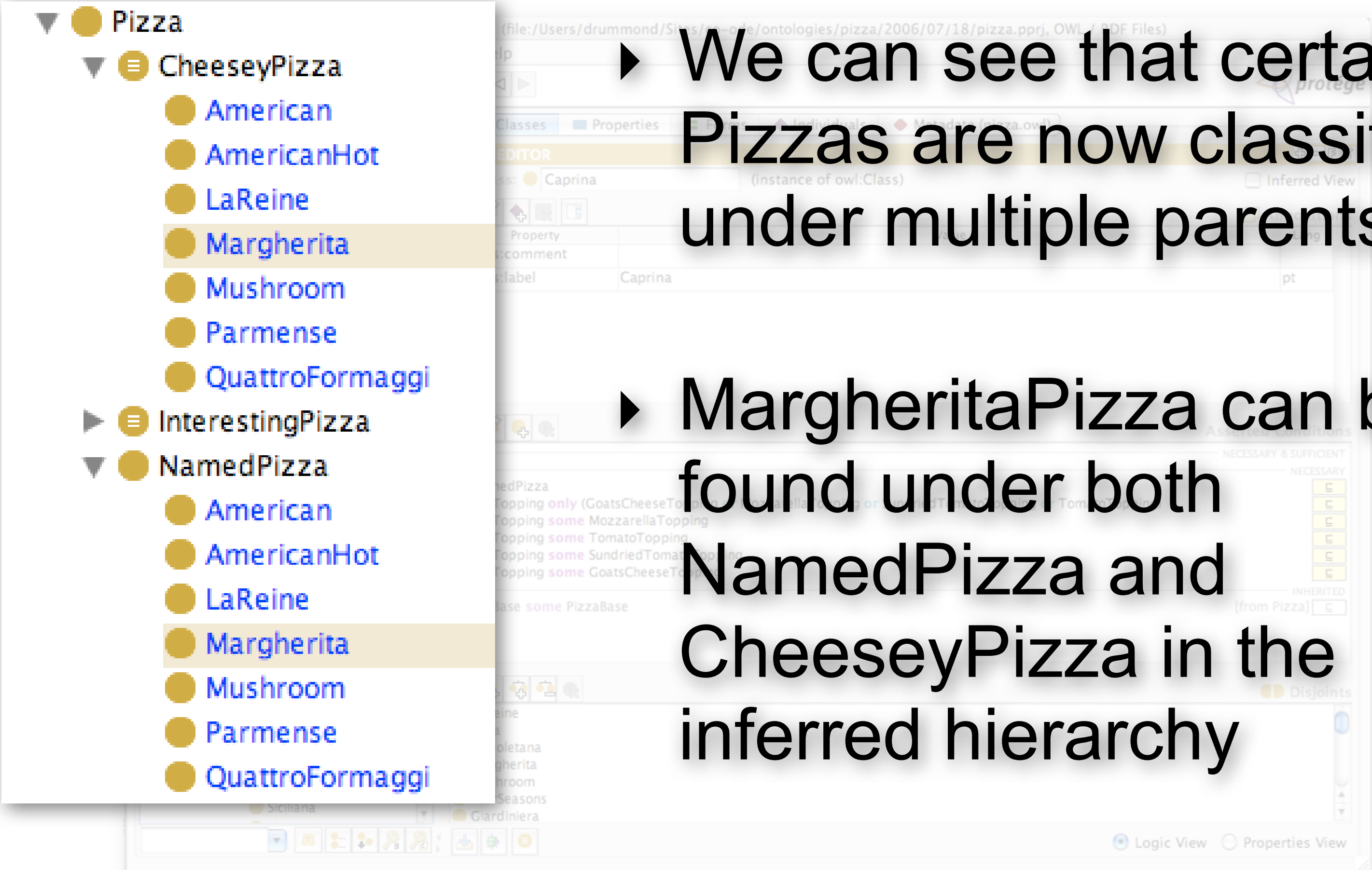
Why? Necessary & Sufficient Conditions

- Each set of necessary & sufficient conditions is an Equivalent Class



- Classes, **all** of whose individuals fit this definition are found to be subclasses of **CheeseyPizza**

Untangling



- We can see that certain Pizzas are now classified under multiple parents
- MargheritaPizza can be found under both NamedPizza and CheeseyPizza in the inferred hierarchy

Untangling

- However, our unclassified version of the ontology is a simple tree, which is much easier to maintain
- We’ve now got a polyhierarchy without asserting multiple superclass relationships
- Plus, we also know **why** certain pizzas have been classified as CheeseyPizzas

MANCHESTER 1824

The University of Manchester

Untangling

- We don’t currently have many kinds of primitive pizza but its easy to see that if we had, it would have been a substantial task to assert **CheeseyPizza** as a parent of lots, if not all, of them
- And then do it all over again for other defined classes like **MeatyPizza** or whatever

Mission Successful!

OWL

The University of Manchester

MANCHESTER 1824

Summary

You should now be able to:

- identify components of the Protégé-OWL Interface
- create a hierarchy of Primitive Classes
- create Properties
- create some basic Restrictions on a Class using Existential qualifiers
- create a simple Defined Class
- and...

Summary

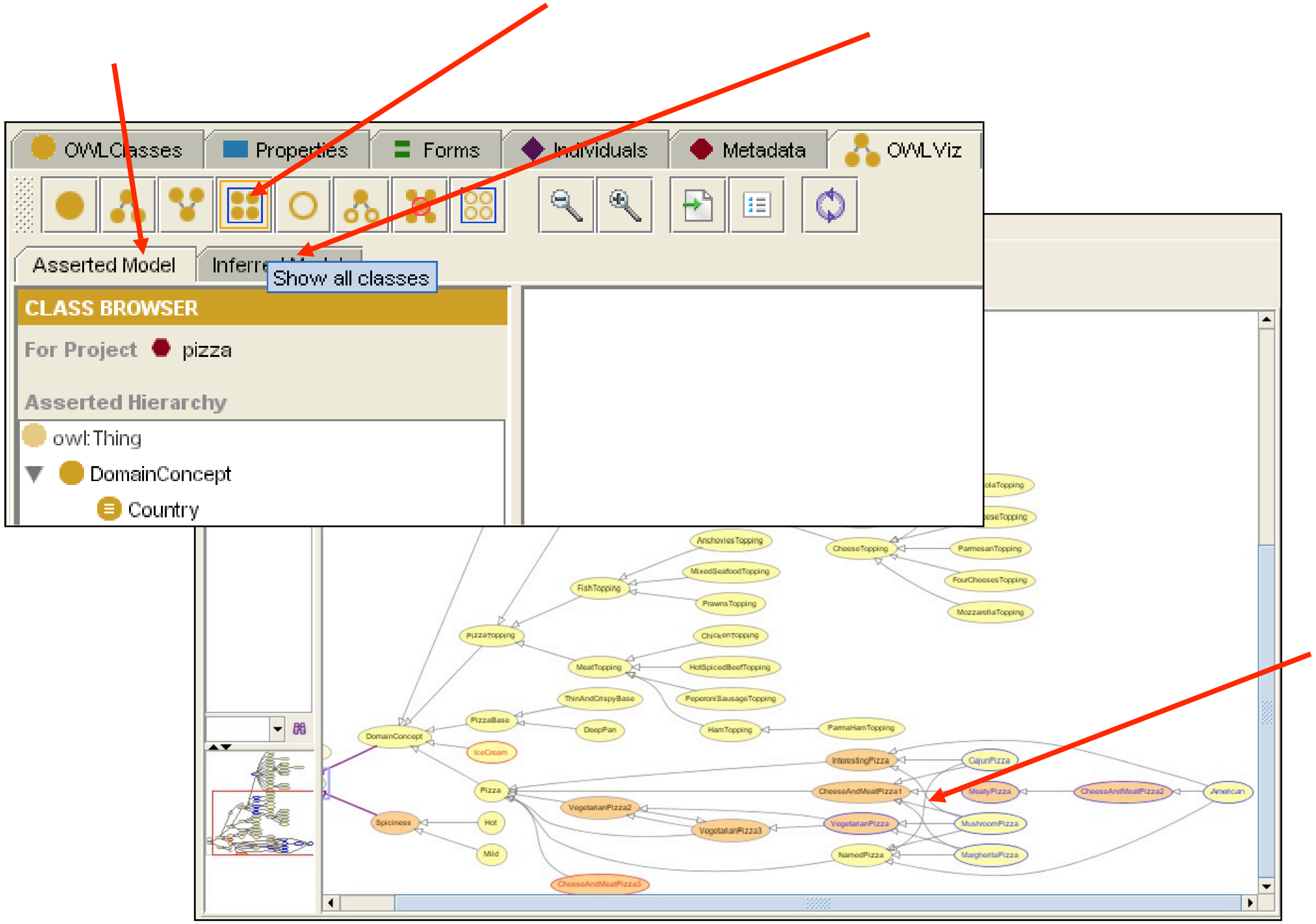
You should now be able to:

- go for at least a week without wanting to see

Additional Material

- OWLViz

OWLViz Tab


OWLClasses
Properties
Forms
Individuals
Metadata
OWLViz
Asserted Model
Show all classes
CLASS BROWSER
For Project
pizza
Asserted Hierarchy
owl:Thing
DomainConcept
Country