Building OWL Ontologies with Protege

CS 431 – April 9, 2008

Carl Lagoze – Cornell University

The University of Manchester

Parts extracted with permission from:

A Practical Introduction to Ontologies & OWL

Session 1: Primitive Classes in OWL

Nick Drummond & Matthew Horridge

The University of Manchester

Parts extracted with permission from:

A Practical Introduction to Ontologies & OWL

Session 2: Defined Classes and Additional Modelling Constructs in OWL

Nick Drummond & Matthew Horridge

Resources

- Protégé - http://protege.stanford.edu/
  - General open-source ontology modeling system with OWL plug-in
  - Use the 3.4 beta
  - Multiple plug-ins are available, download full
- Pellet - http://pellet.owldl.com/
  - Open-source OWL DL reasoner
  - Server-based (DIG protocol port 8081)
  - Integrates with Protégé-OWL

Resources (2)

- CO-ODE Resources - http://www.co-ode.org/
  - Protégé OWL Tutorial - http://www.co-ode.org/resources/tutorials/protege-owl-tutorial.php
  - Tutorial: A Practical Introduction to Ontologies & OWL - http://www.co-ode.org/resources/tutorials/intro/

What is an *Ontology*?

- A formal specification of conceptualization shared in a community
- Vocabulary for defining a set of things that exist in a world view
- Formalization allows communication across application systems and extension
- Global vs. Domain-specific

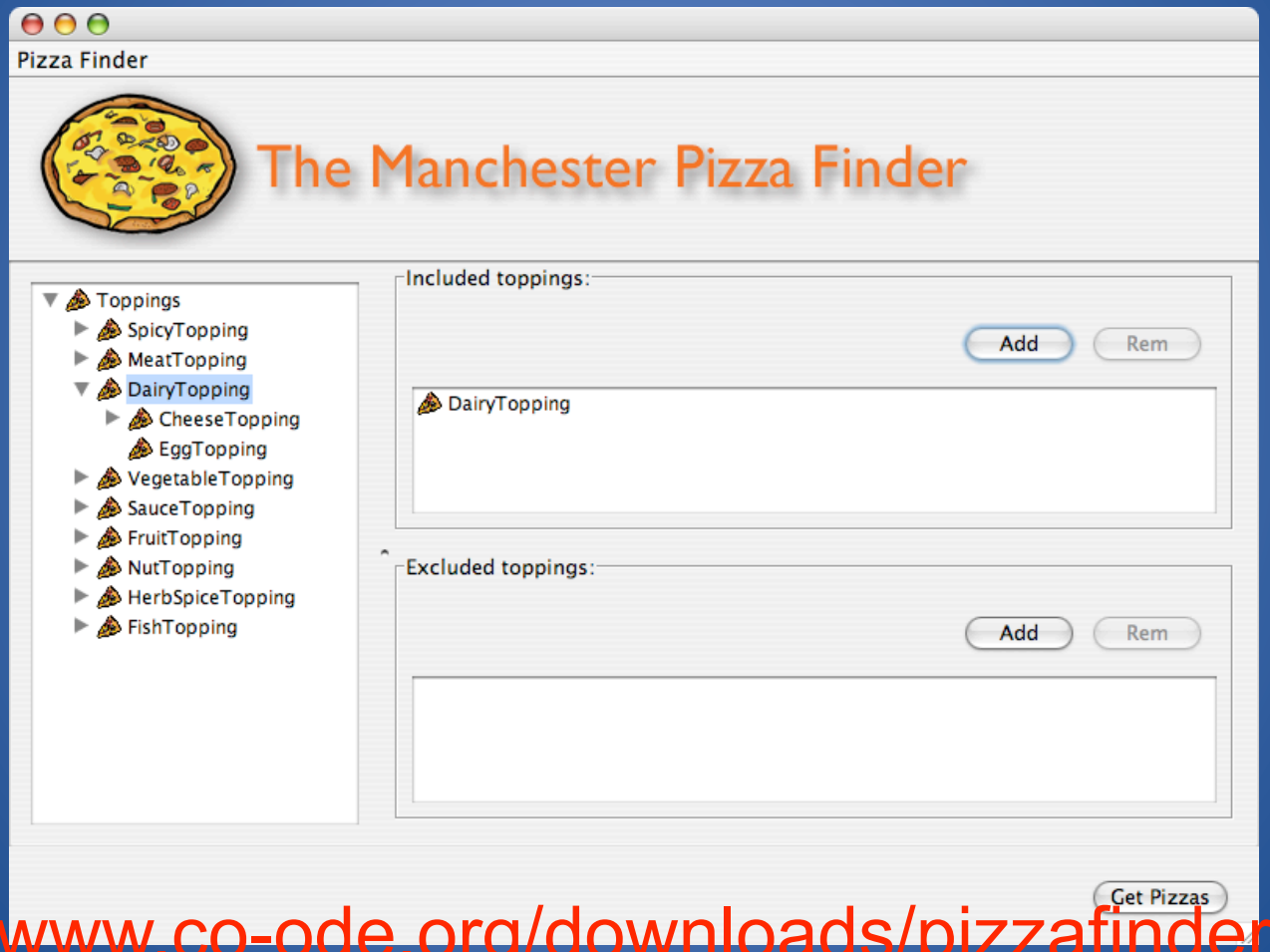
Components of an Ontology

- Vocabulary (concepts)
- Structure (attributes of concepts and hierarchy)
- Relationships between concepts
- Logical characteristics of relationships
  - Domain and range restrictions
  - Properties of relations (symmetry, transitivity)
  - Cardinality of relations
  - etc.

Some guiding rules of ontology design

- In most cases there are many ways to model a domain
- Ontology development, like program development, is by nature iterative
- The ontology should closely correspond to the objects (nouns) and relationships (verbs) in the sentences describing your domain of interest
- When building an ontology we need an application in mind – ontologies should not be built for the sake of it
- Keep the application in mind when creating concepts – this should help you scope the project

Our Application



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

Ontology Design is non-trivial

- different viewpoints
  - Tomato – Vegetable or Fruit?
  - culinary vs biological
- Ambiguity
  - words not concepts
- Missing Knowledge
  - What is peperonata?
- multiple classifications (2+ parents)
- lots of missing categories (superclasses?)
- competency questions
  - What are we likely to want to "ask" our ontology?
  - bear the application in mind

OWL Constructs: Classes

Eg Mammal, Tree, Person, Building, Fluid, Company

- Classes are sets of Individuals
- aka “Type”, “Concept”, “Category”
- 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 color 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)
- Classes that cannot possibly contain any Individuals are said to be **inconsistent**

OWL Constructs: Properties

Eg hasPart, isInhabitedBy, isNextTo, occursBefore

- Properties are used to relate Individuals
- We often say that Individuals are related **along** a given property
- Relationships in OWL are binary:

  Subject → predicate → Object

  Individual a → hasProperty → Individual b

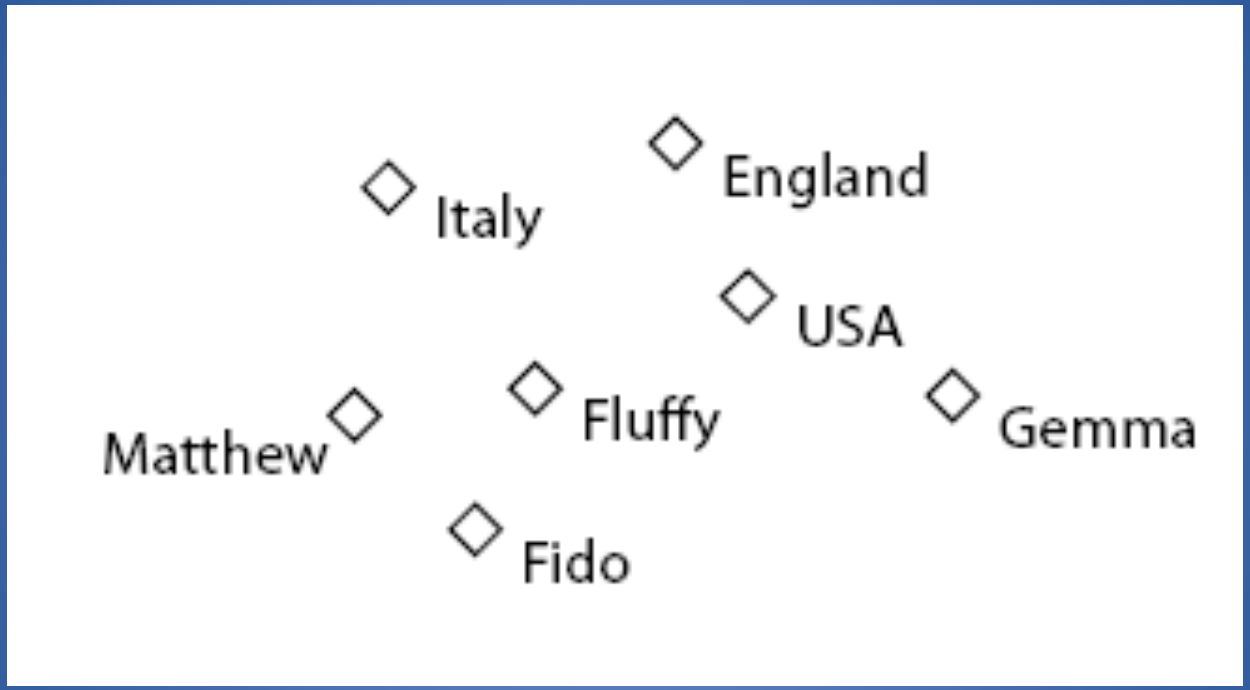
  nick_drummond → givesTutorial → Manchester_ProtegeOWL_tutorial_29_June_2005

OWL Constructs: Individuals

Eg me, you, this lecture, this room

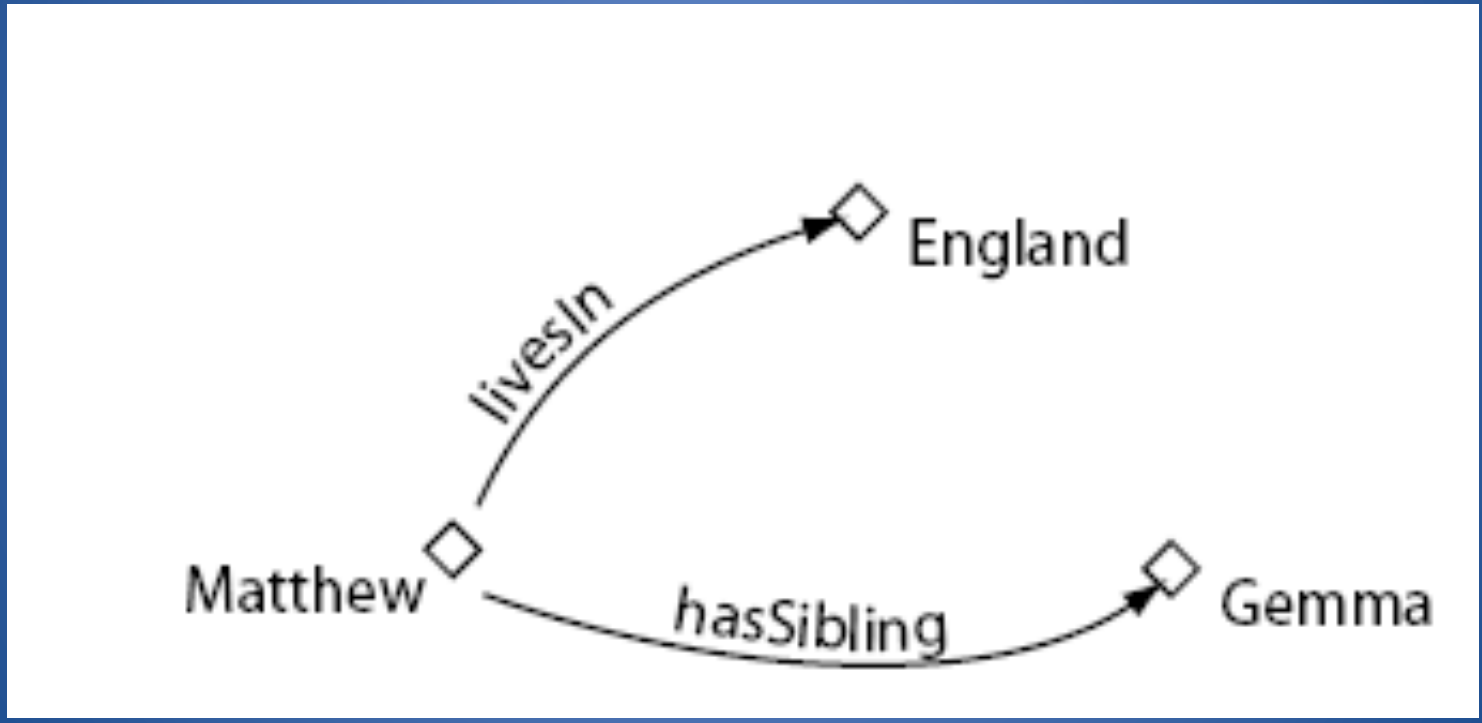
- Individuals are the objects in the domain
- aka “Instance”, “Object”
- Individuals may be (and are likely to be) a member of multiple Classes

Components of OWL Ontologies: Individuals

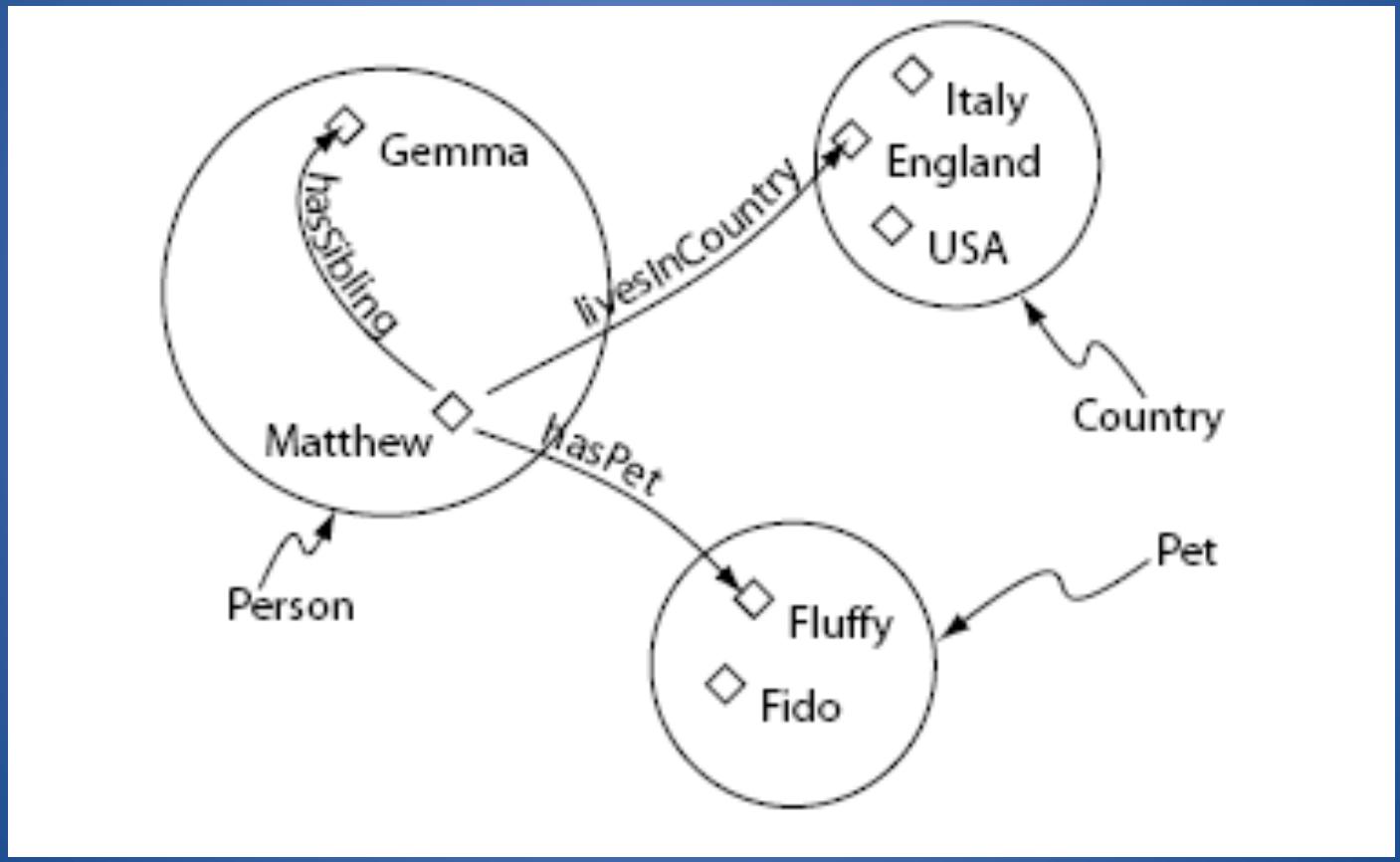


Unique name assumption: Two individuals are not equivalent (even if ids are different) unless explicitly stated so (open world)

Components of OWL Ontologies: Properties among Individuals

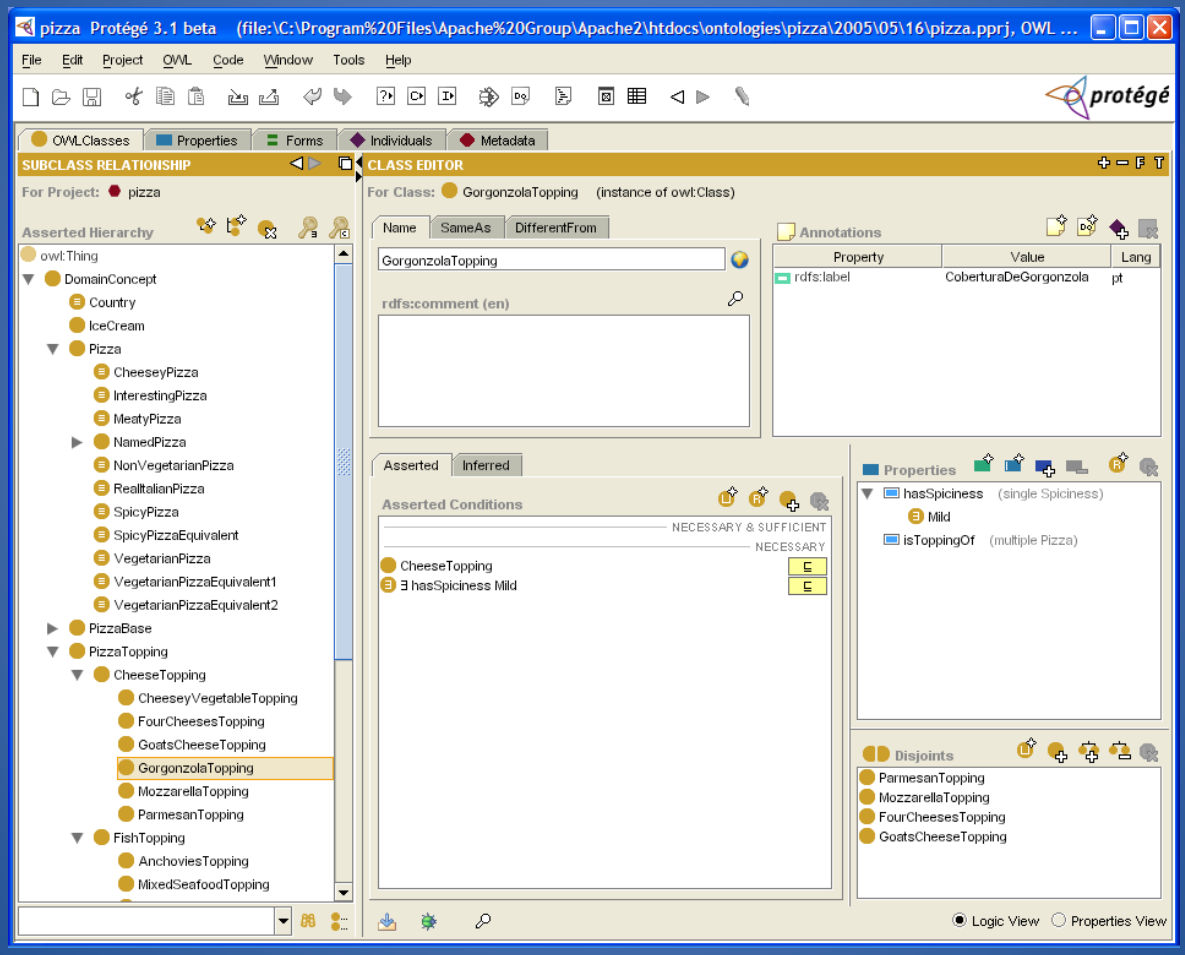


Components of OWL Ontologies: Classes, Properties, and Individuals

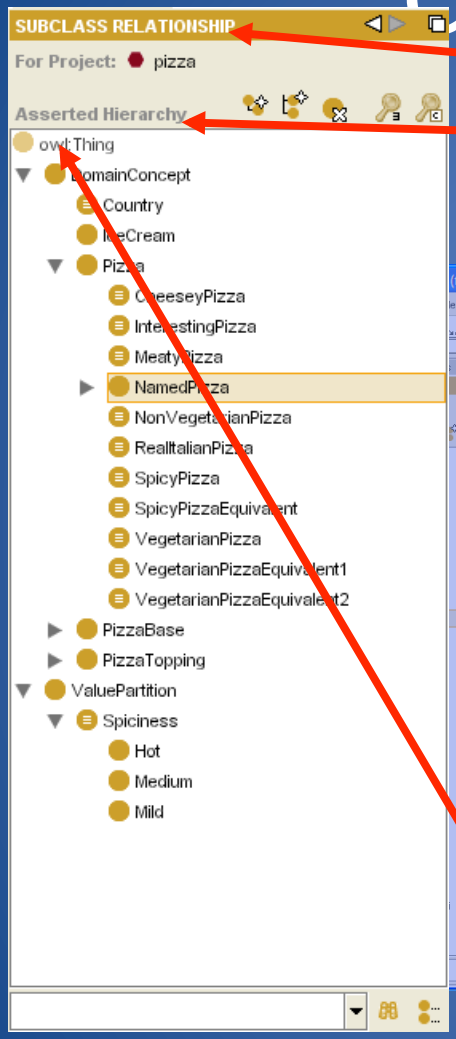


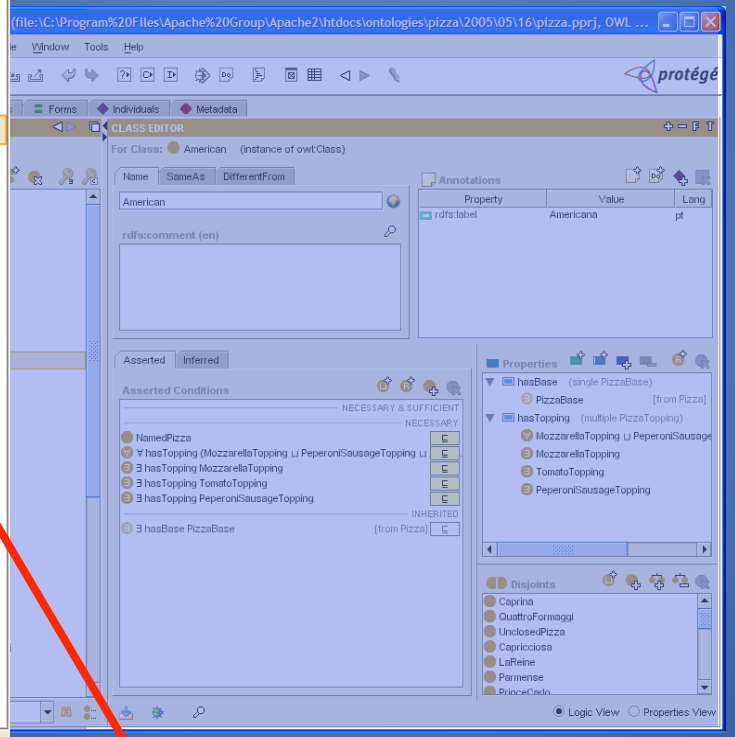
- All individuals must be in a class
- Define sets of individuals

Protégé-OWL


pizza Protégé 3.1 beta (file:\C:\Program%20Files\Apache%20Group\Apache2\htdocs\ontologies\pizza\2005\05\16\pizza.pprj, OWL ...
File Edit Project OWL Code Window Tools Help
protégé
OWLClasses Properties Forms Individuals Metadata
SUBCLASS RELATIONSHIP
For Project: pizza
Asserted Hierarchy
owl:Thing
DomainConcept
Country
IceCream
Pizza
CheeseyPizza
InterestingPizza
MeatyPizza
NamedPizza
NonVegetarianPizza
RealItalianPizza
SpicyPizza
SpicyPizzaEquivalent
VegetarianPizza
VegetarianPizzaEquivalent1
VegetarianPizzaEquivalent2
PizzaBase
PizzaTopping
CheeseTopping
CheeseyVegetableTopping
FourCheesesTopping
GoatsCheeseTopping
GorgonzolaTopping
MozzarellaTopping
ParmesanTopping
FishTopping
AnchoviesTopping
MixedSeafoodTopping
CLASS EDITOR
For Class: GorgonzolaTopping (instance of owl:Class)
Name SameAs DifferentFrom
GorgonzolaTopping
rdfs:comment (en)
Annotations
Property Value Lang
rdfs:label CoberturaDeGorgonzola pt
Asserted Inferred
Asserted Conditions
NECESSARY & SUFFICIENT
NECESSARY
CheeseTopping
∃ hasSpiciness Mild
Properties
hasSpiciness (single Spiciness)
Mild
isToppingOf (multiple Pizza)
Disjoints
ParmesanTopping
MozzarellaTopping
FourCheesesTopping
GoatsCheeseTopping
Logic View
Properties View

Class Hierarchy





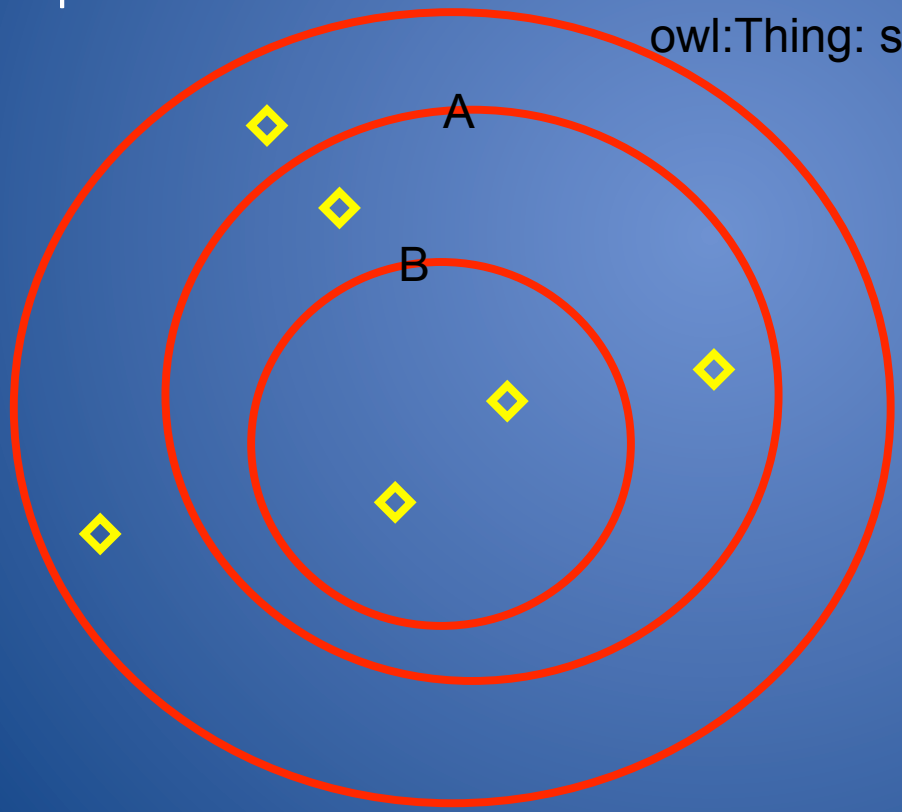
Subsumption hierarchy

Structure as asserted by the ontology engineer

owl:Thing is the root class

Subsumption

- Superclass/subclass relationship, “isa”
- **All** members of a subclass can be inferred to be members of its superclasses

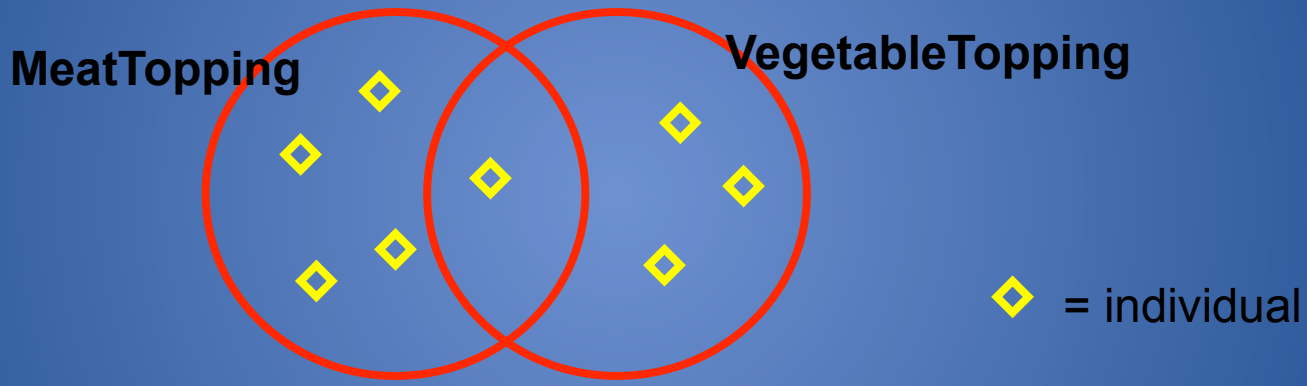


- A subsumes B
- A is a superclass of B
- B is a subclass of A
- **All** members of B are also members of A

Defined explicitly or inferred by a reasoner

Disjointness

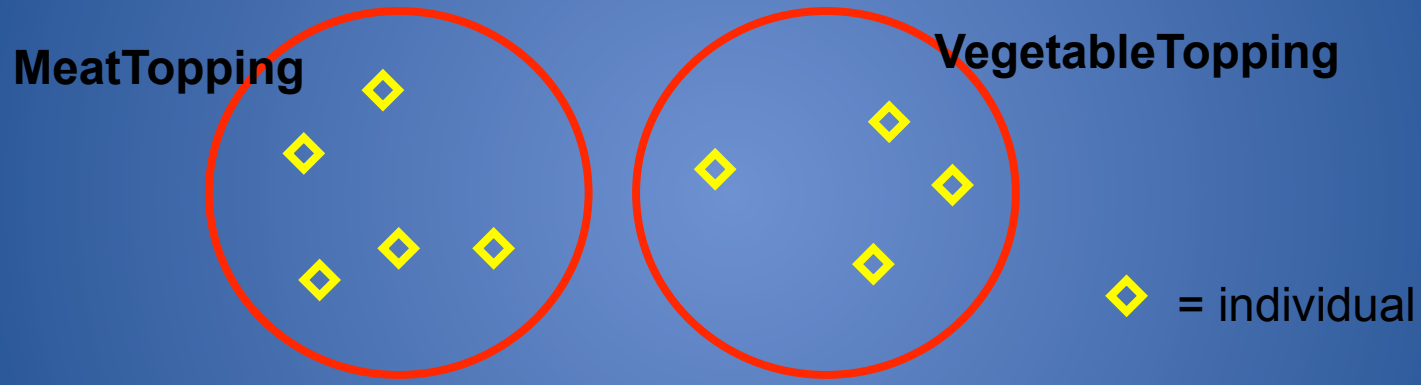
- OWL assumes that classes overlap



  - This means an individual could be both a **MeatTopping** and a **VegetableTopping** at the same time
  - We want to state this is not the case

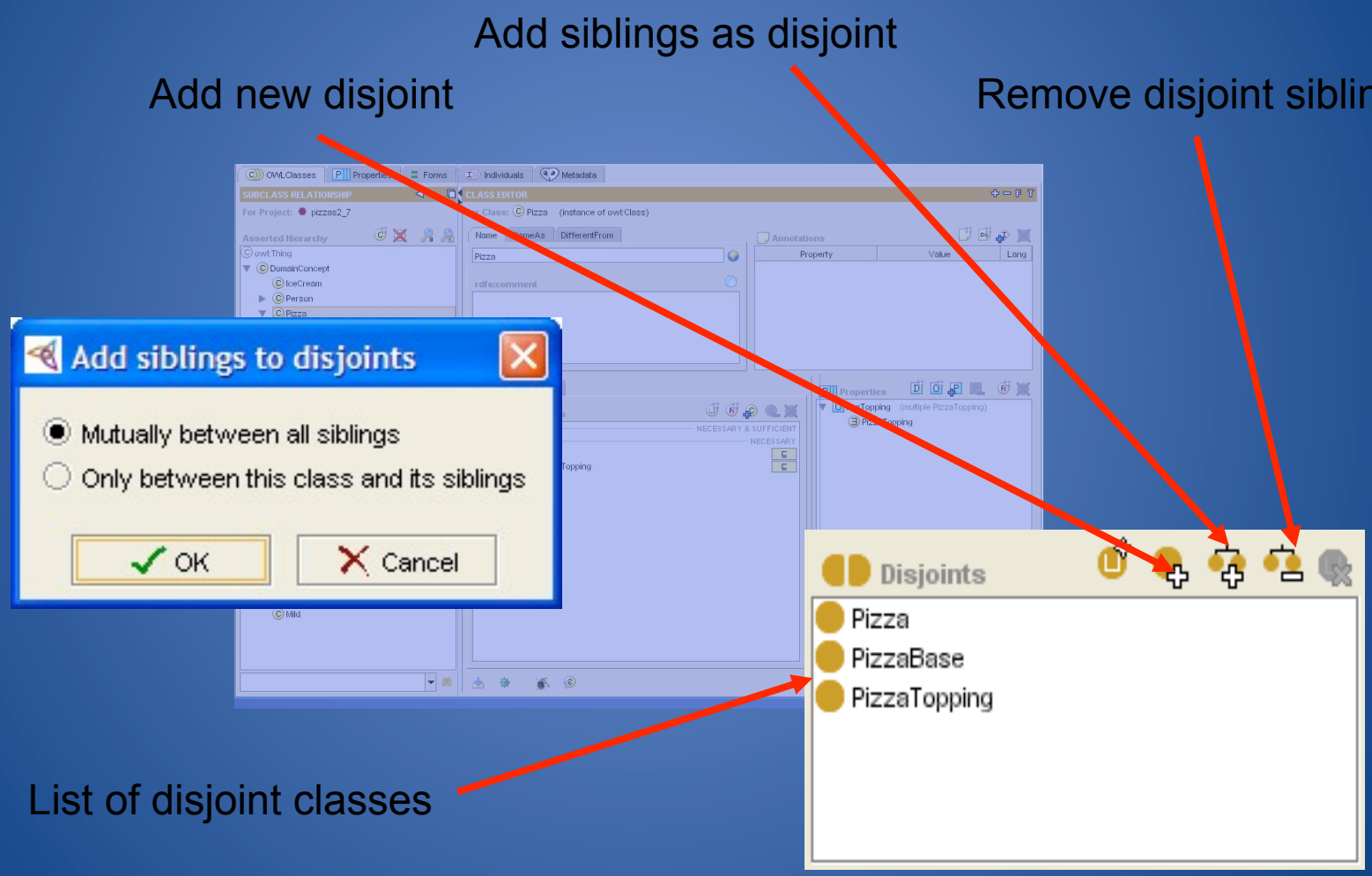
Disjointness

- If we state that classes are disjoint



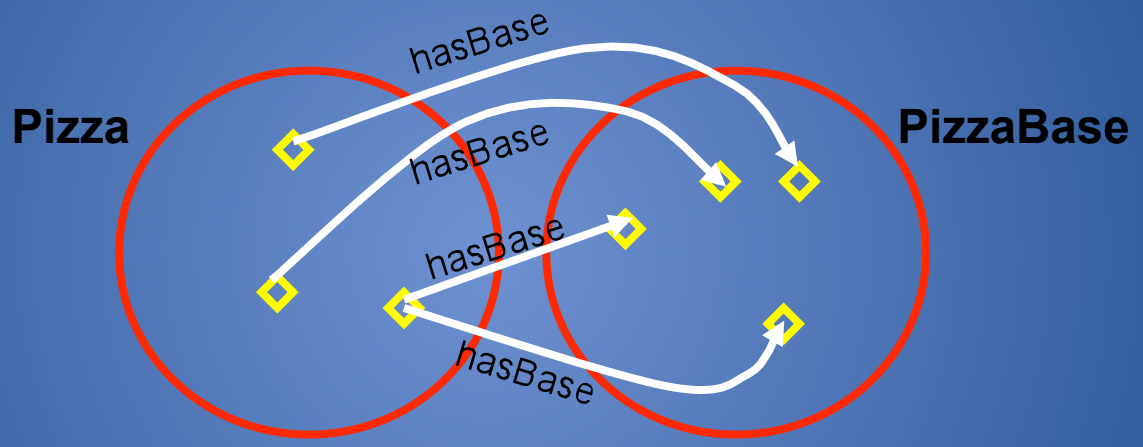
  - ► This means an individual cannot be both a **MeatTopping** and a **VegetableTopping** at the same time
  - ► We must do this explicitly in the interface

ClassesTab: Disjoints Widget



Restrictions

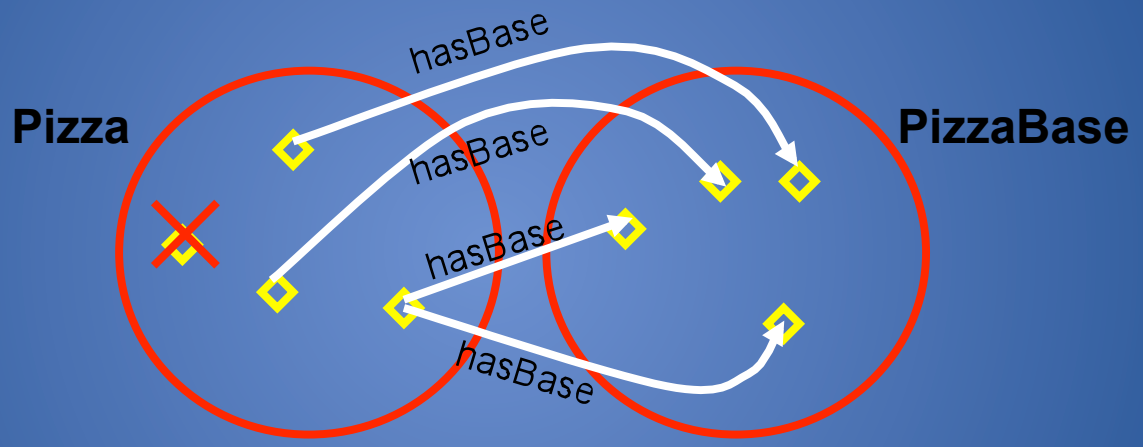
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



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

What does this mean?

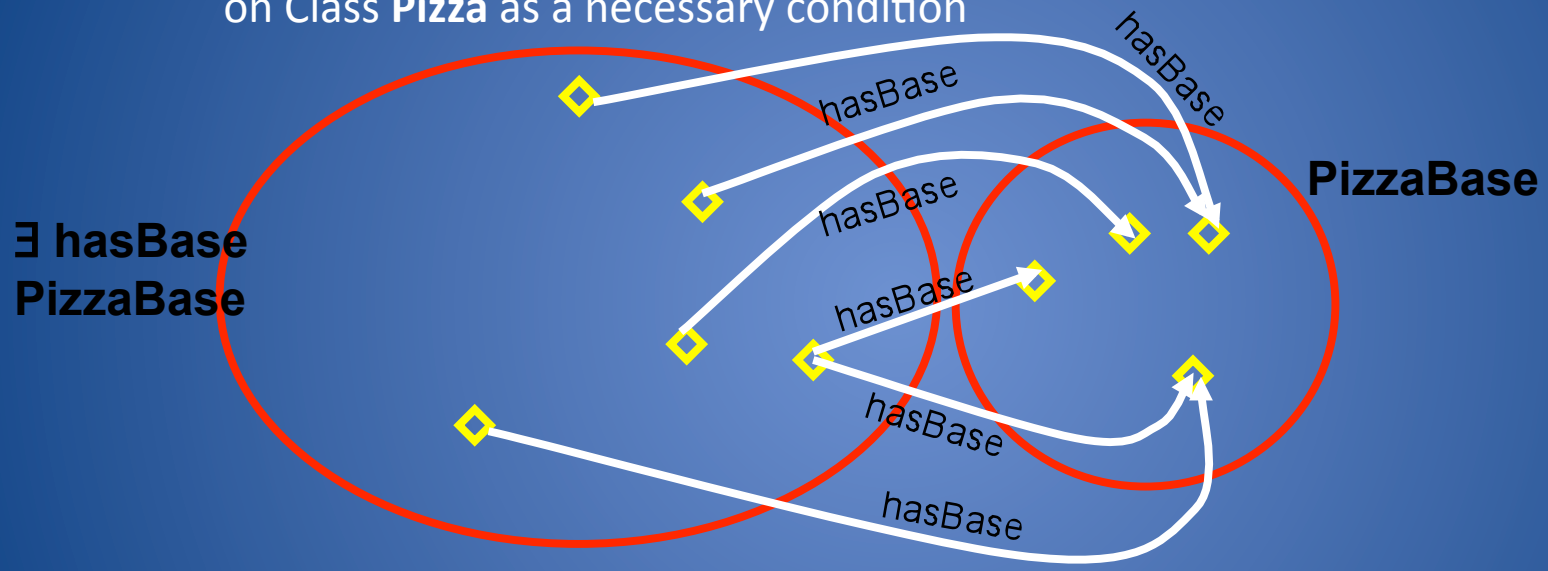
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



► "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?

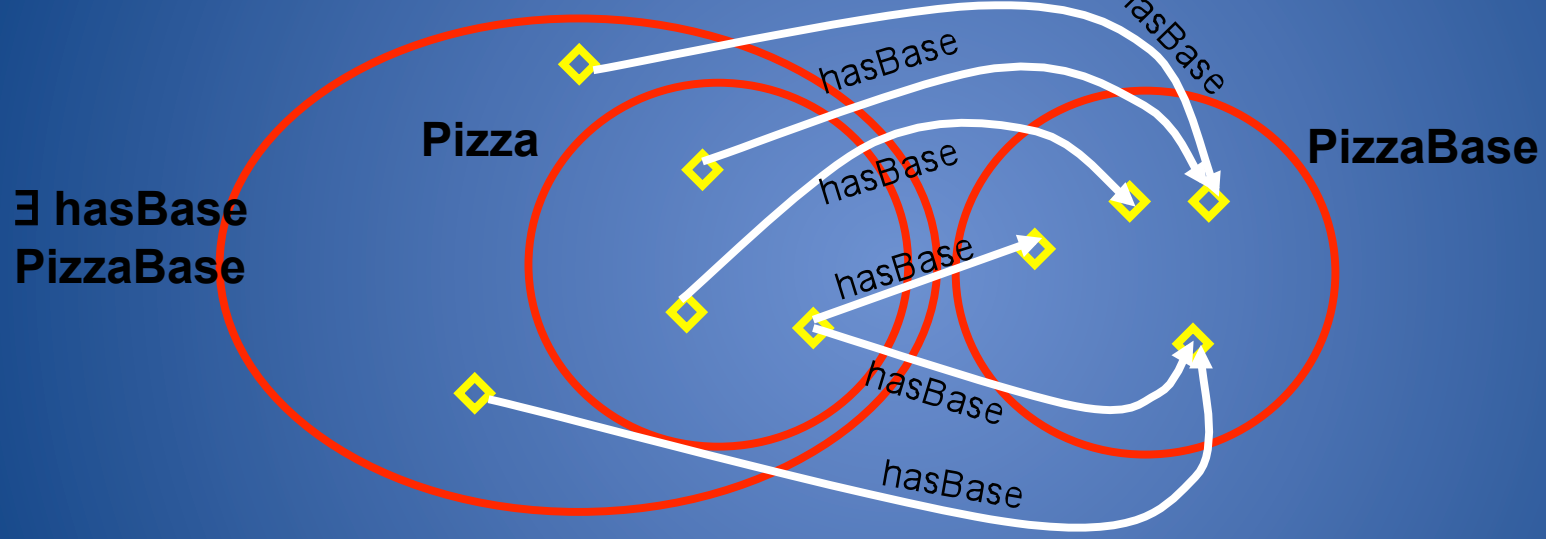
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



- Each Restriction or Class Expression describes the set of all individuals that satisfy the condition

Why? Necessary conditions

- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



- Each necessary condition on a class is a superclass of that class
- ie The restriction ∃ hasBase **PizzaBase** is a superclass of **Pizza**
- As **Pizza** is a subclass of the restriction, all **Pizza**s must satisfy the restriction that they have at least one base from **PizzaBase**

Consistency Checking

- Create a class that doesn't really make sense
  - What is a MeatyVegetableTopping?
- We'd like to be able to check the logical consistency of our model
- This is one of the tasks that can be done automatically by software known as a **Reasoner**
- Being able to use a reasoner is one of the main advantages of using a logic-based formalism such as OWL (and why we are using OWL-DL)

Reasoners

- Reasoners are used to infer information that is not explicitly contained within the ontology
- You may also hear them being referred to as Classifiers
- Standard reasoner services are:
  - Consistency Checking
  - Subsumption Checking
  - Equivalence Checking
  - Instantiation Checking

Reasoners and Protégé

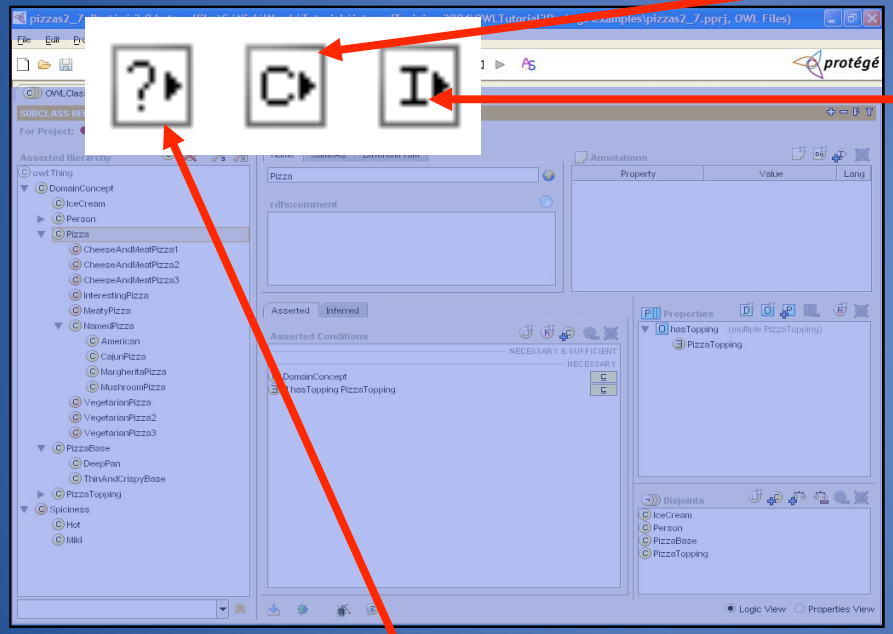
- Protégé-OWL supports the use of reasoners implementing the DIG interface
- This means that the reasoner you choose is independent of the ontology editor, so you can choose the implementation you want depending on your needs (eg some may be more optimised for speed/memory, others may have more features)
- These reasoners typically set up a service running locally or on a remote server – Protégé-OWL can only connect to reasoners over an http:// connection

- We will use Pellet

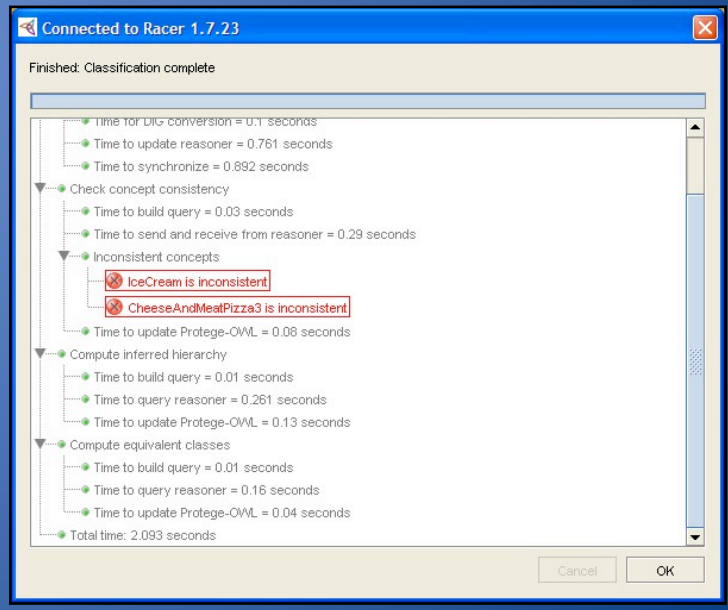
Accessing the Reasoner

Classify taxonomy
(and check consistency)

Compute inferred types
(for individuals)



Just check consistency
(for efficiency)



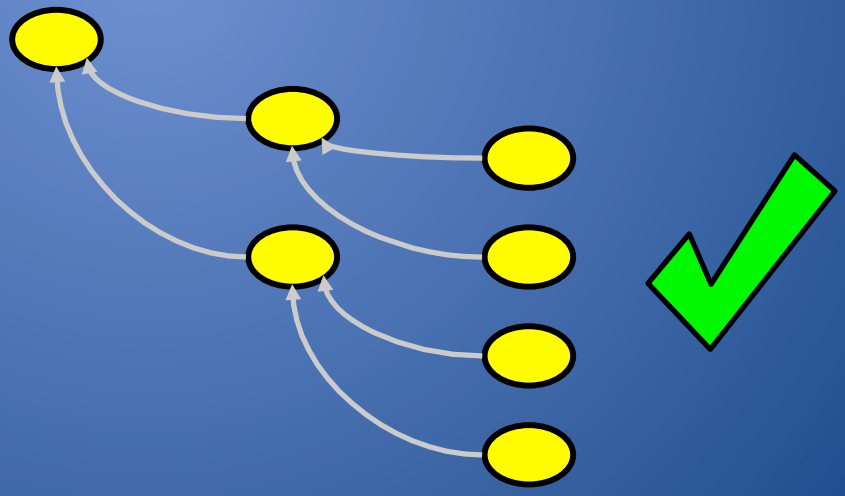
Reasoning about our Pizzas

- *When we classify an ontology we could just use the "Check Consistency" button but we'll get into the habit of doing a full classification as we'll be doing this later*
- *The reasoner dialog will pop up while the reasoner works*
- *When the reasoner has finished, you will see an inferred hierarchy appear, which will show any movement of classes in the hierarchy*
- *If the reasoner has inferred anything about our model, this is reported in the reasoner dialog and in a separate results window*
- *inconsistent classes turn red*
- *moved classes turn blue*

Primitive Classes

- Primitive Class = only Necessary Conditions
- Can not yet judge an individual based on primitive classes – why?

Start with building a disjoint tree of primitive classes



Polyhierarchies

- We want to create a **VegetarianPizza**
- Some of our existing Pizzas should be types of **VegetarianPizza**
- However, they could also be types of **SpicyPizza** or **CheeseyPizza**

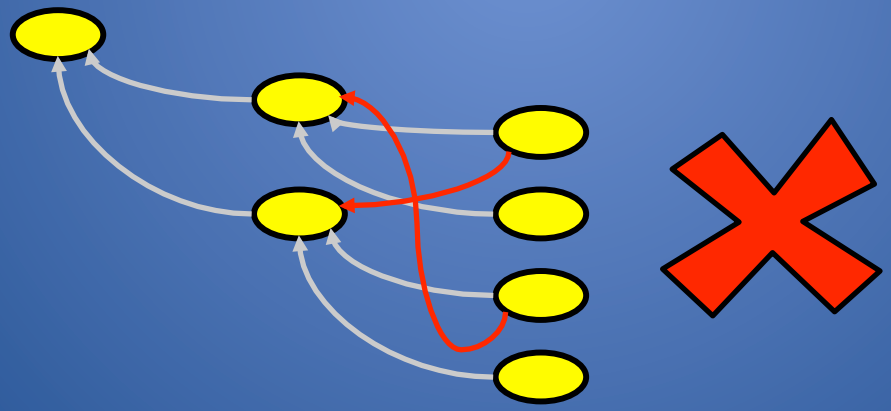
- We need to be able to give them multiple parents in a principled way
- We could just assert multiple parents like we did with **MeatyVegetableTopping** (without disjoints)

BUT…

Asserted Polyhierarchies

In most cases asserting polyhierarchies is bad

- We lose some encapsulation of knowledge
  - Why is this class a subclass of that one?
- Difficult to maintain
  - Adding new classes 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!

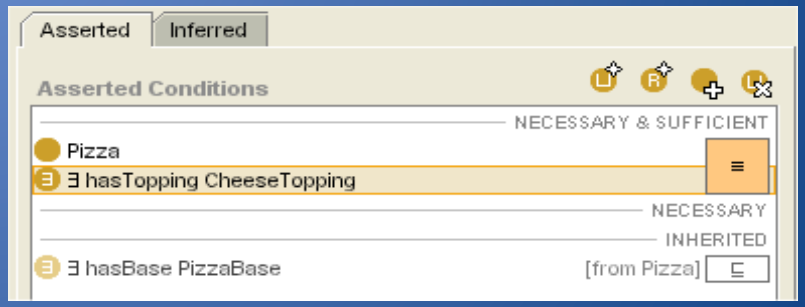
CheeseyPizza

- A CheeseyPizza is any pizza that has some cheese on it
- We would expect then, that some pizzas might be 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

Creating a CheeseyPizza

- We normally create primitive classes and then migrate them to defined classes
- All of our defined pizzas will be direct subclasses of Pizza
- So, we create a CheesyPizza Class (do not make it disjoint) and add a restriction:
  "Every **CheeseyPizza** must have at least one **CheeseTopping**"
- Classifying shows that we currently don't have enough information to do any classification

- We then move the conditions from the *Necessary* block to the *Necessary & Sufficient* block which changes the meaning



- And classify again…

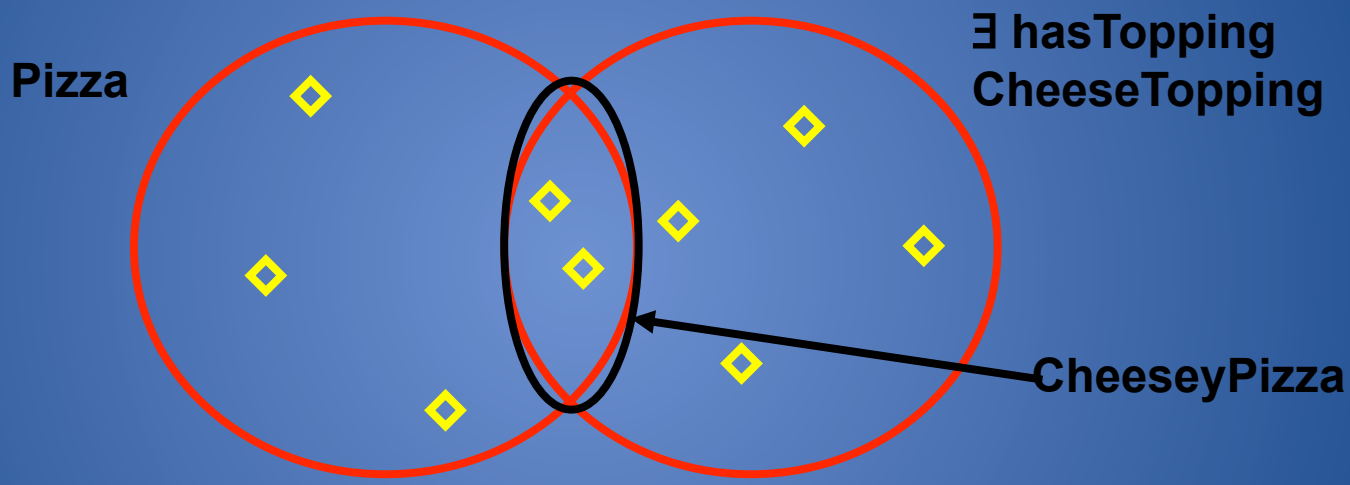
Reasoner Classification

- The reasoner has been able to infer that anything that is a **Pizza** that has at least one topping from **CheeseTopping** is a **CheeseyPizza**

► The inferred hierarchy is updated to reflect this and moved classes are highlighted in blue

Why?
Necessary & Sufficient Conditions

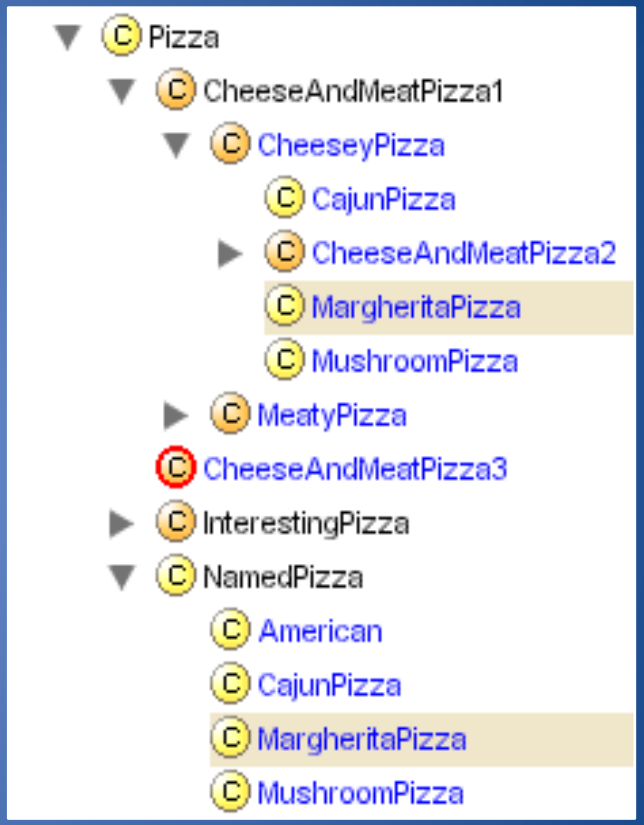
- Each set of necessary & sufficient conditions is an Equivalent Class



- **CheeseyPizza** is equivalent to the intersection of **Pizza** and **∃ hasTopping CheeseTopping**
- Classes, all of whose individuals fit this definition are found to be subclasses of **CheeseyPizza**, or are subsumed by **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



Mission Successful!

Untangling

- However, our unclassified version of the ontology is a simple tree, which is much easier to maintain

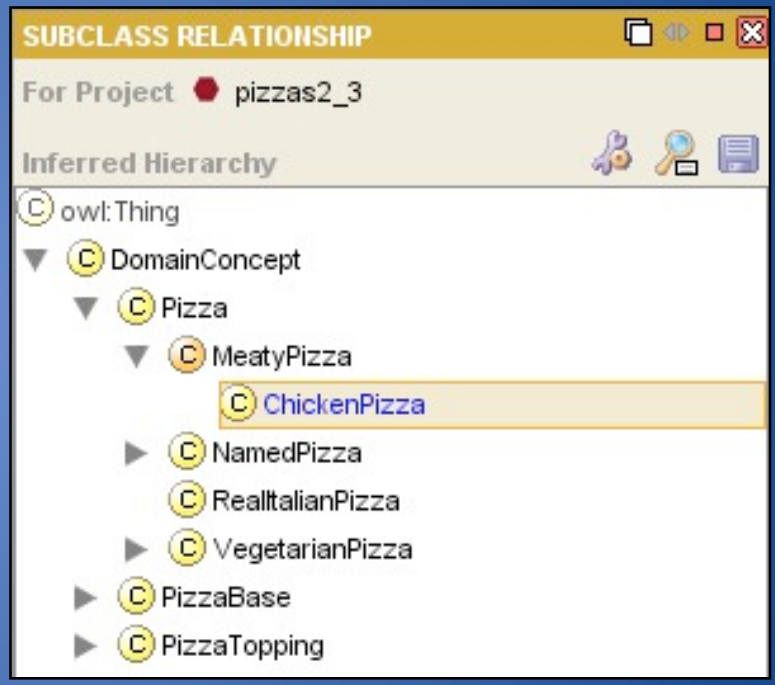
- We've now got a polyhierarchy without asserting multiple superclass relationships
- Plus, we also know why certain pizzas have been classified as CheeseyPizzas

Untangling

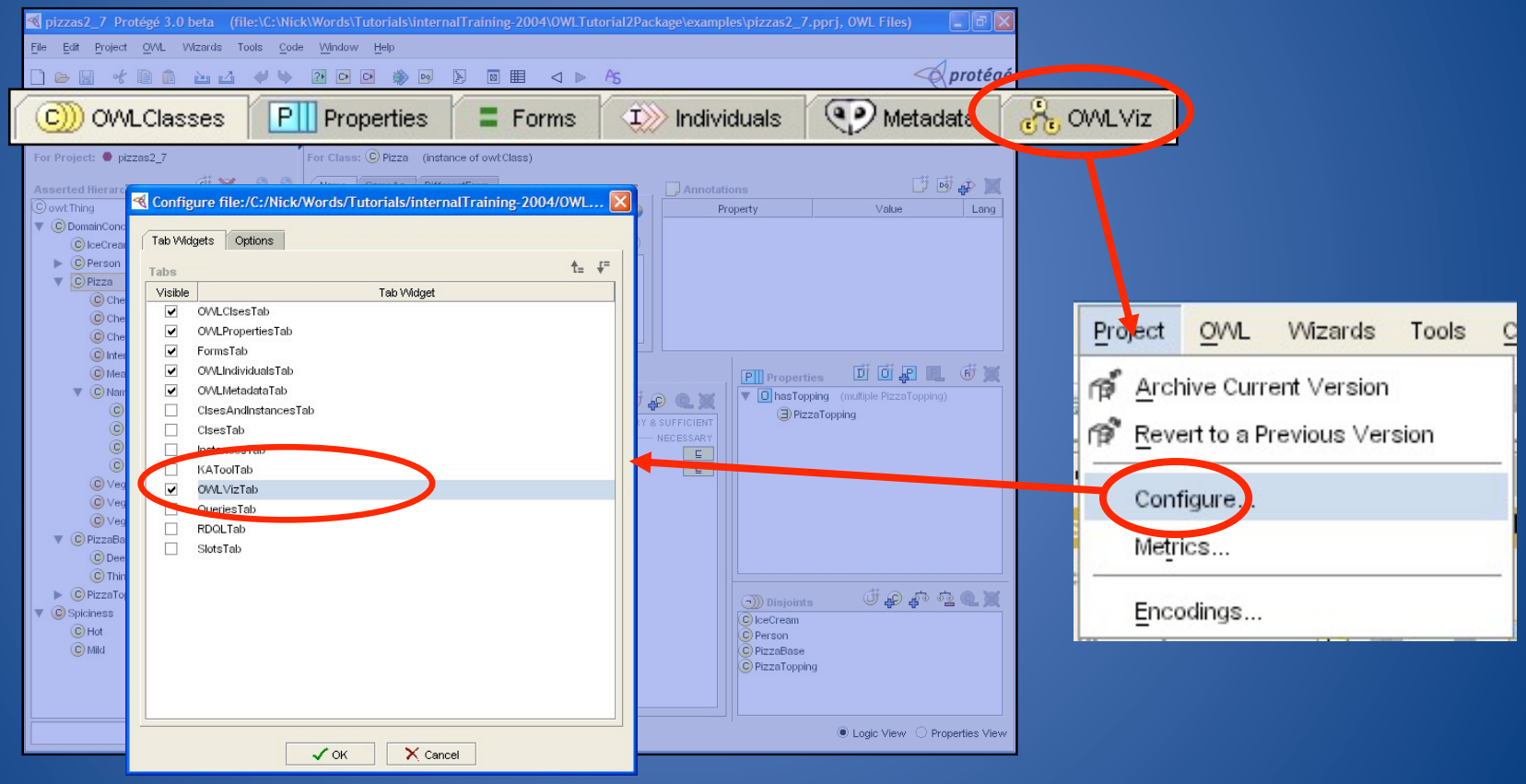
- We don’t currently have many kinds of primitive pizza but its easy to see that if we had, it would have been a substantial task to assert **CheeseyPizza** as a parent of lots, if not all, of them
- And then do it all over again for other defined classes like **MeatyPizza** or whatever

Viewing polyhierarchies

- As we now have multiple inheritance, the tree view is less than helpful in viewing our “hierarchy”



Viewing our Hierarchy Graphically

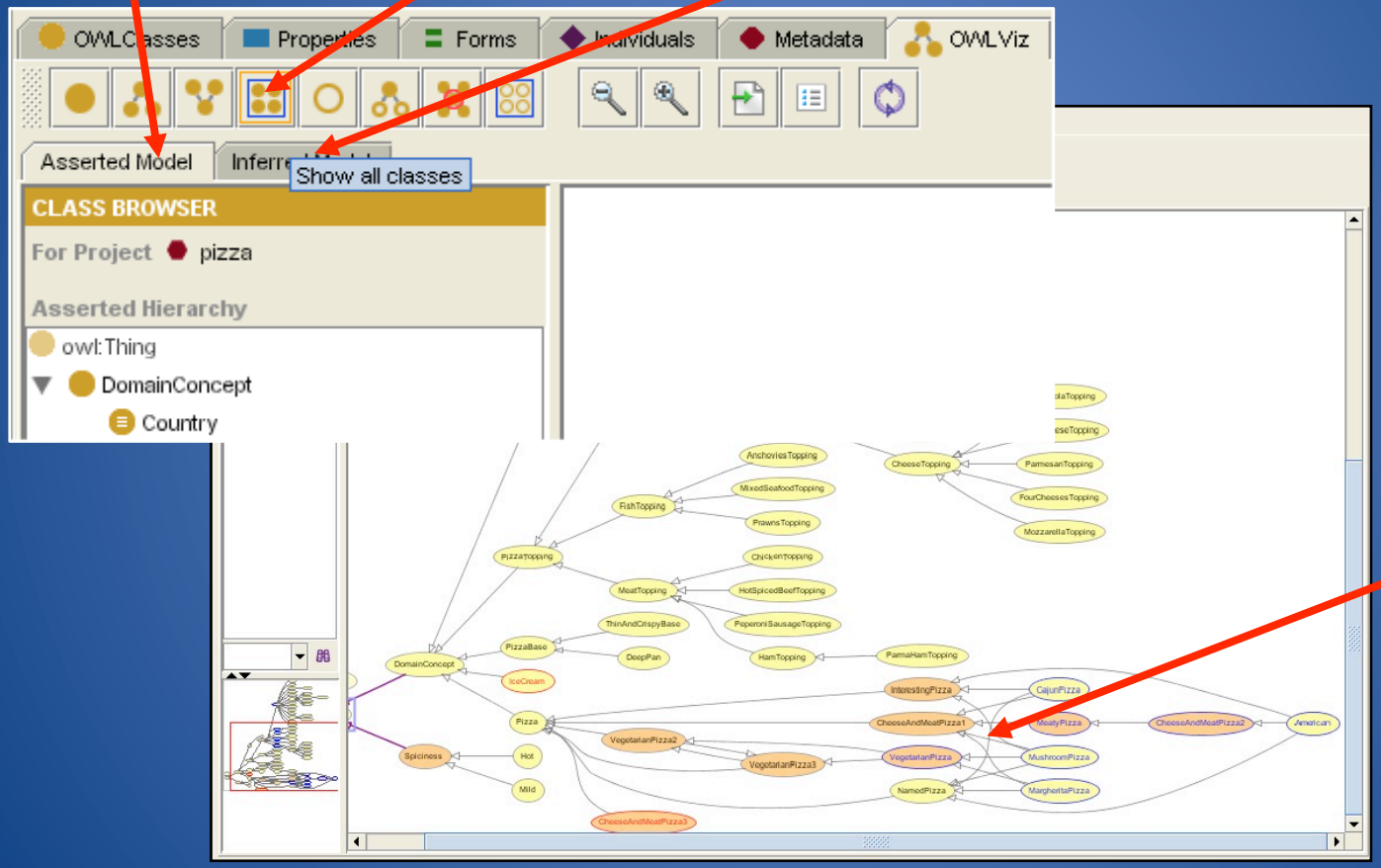


OWLViz Tab

Show All Classes

View Asserted Model

View Inferred Model



Polyhierarchy tangle

Using OWLViz to untangle

- The asserted hierarchy should, ideally, be a tidy tree of disjoint primitives
- The inferred hierarchy will be tangled
- By switching from the asserted to the inferred hierarchy, it is easy to see the changes made by the reasoner
- OWLViz can be used to spot tangles in the primitive tree and also disjoints (including inherited ones) are marked (with a ¬)

Defined Classes

- We've created a Defined Class, **CheeseyPizza**
  - It has a definition. That is *at least one* Necessary and Sufficient condition
  - Classes, all of whose individuals satisfy this definition, can be inferred to be subclasses
  - Therefore, we can use it like a query to "collect" subclasses that satisfy its conditions
  - Reasoners can be used to organise the complexity of our hierarchy
- It's marked with an equivalence symbol in the interface
- Defined classes are rarely disjoint

Define a Vegetarian Pizza

- Not as easy as it looks…
- Define in words?
  - "a pizza with only vegetarian toppings"?
  - "a pizza with no meat (or fish) toppings"?
  - "a pizza that is not a MeatyPizza"?
- More than one way to model this

We'll start with the first example

Define a Vegetarian Pizza

To be able to define a vegetarian pizza as
a Pizza with only Vegetarian Toppings

we need:

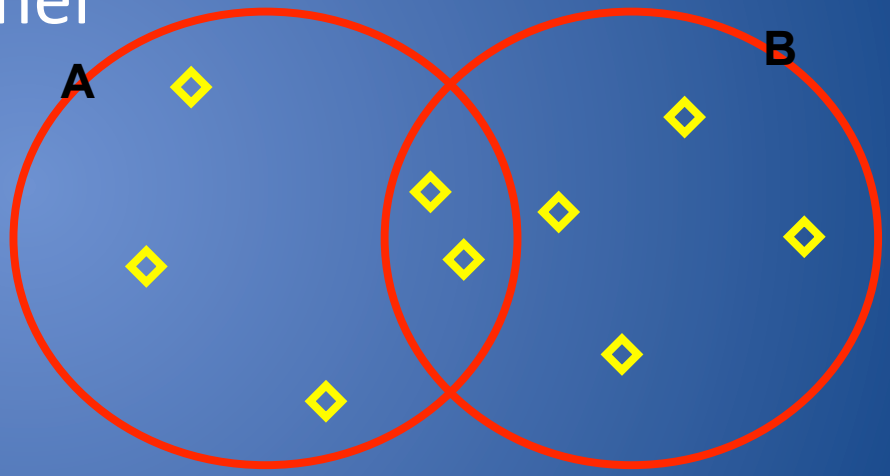
1. To be able to create a vegetarian topping
   This requires a Union Class

2. To be able to say “only”
   This requires a Universal Restriction

Union Classes

- aka “disjunction”
- This OR That OR TheOther
- This ⊔ That ⊔ TheOther

A⊔B includes all individuals of class A and all individuals from class B and all individuals in the overlap (if A and B are not disjoint)



- Commonly used for:
  - Covering axioms
  - Closure

Covering Axioms

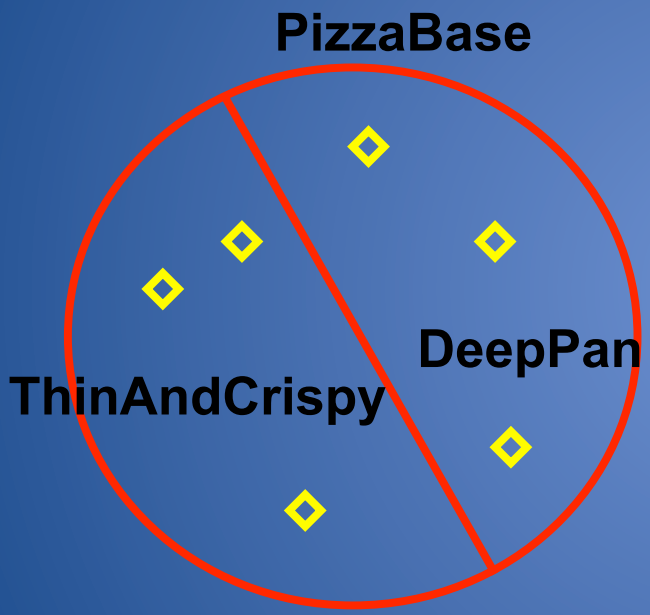
- Covering axiom – a union expression containing several covering classes

- A covering axiom in the *Necessary & Sufficient* Conditions of a class means:
  the class cannot contain any instances other than those from the covering classes

- NB. If the covering classes are subclasses of the covered class, the covering axiom only needs to be a Necessary condition – it doesn't harm to make it Necessary & Sufficient though – its just redundant

Covering PizzaBase

PizzaBase ≡ ThinAndCrispy ⊔ DeepPan



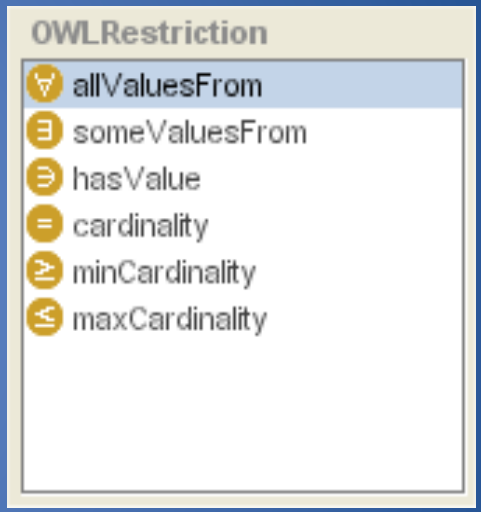
- In this example, the class **PizzaBase** is covered by **ThinAndCrispy** or **DeepPan**
- "All **PizzaBase**s must be **ThinAndCrispy** or **DeepPan**"
- "There are no other types of **PizzaBase**"

Universal Restrictions

- We need to say our **VegetarianPizza** can only have toppings that are vegetarian toppings
- We can do this by creating a Universal or AllValuesFrom restriction
- We'll first look at an example…

Real Italian Pizzas

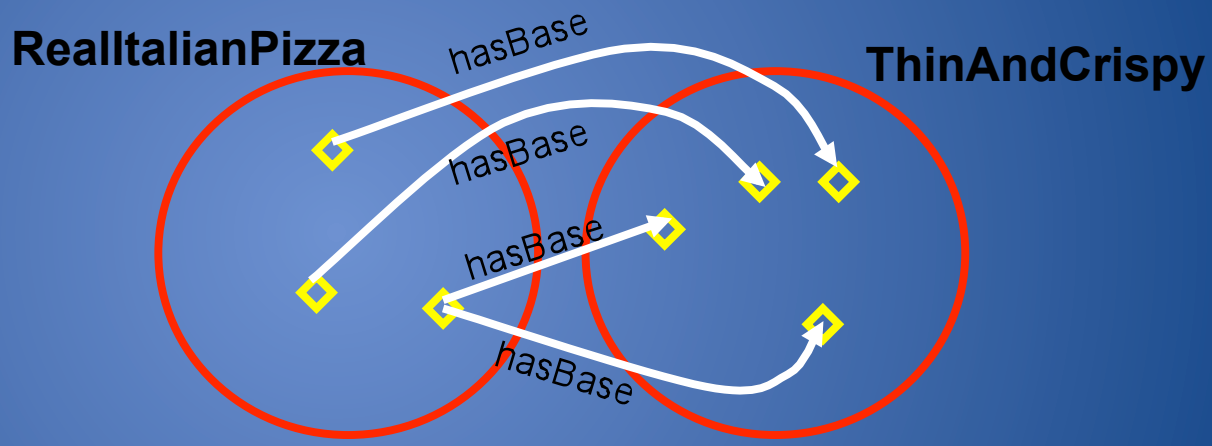
- "RealItalianPizzas only have bases that are ThinAndCrispy"
- A Universal Restriction is added just like an Existential one, but the restriction type is different



- For now, this can be primitive – you can make it defined if you like

What does this mean?

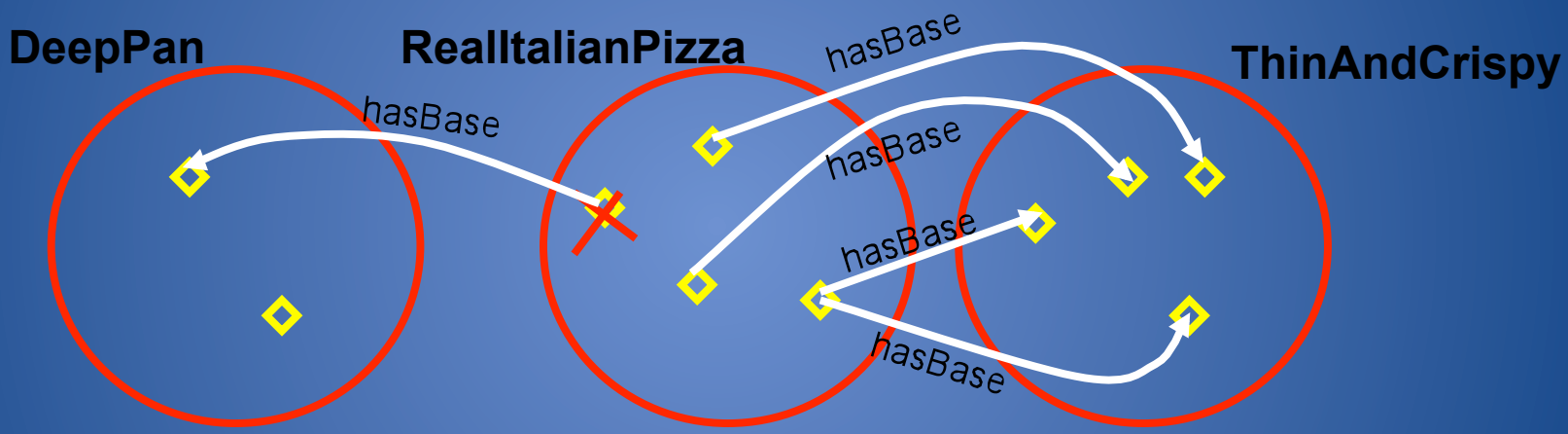
- We have created a restriction: ∀ hasBase **ThinAndCrispy** on Class **RealItalianPizza** as a necessary condition



- "If an individual is a member of this class, it is necessary that it must only have a hasBase relationship with an individual from the class **ThinAndCrispy**"

What does this mean?

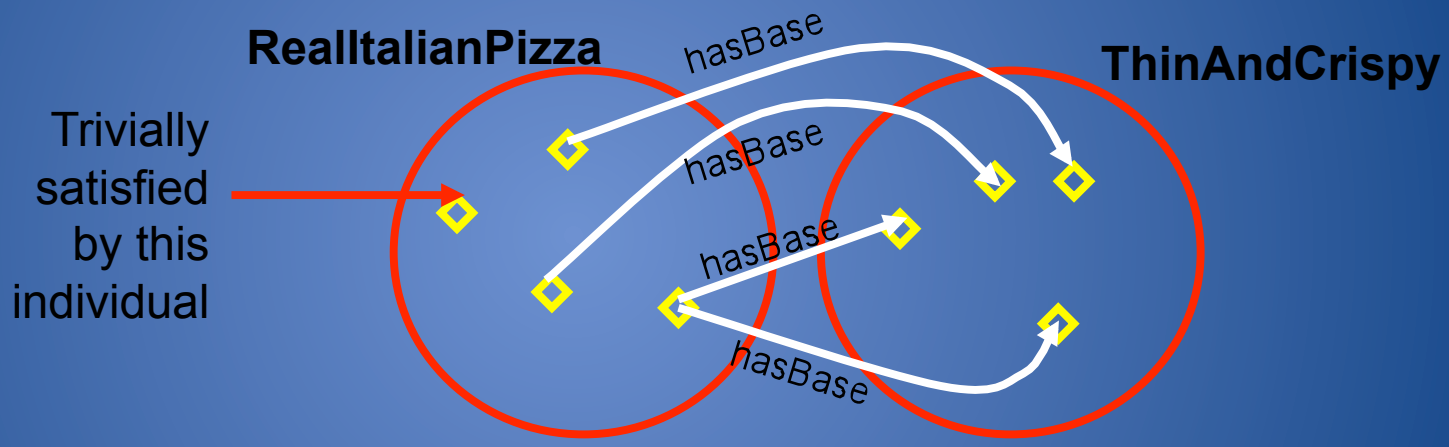
- We have created a restriction: ∀ hasBase **ThinAndCrispy** on Class **RealItalianPizza** as a necessary condition



- "No individual of the **RealItalianPizza** class can have a base from a class other than **ThinAndCrispy**"
- NB. DeepPan and ThinAndCrispy are disjoint

Warning: Trivial Satisfaction

- If we had not already inherited: ∃ hasBase **PizzaBase** from Class **Pizza** the following could hold



- "If an individual is a member of this class, it is necessary that it must only have a hasBase relationship with an individual from the class **ThinAndCrispy,** or no hasBase relationship at all"
- Universal Restrictions by themselves do not state "at least one"

VegetarianPizza Classification

- Nothing classifies under **VegetarianPizza**
- Actually, there is nothing wrong with our definition of **VegetarianPizza**
- It is actually the descriptions of our **Pizza**s that are incomplete
- The reasoner has not got enough information to infer that any **Pizza** is subsumed by **VegetarianPizza**
- This is because OWL makes the Open World Assumption

Open World Assumption

- In a closed world (like DBs), the information we have is everything
- In an open world, we assume there is always more information than is stated

- Where a database, for example, returns a negative if it cannot find some data, the reasoner makes no assumption about the completeness of the information it is given
- The reasoner cannot determine something does not hold unless it is explicitly stated in the model

Open World Assumption

- Typically we have a pattern of several Existential restrictions on a single property with different fillers – like primitive pizzas on hasTopping
- Existential restrictions should be paraphrased by "amongst other things…"
- Must state that a description is complete
- We need closure for the given property

Closure

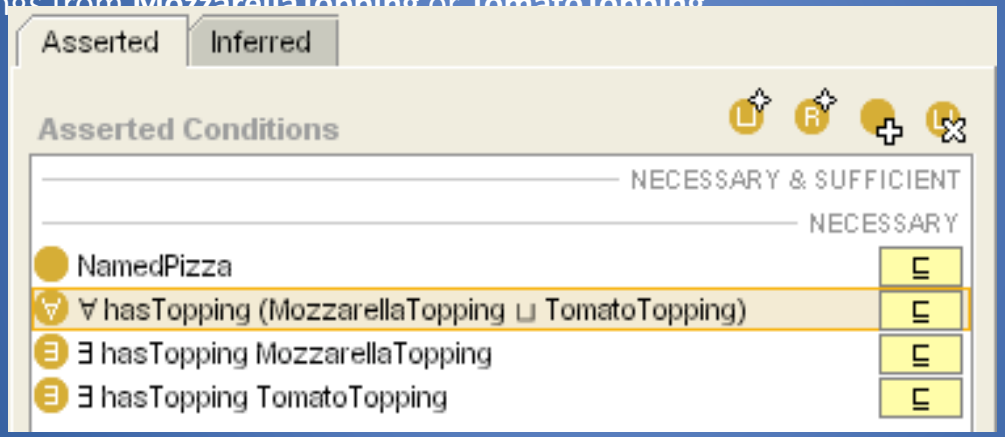
- This is in the form of a Universal Restriction with a filler that is the Union of the other fillers for that property

- Closure works along a single property

Closure example: MargheritaPizza

All **MargheritaPizzas** must have:

at least 1 topping from **MozzarellaTopping** and
at least 1 topping from **TomatoTopping** and
only toppings from **MozzarellaTopping** or **TomatoTopping**



- The last part is paraphrased into “no other toppings”
- The union closes the hasTopping property on **MargheritaPizza**