AN INTRODUCTION TO PROTEGE

(Based on the OWL tutorial by N. Drummond and M. Horridge)

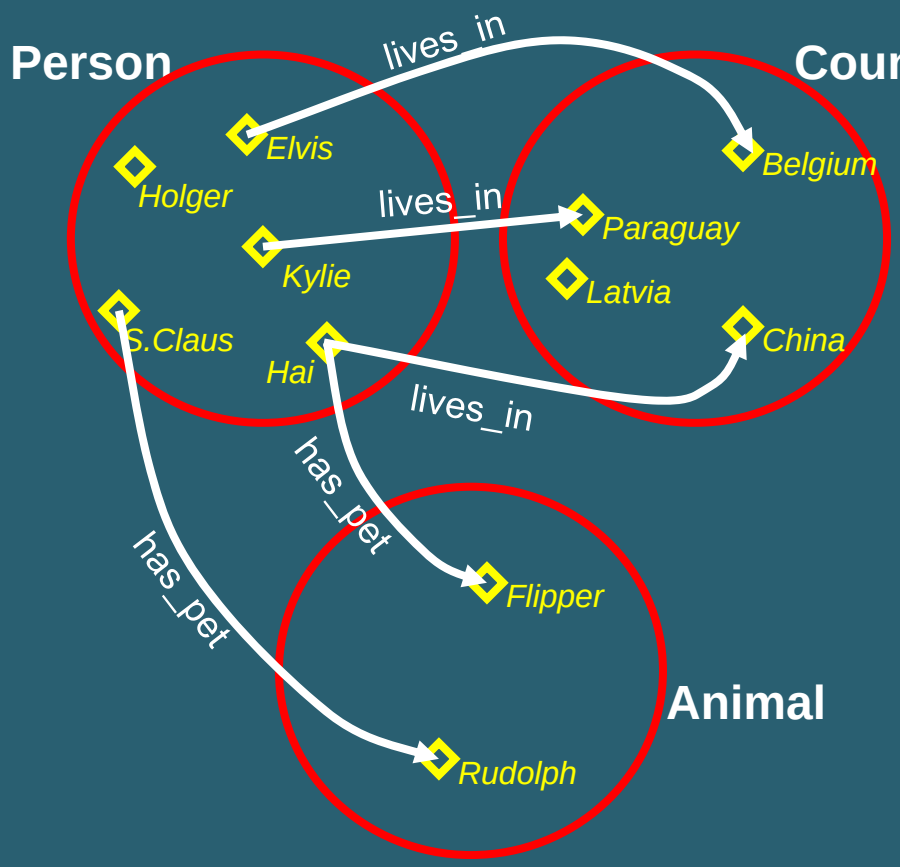
Review of OWL

OWL…

- is a W3C standard – Web Ontology Language
- is generally found in XML/RDF syntax
- is therefore not much fun to write by hand

So, we have tools to help us

OWL Constructs



= class (concept)

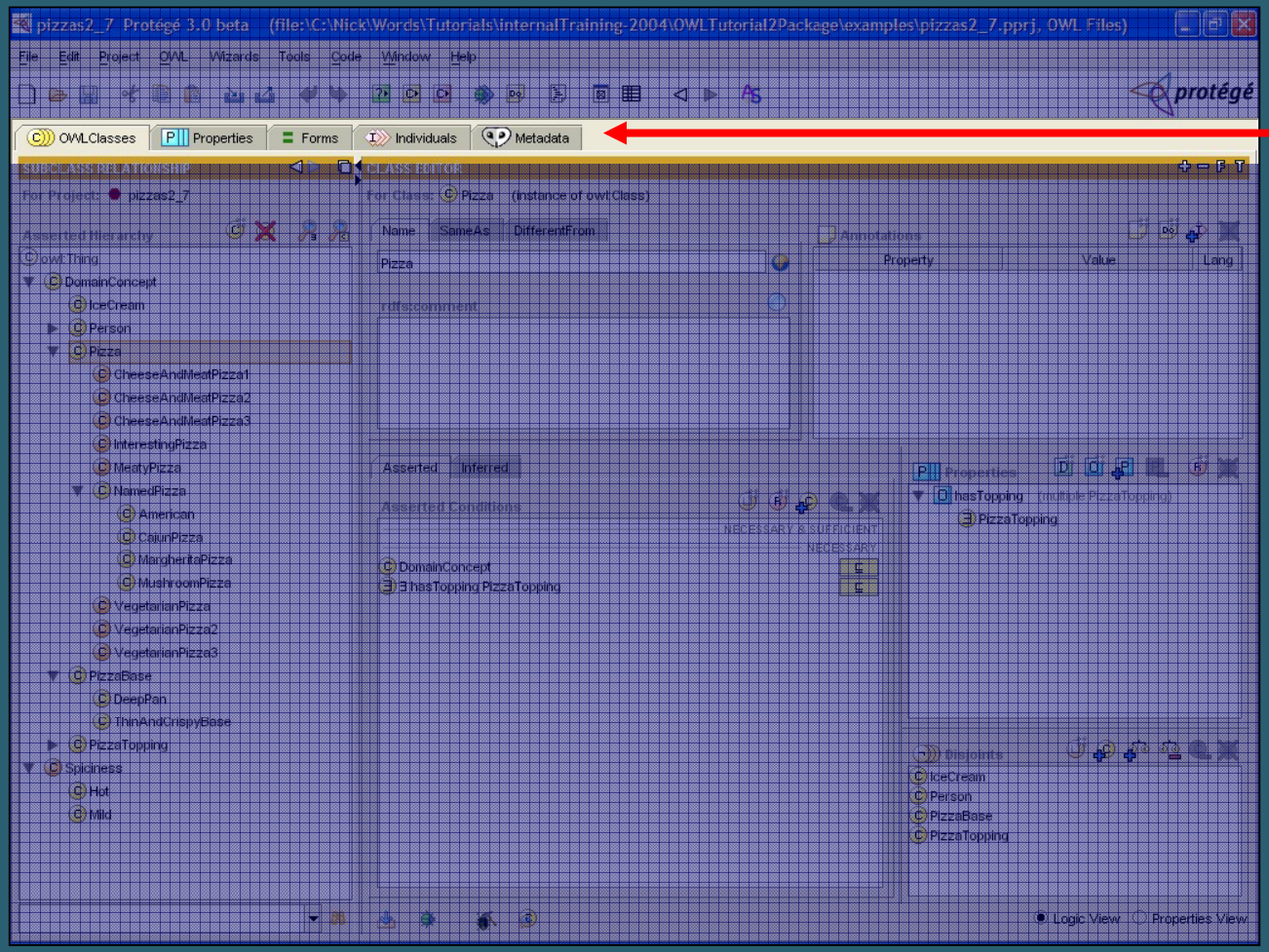
= property (relationship)

= individual (instance)

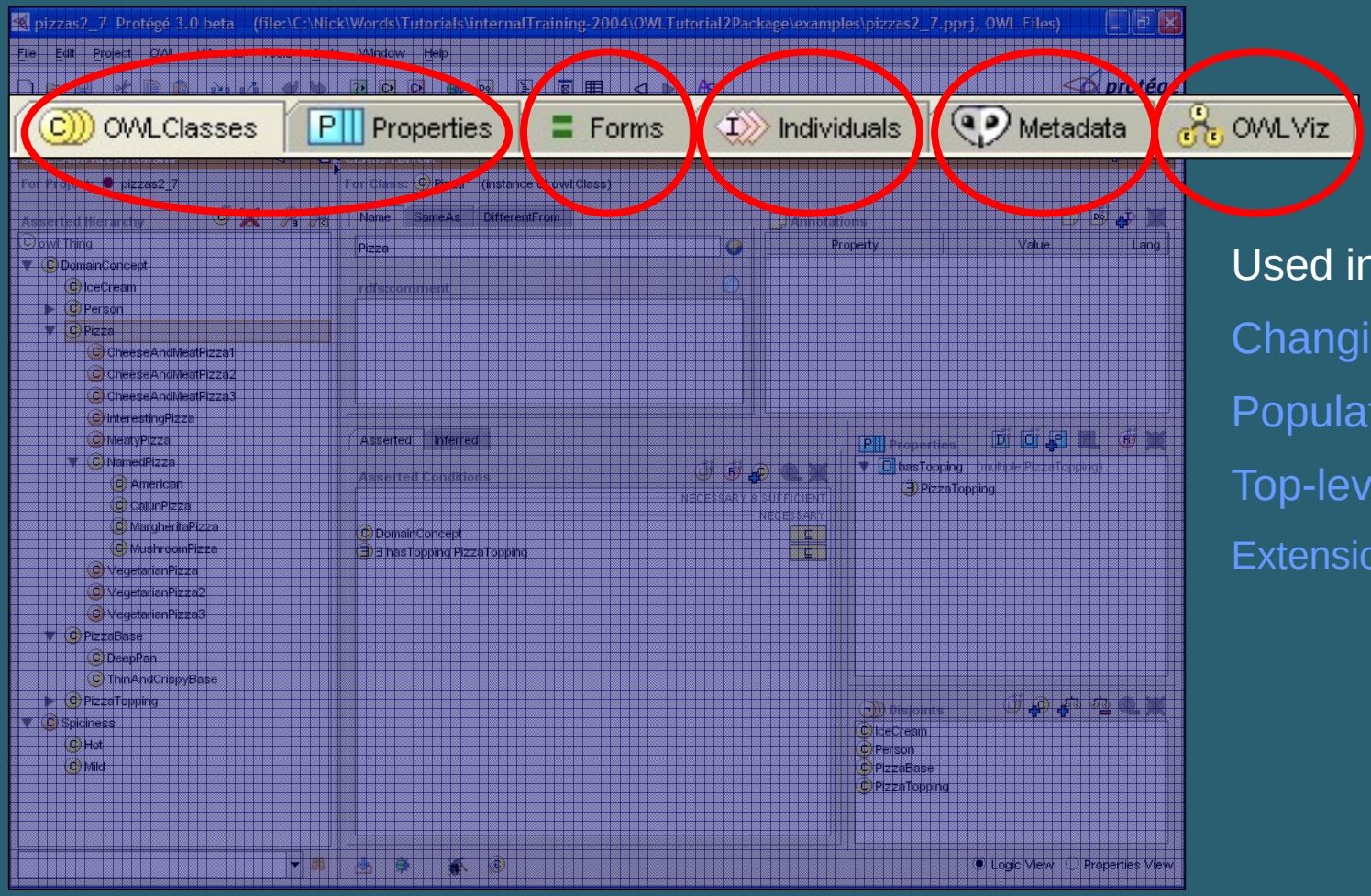
Installation of Protege

- Install Protege 3 (the software will be provided)
  1. Go to: http://www-inf.it-sudparis.eu/~gaaloulw/KM/
  2. Download Protege3.0.zip
  3. Extract to your home directory
  4. Open “run_protege.bat”

Protégé OWL plugin



Protégé OWL plugin: Tabs



Used in this tutorial

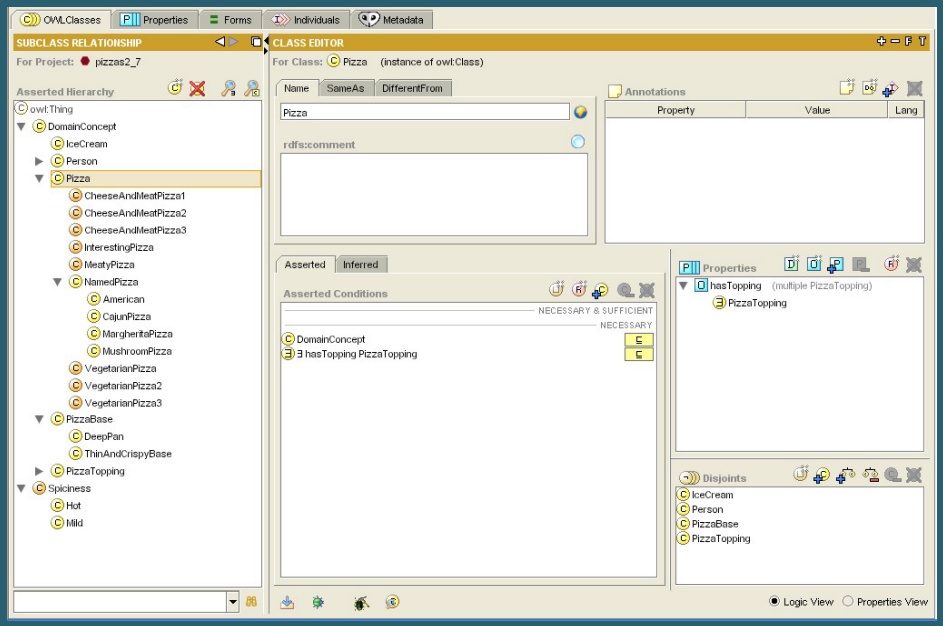
Changing the GUI

Populating the model

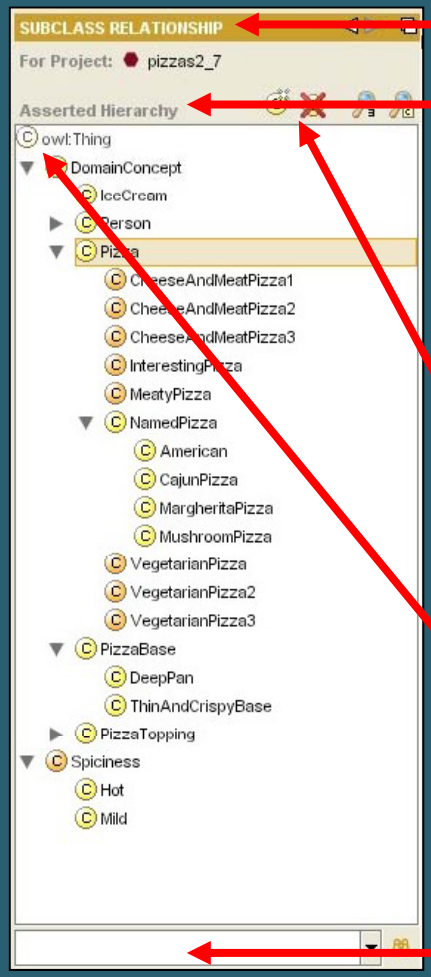
Top-level functionality

Extensions (visualisation)

Classes Tab

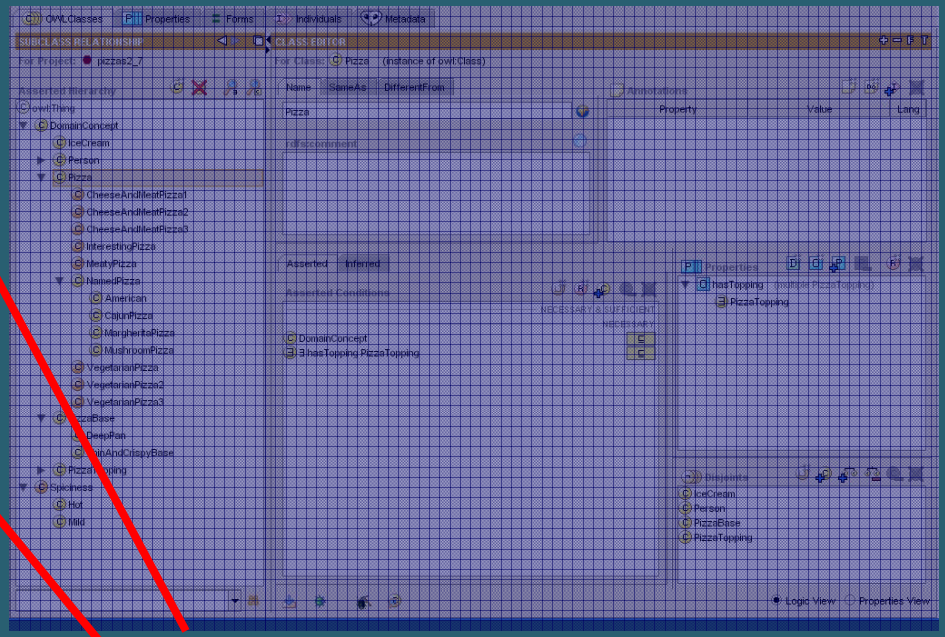

OWLClasses
Properties
Forms
Individuals
Metadata
SUBCLASS RELATIONSHIP
For Project: pizzas2_7
Asserted Hierarchy
owl:Thing
DomainConcept
IceCream
Person
Pizza
CheeseAndMeatPizza1
CheeseAndMeatPizza2
CheeseAndMeatPizza3
InterestingPizza
MeatyPizza
NamedPizza
American
CajunPizza
MargheritaPizza
MushroomPizza
VegetarianPizza
VegetarianPizza2
VegetarianPizza3
PizzaBase
DeepPan
ThinAndCrispyBase
PizzaTopping
Spiciness
Hot
Mild
CLASS EDITOR
For Class: Pizza (instance of owl:Class)
Name
SameAs
DifferentFrom
Pizza
rdfs:comment
Annotations
Property
Value
Lang
Asserted
Inferred
Asserted Conditions
NECESSARY & SUFFICIENT
NECESSARY
DomainConcept
∃ hasTopping PizzaTopping
Properties
hasTopping (multiple PizzaTopping)
PizzaTopping
Disjoints
IceCream
Person
PizzaBase
PizzaTopping
Logic View
Properties View

ClassesTab: Asserted Class Hierarchy



Subsumption hierarchy (superclass/subclass)

Structure as asserted by the ontology engineer

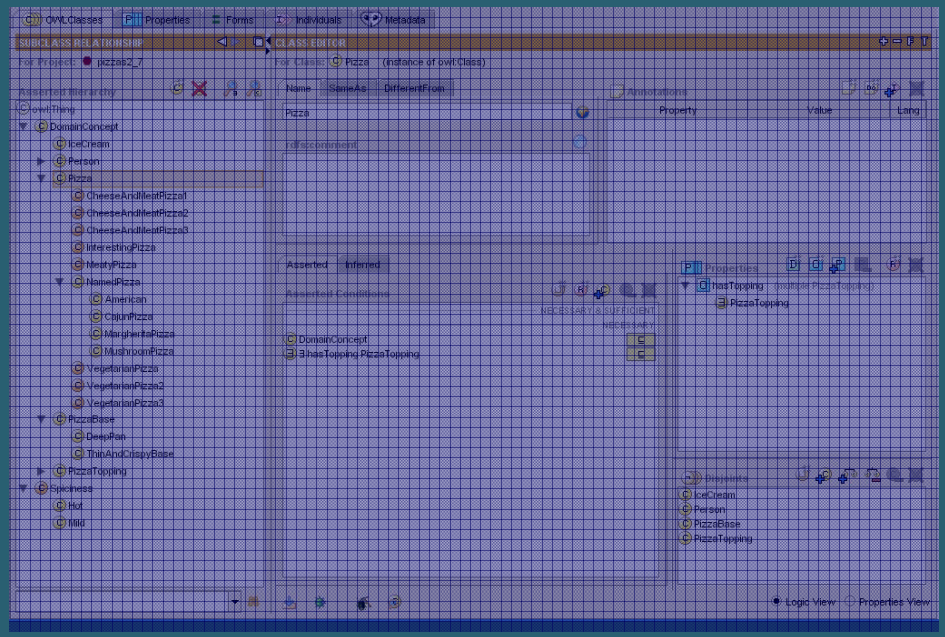


Create and Delete classes (actually subclasses!!)

Everything is a subclass of owl:Thing

Search for class

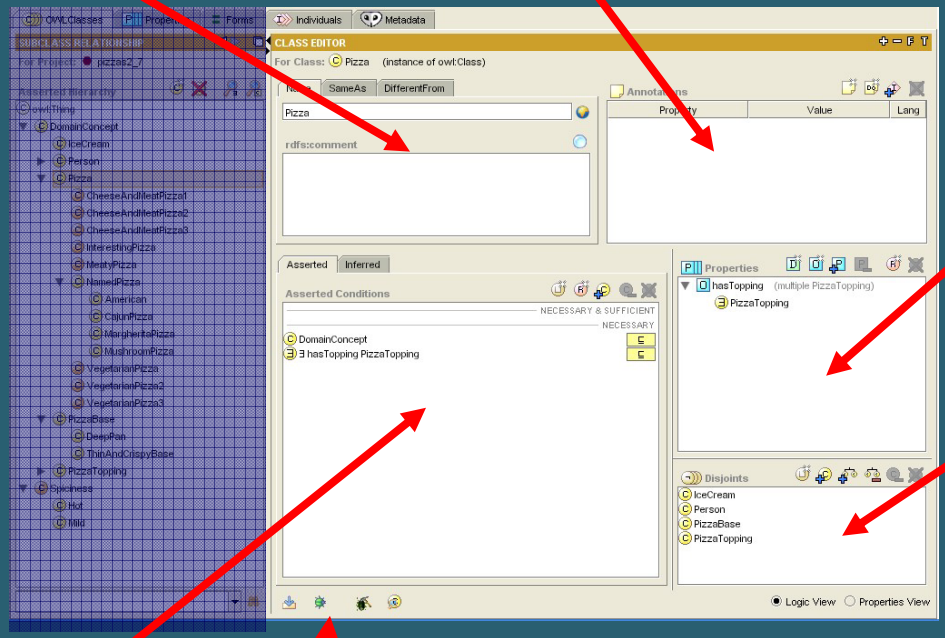
ClassesTab: Class Editor



ClassesTab: Class Editor

Class annotations (for class metadata)

Class name and documentation



Properties "available" to Class

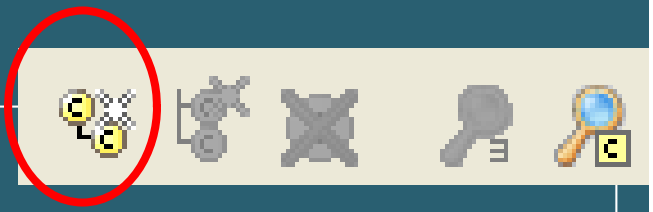
Disjoints widget

Conditions Widget

Class-specific tools (find usage etc)

Exercise: Create Classes

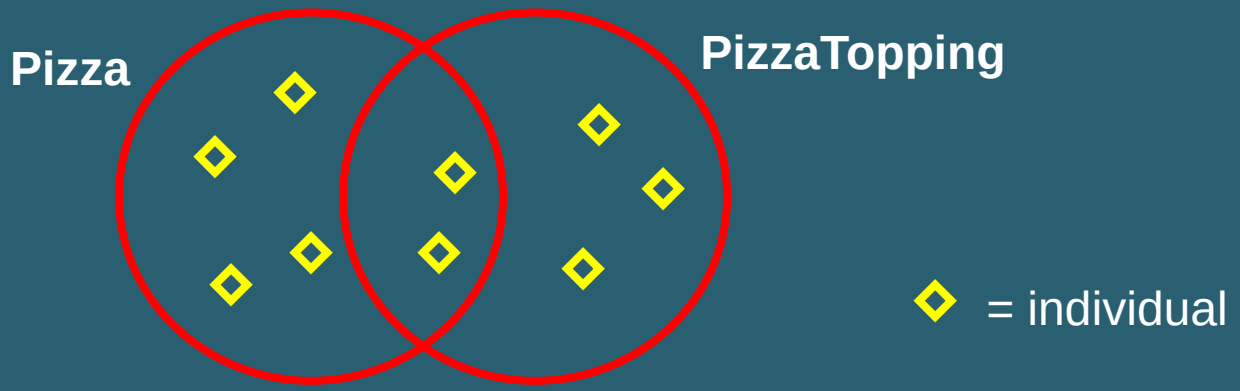
Create a new OWL ontology



1. **Click the "Create SubClass" button**
   (this is above the class hierarchy)
   A new class will be created as a subclass of **owl:Thing**
2. **Type in a new name "DomainConcept" over the default**
   (return updates the hierarchy)
3. **Document your class using the rdfs:comment field**
4. **Create another class called "Pizza" using the same method**
   You will notice that **Pizza** *has been created as a subclass of* **DomainConcept** *as this was the class selected when the button was pressed. You can also right-click any class and select "Create subClass"*
5. **Create two more subclasses of DomainConcept called "PizzaTopping" and "PizzaBase".**
   Any mistakes, use the "Delete Class" button next to "Create Class"

Disjointness

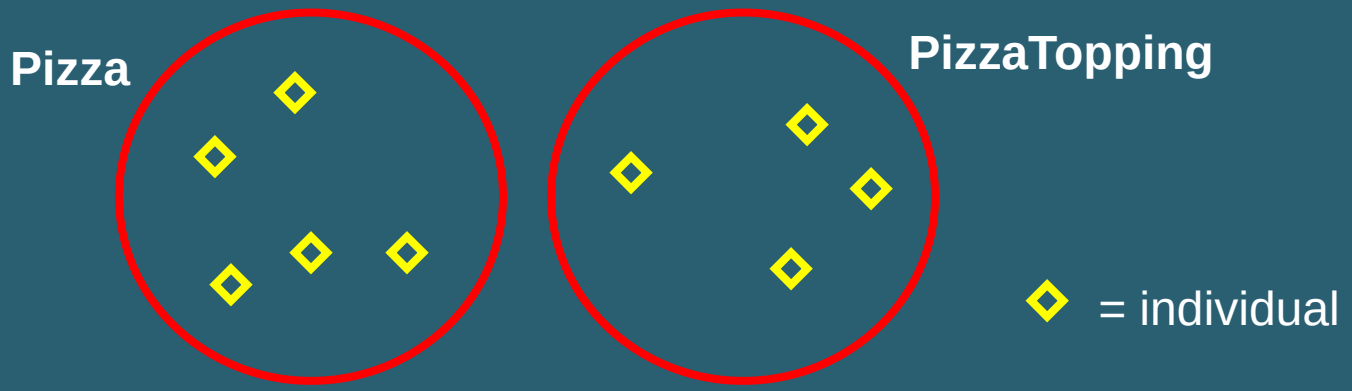
- OWL assumes that classes overlap



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

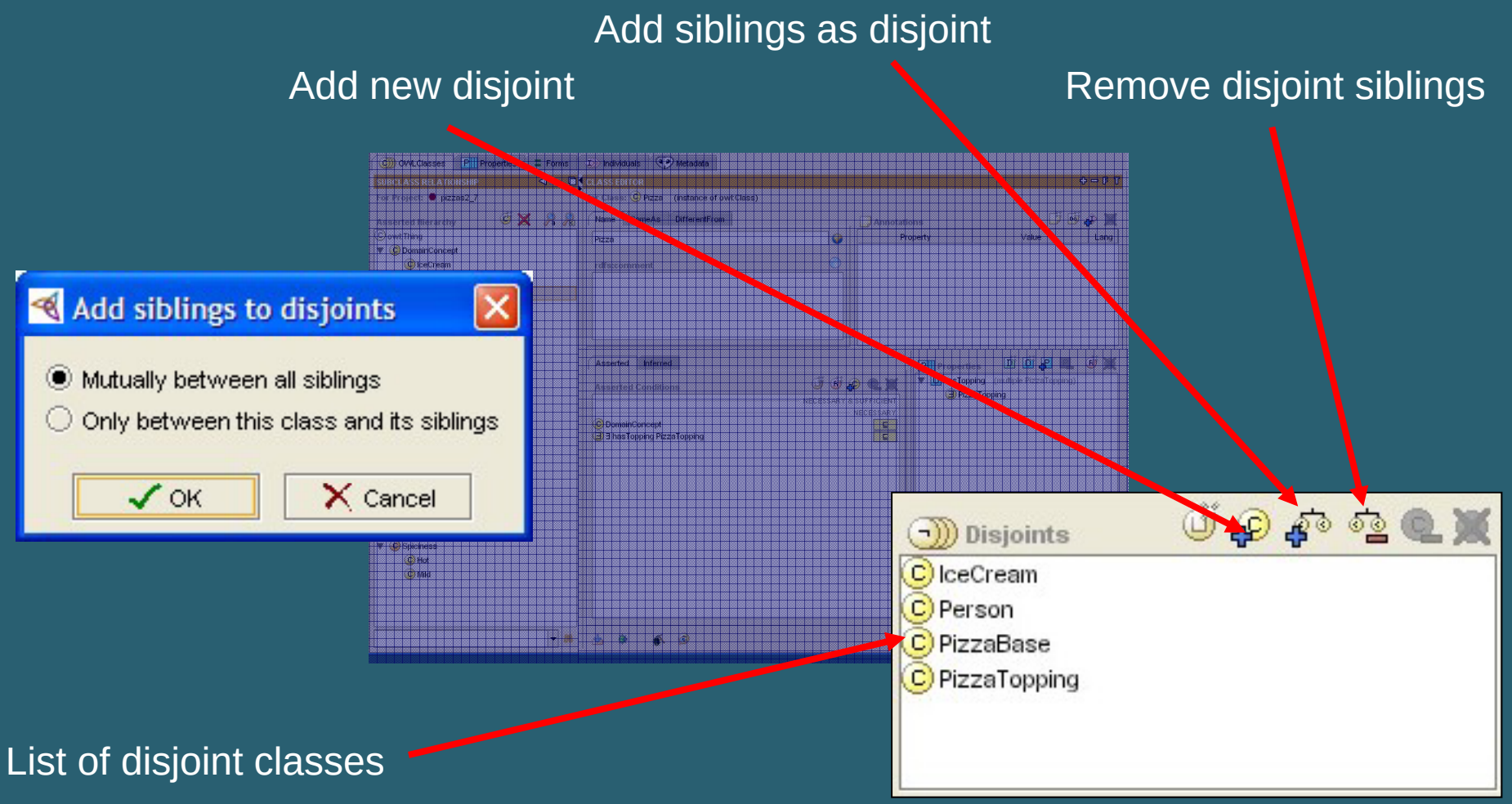
Disjointness

- If we state that classes are disjoint



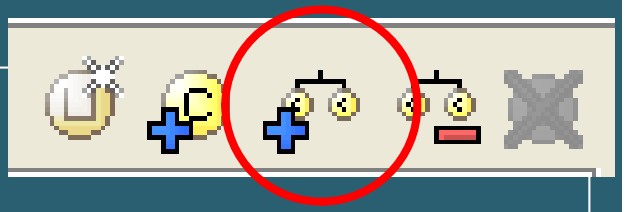
- This means an individual **cannot** be both a **Pizza** and a **PizzaTopping** at the same time
- We must do this **explicitly** in the interface

ClassesTab: Disjoints Widget



Exercise: Make Classes Disjoint

Start with your existing ontology



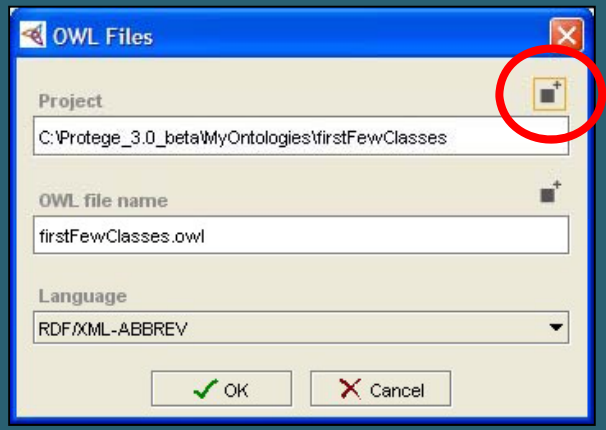
1. ***Select the* Pizza class**
   You will notice that the disjoints widget is empty
2. **Click the "Add all siblings…" button**
   The "Add siblings to disjoints dialog pops up
3. **Select the "Mutually between all siblings" option and OK**
   PizzaTopping *and* **PizzaBase** *appear in the disjoints widget*
4. **Select the PizzaTopping class**
   Pizza *and* **PizzaBase** *are already in the disjoints widget*
5. **Note that the same applies for PizzaBase**

Exercise: Save Your Work

OWL = easy to make mistakes – save regularly

1. **Select File → Save Project**
   A dialog (as shown) will pop up
2. **Select a file using a file selector by clicking the button on the top right**

   You will notice that there are 2 files 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



3. **Select OK**

Exercise: Create PizzaToppings

Start with your existing ontology

1. **Create subclasses of PizzaTopping:**
   CheeseTopping
   VegetableTopping
   MeatTopping
2. **Make these subclasses all disjoint from one another**
   (remember to chose "Mutually between all siblings" when prompted)
3. **Create subclasses of CheeseTopping:**
   MozzarellaTopping, ParmesanTopping
4. **Make these subclasses all disjoint from one another**
5. **Create subclasses of VegetableTopping and make them disjoint:**
   TomatoTopping, MushroomTopping
6. **Save**

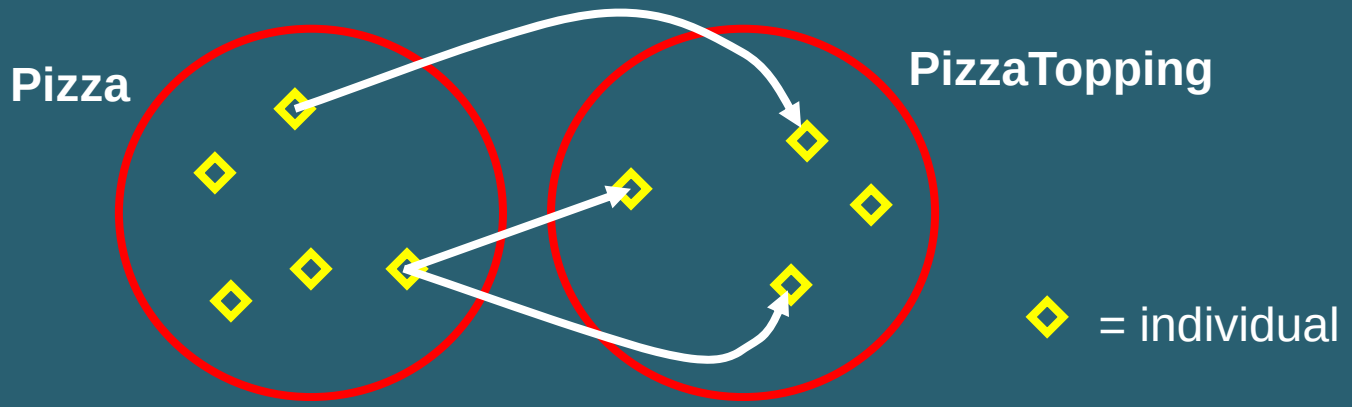
What have we got?

- We've created a tree of disjoint classes
- Disjoints are inherited down the tree
  eg something that is a **TomatoTopping** cannot be a **Pizza** because its superclass, **PizzaTopping**, is disjoint from **Pizza**

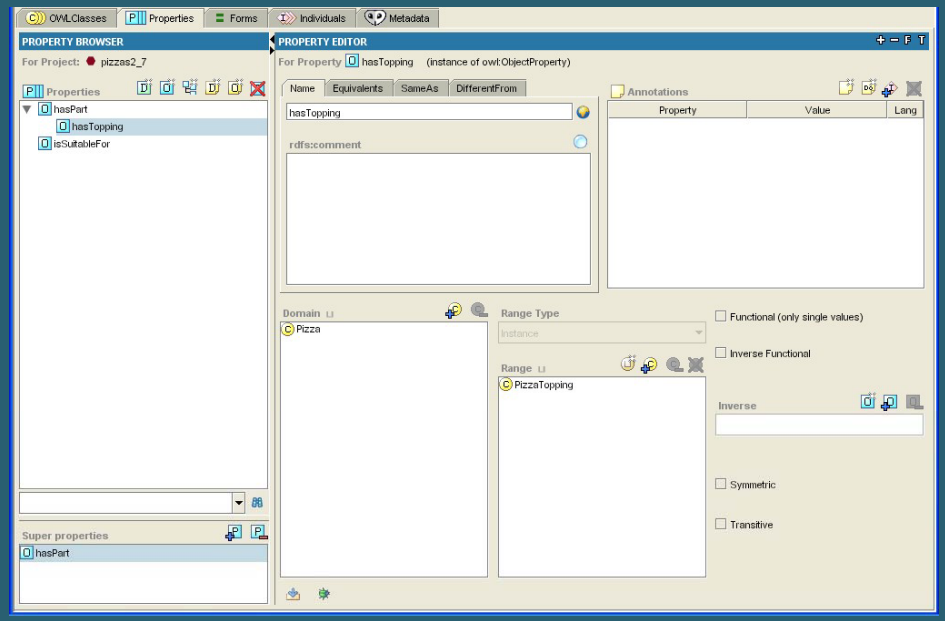
- You should now be able to select every class (except **DomainConcept**) and see its siblings in the disjoints widget

What are we missing?

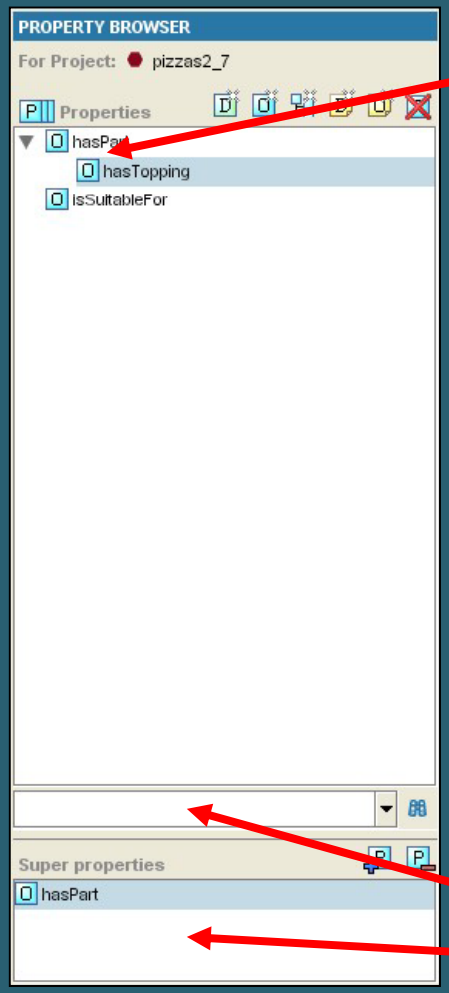
- This is not a semantically rich model
- Apart from "is kind of" and "is not kind of", we currently don't have any other information of interest
- We want to say more about **Pizza** individuals, such as their relationship with other individuals
- We can do this with properties



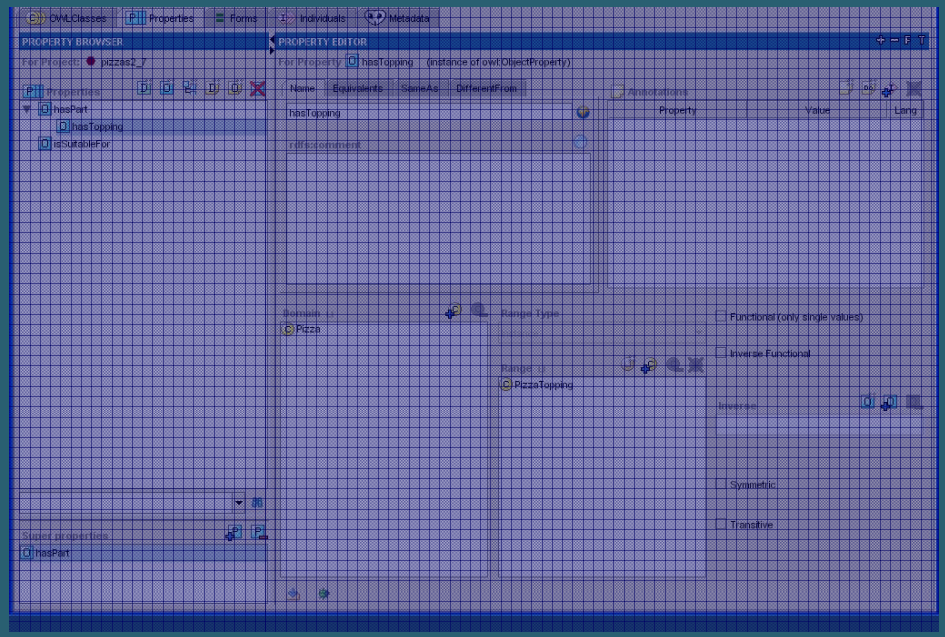
Properties Tab



Properties Tab: Property Browser



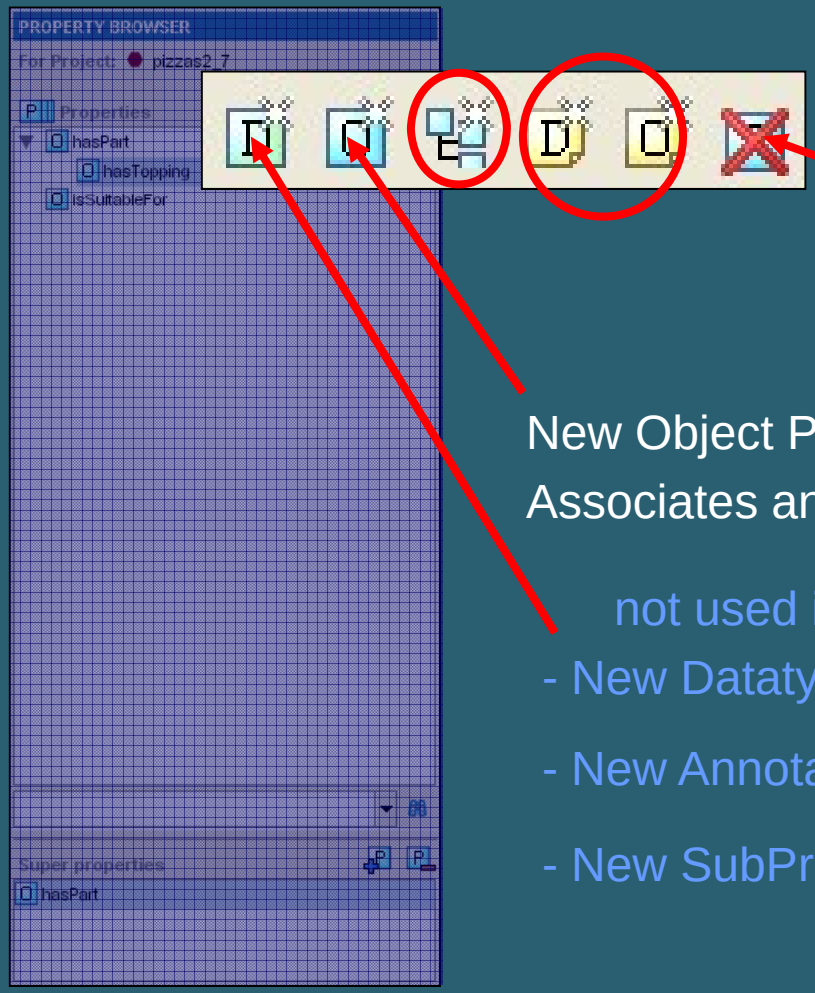
Properties can be in a hierarchy



Search for property

SuperProperties of the current selected

Properties Tab: Property Browser



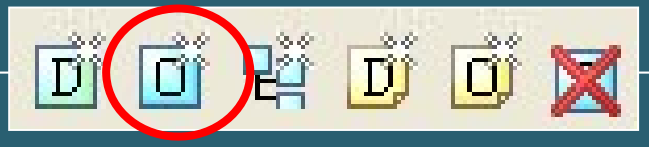
Delete Property

New Object Property:
Associates an individual to another individual

not used in the following exercise:

- New Datatype Property (String, int etc)
- New Annotation Properties for metadata
- New SubProperty – ie create “under” the current selection

Exercise: Create a Property



1. *Switch to the Properties tab*
   There are currently no properties, so the list is blank
2. *Create a new Object property using the button in the property browser*
3. *Call the new Property "hasTopping"*
4. *Create another Object Property called "hasBase"*
5. *Save*

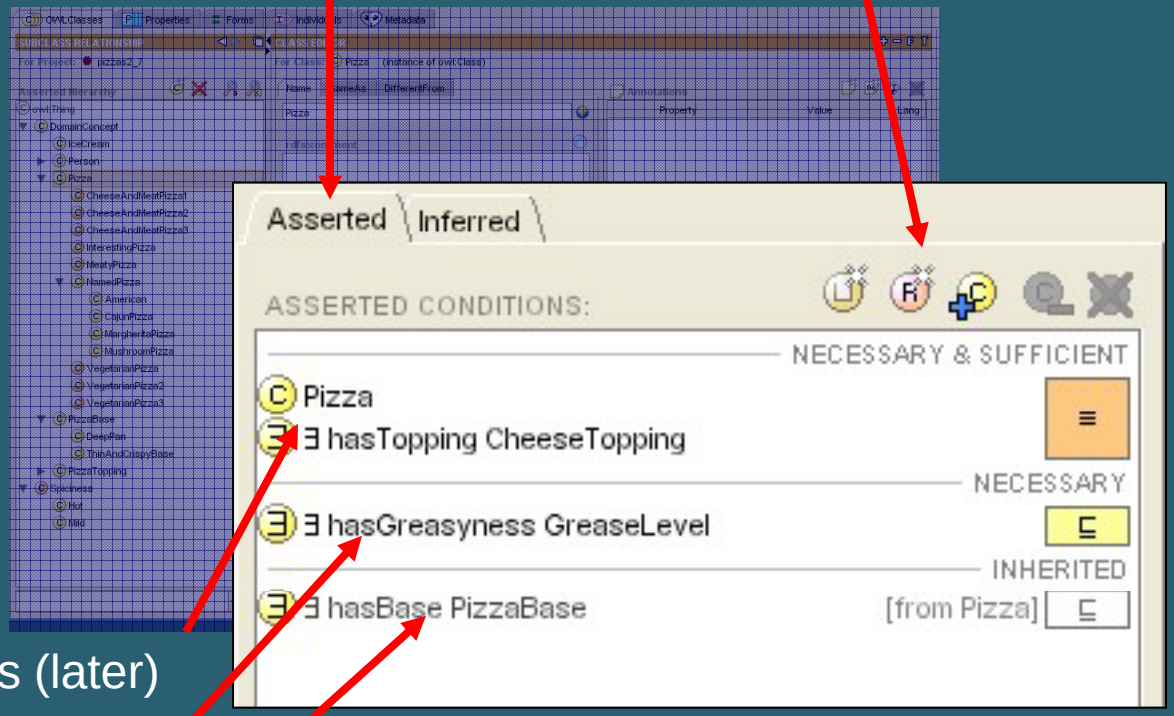
Associating Properties with Classes

- We now have two properties we want to use to describe **Pizza** individuals.
- To do this, we must go back to the **Pizza** class and add some further information
- This comes in the form of Restrictions (which are a type of Condition)

ClassesTab: Conditions Widget

Conditions asserted by the ontology engineer

Add different types of condition



Definition of the class (later)

Description of the class

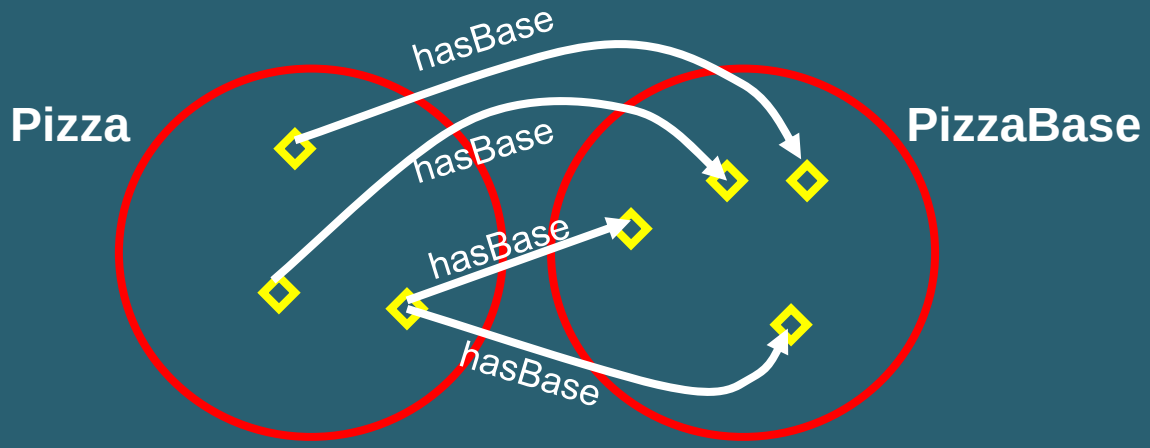
Conditions inherited from superclasses

Exercise: Create a Restriction

1. ***Switch to the OWL Classes tab***
2. ***Select* Pizza**
   Notice that the conditions widget only contains one item, **DomainConcept** *with a Class icon.*
   Superclasses show up in the conditions widget in this way
3. ***Click the "Create Restriction" button***
   A dialog pops up that we will investigate in a minute
4. ***Select "hasBase" from the Restricted Property pane***
5. ***Leave the Restriction type as "someValuesFrom"***
6. ***Type "PizzaBase" in the Filler expression editor***
7. ***Click OK***
   A restriction has been added to the Conditions widget

What does this mean?

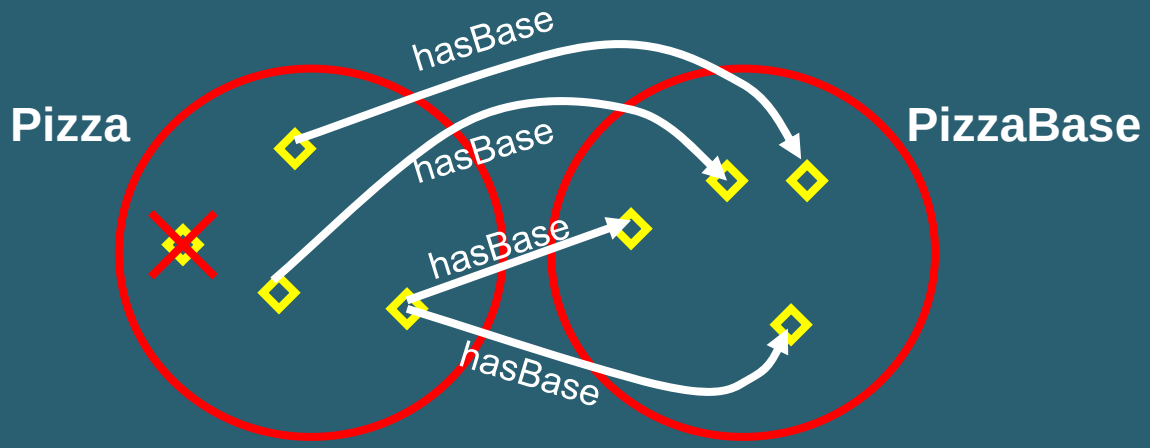
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



- **“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**”

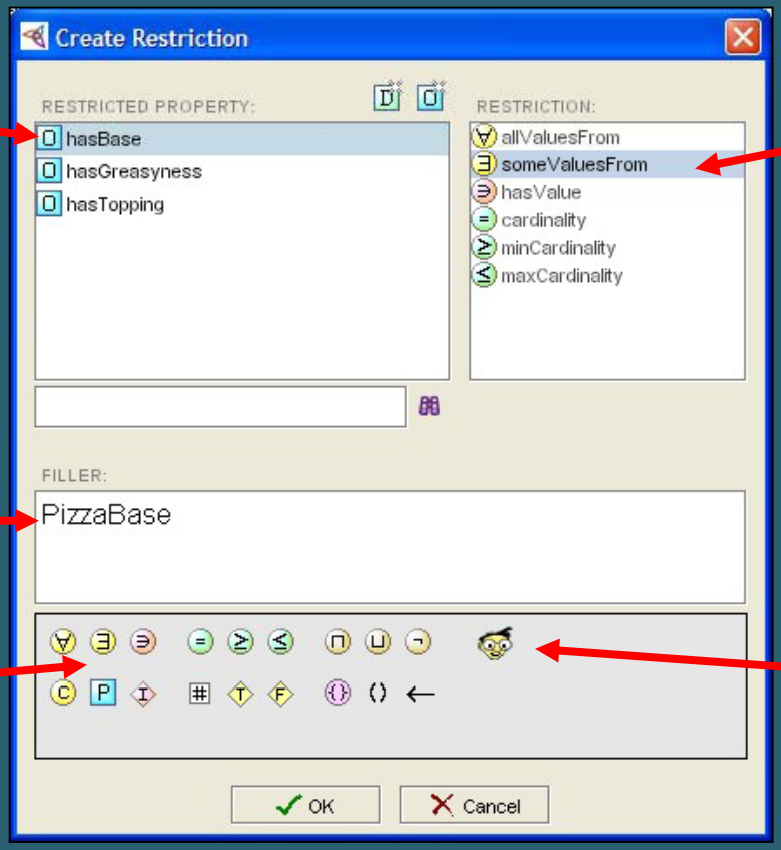
What does this mean?

- 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**"

Restrictions Popup



Restricted Property

Restriction Type

Filler Expression

Expression Construct Palette

Syntax check

Restriction Types

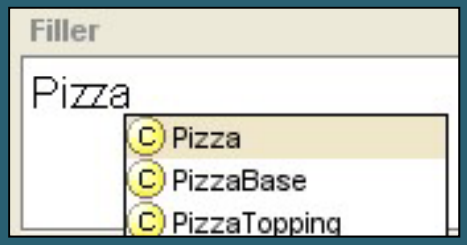
∃	Existential, someValuesFrom	“Some”, “At least one”
∀	Universal, allValuesFrom	“Only”
∋	**hasValue**	**“equals x”**
=	**Cardinality**	**“Exactly n”**
≤	**Max Cardinality**	**“At most n”**
≥	**Min Cardinality**	**“At least n”**

Exercise:
Another Existential Restriction

1. *Make sure* Pizza *is selected*
2. *Create a new Existential (SomeValuesFrom) Restriction with the hasTopping property and a filler of* PizzaTopping

When entering the filler, you have 2 shortcut methods rather than typing the entire classname:

1) enter a partial name and use Tab to autocomplete



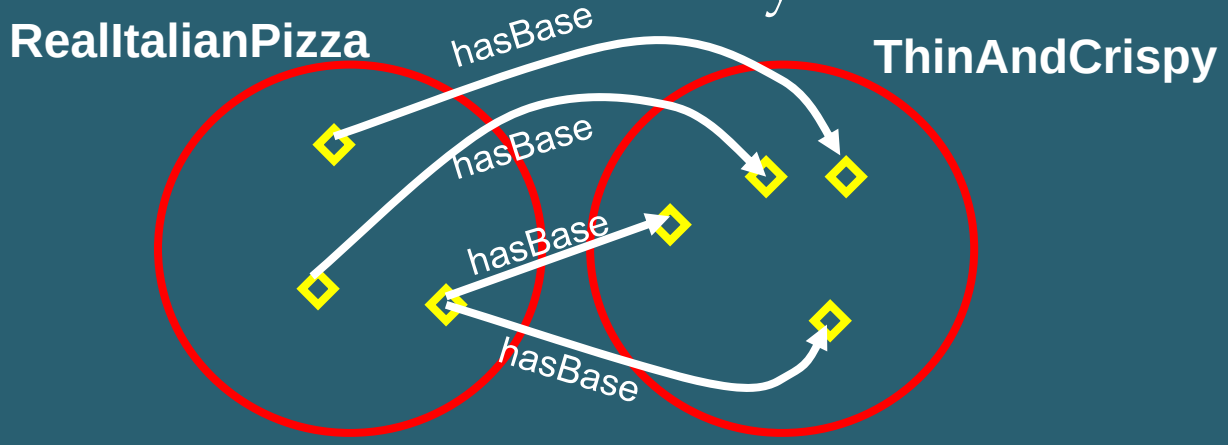
2) use the select Class button on the editor palette

Exercise:
Create a Universal Restriction

1. *Create 2 disjoint subclasses of* **PizzaBase** *called "ThinAndCrispy" and "DeepPan"*
2. *Create a subclass of* **Pizza** *called "RealItalianPizza"*
3. *Create a new Universal (AllValuesFrom) Restriction on* **RealItalianPizza** *with the hasBase property and a filler of* **ThinAndCrispy**

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”**

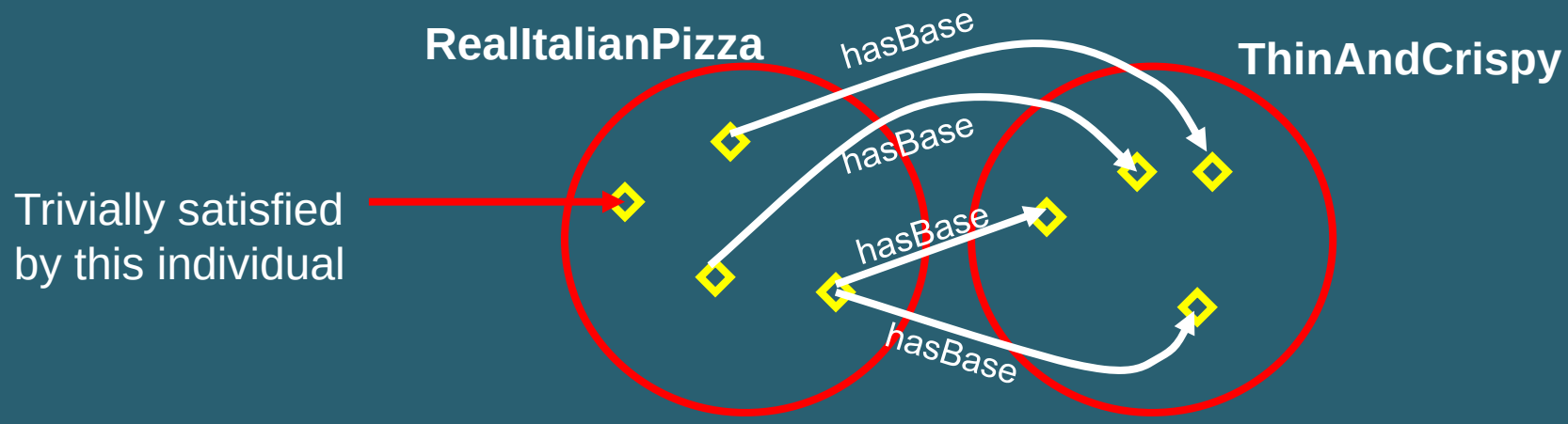
What does this mean?

- 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**"

Universal 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"
- **ie Universal Restrictions by themselves** do not **state "at least one"**

Summary

You should now be able to:

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

Exercise:
Create a MargheritaPizza

1. *Create a subclass of* **Pizza** *called* **NamedPizza**
2. *Create a subclass of* **NamedPizza** *called* **MargheritaPizza**
3. *Create a restriction to say that:*
   "Every MargheritaPizza must have at least one topping from TomatoTopping"
4. *Create another restriction to say that:*
   "Every MargheritaPizza must have at least one topping from MozzarellaTopping"

Exercise:
Create other pizzas

Extend the example by creating new types of pizzas that does not already exist. To do so:

1. *Add more topping ingredients as subclasses of PizzaTopping. Use the hierarchy, but be aware of disjoints*
2. *Create new classes that represent the new types of pizzas.*
3. *Express the fact how this new class is related to other types using a disjunction constraint.*
4. *Create restrictions on these pizzas to describe their ingredients*

Vegetarian Pizza

- Start from the pizzas2_1.owl available on the Labs page:
  - Select File → Build New Project → OWL Files and chose pizzas2_1.owl
- Create a new pizza called "VegetarianPizza" under Pizza

  make this disjoint from its siblings as we have been doing

- Select MargheritaPizza.

  you will notice that it only has a single parent, NamedPizza
- Add VegetarianPizza as a new parent using the conditions widget "Add Named Class" button

  notice that MargheritaPizza now occurs in 2 places in the asserted hierarchy we have asserted that MargheritaPizza has 2 parents

Reasoning

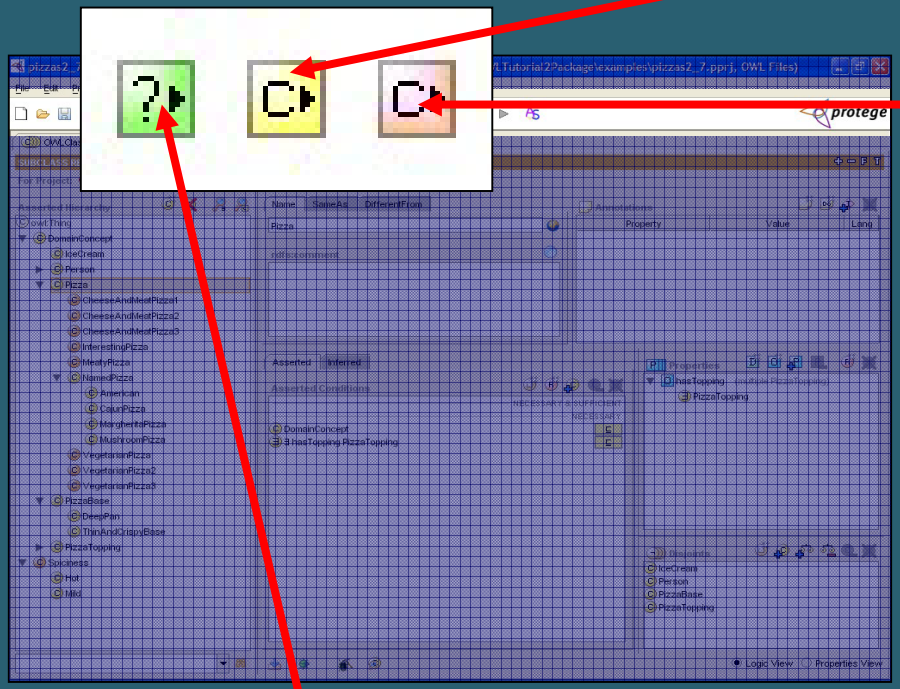
- We'd like to be able to check the logical consistency of our model
- We'd also like to make automatic inferences about the subsumption hierarchy. A process known as classifying
  - ie Moving classes around in the hierarchy based on their logical definition
- Generic software capable of these tasks are known as reasoners (although you may hear them being referred to as Classifiers)
- Pellet is a reasoner

Running Pellet

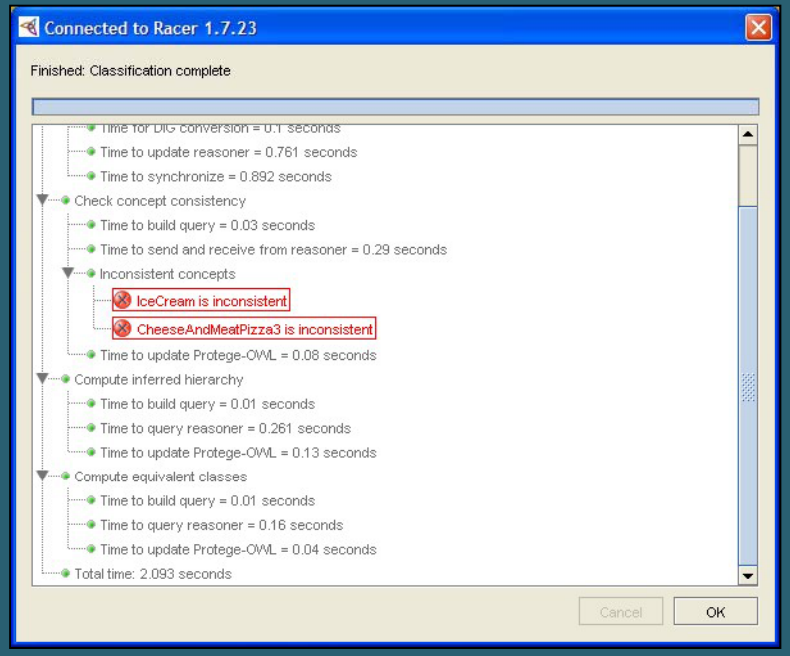
1. *Download and extract Pellet from the Labs page*
2. Execute LaunchServer.bat to launch the pellet server

Classifying

Classify taxonomy (and check consistency)

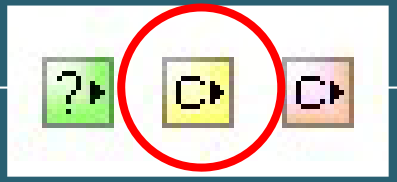


Compute inferred types (for individuals)



Just check consistency (for efficiency)

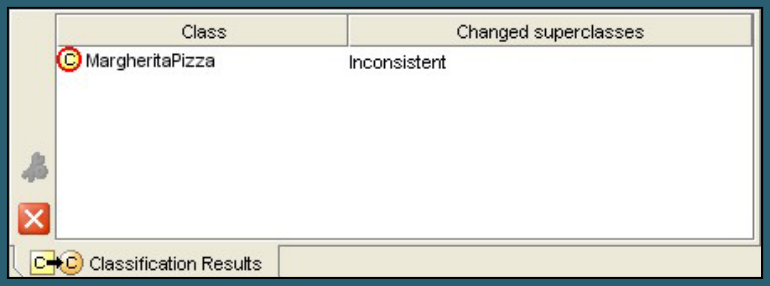
Reasoning about our Pizzas

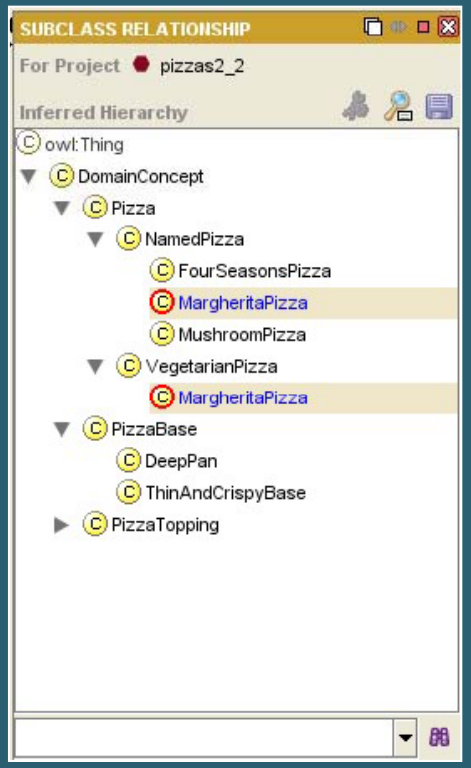


1. *Classify your ontology*

   You will see an inferred hierarchy appear, which will show any movement of classes in the hierarchy
   You will also see a results window appear at the bottom of the screen which describes the results of the reasoner

   MargheritaPizza turns out to be inconsistent – why?

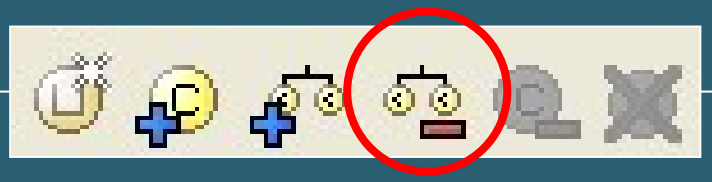




Why is MargheritaPizza inconsistent?

- We are asserting that a **MargheritaPizza** is a subclass of two classes we have stated are disjoint
- The disjoint means nothing can be a **NamedPizza** and a **VegetarianPizza** at the same time
- This means that the class of **MargheritaPizza**s can never contain any individuals
- The class is therefore inconsistent

Attempting again

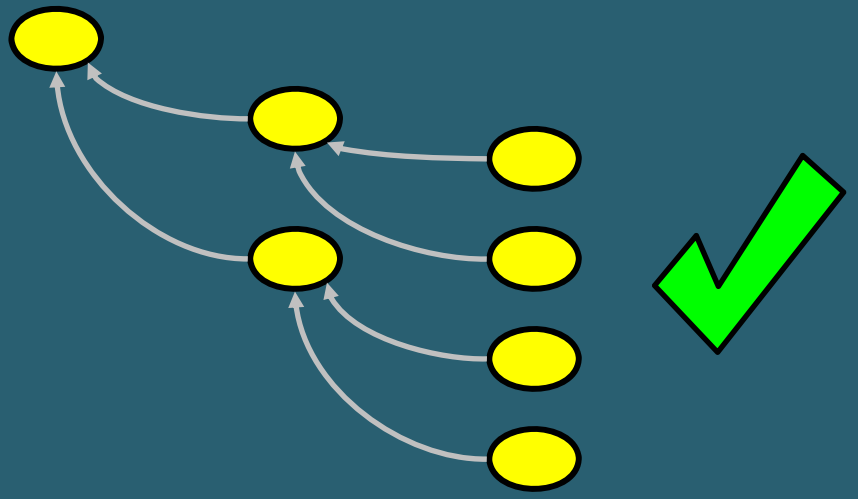


1. *Close the inferred hierarchy and classification results pane*
2. *Remove the disjoint between* **VegetarianPizza** *and its siblings*
   When prompted, choose to remove only between this class and its siblings
3. *Re-Classify your ontology*

   This should now be accepted by the reasoner with no inconsistencies

Primitive Classes

- All classes in our ontology so far are Primitive
- We describe primitive pizzas
- Primitive Class = only Necessary Conditions
- They are marked as yellow in the class hierarchy



Defined Classes

- Have a definition. That is *at least one* Necessary and Sufficient condition
- Are marked in orange in the interface
- Classes, all of whose individuals satisfy this definition, can be inferred to be subclasses
- Reasoners can perform this inference

Describing a MeatyPizza

Start with pizzas2_3.owl, *close the reasoner panes*

1. *Create a subclass of* **Pizza** *called* **MeatyPizza**
   Don't put in the disjoints or you'll get the same problems as before
   In general, defined classes are not disjoint
2. *Add a restriction to say:*
   "Every **MeatyPizza** *must have at least one meat topping"*
3. *Classify your ontology*
   What happens?

Defining a MeatyPizza

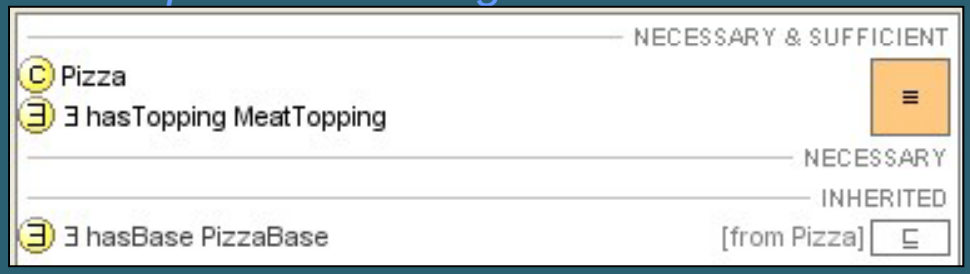
1. *Click and drag your ∃ hasTopping* **MeatTopping** *restriction from* “Necessary” *to* “Necessary & Sufficient”

   The **MeatyPizza** *class now turns orange, denoting that it is now a defined class*

2. *Click and drag the* **Pizza** *Superclass from* “Necessary” *to* “Necessary & Sufficient”

   Make sure when you release you are on top of the existing restriction otherwise you will get 2 sets of conditions.

   You should have a single orange icon on the right stretching across both conditions like this…



3. *Classify your ontology*

   What happens?

How do we Define a Vegetarian Pizza?

- 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

Defining a Vegetarian Topping

Start with pizzas2_5.owl

1. *Create a subclass of* **PizzaTopping** *called* **VegetarianTopping**
2. *Click "Create New Expression" in the Conditions Widget*
   Type in each of the top level **PizzaToppings** *that are not meat or fish (ie* **DairyTopping**, **FruitTopping** *etc) and between each, type insert the union symbol*
3. *Press Return when finished*
   you have created an anonymous class described by the expression
4. *Make this a defined class by moving both conditions from the "Necessary" to the "Necessary & Sufficient" conditions*
5. *Classify your ontology*

Vegetarian Pizza attempt 2

1. *Select* **MargheritaPizza** *and remove* **VegetarianPizza** *from its superclasses*
2. *Select* **VegetarianPizza** *and create a restriction to say that it "only has toppings from* **VegetarianTopping***"*
3. *Make this a defined class by moving all conditions from* "Necessary" *to* "Necessary & Sufficient"
   Make sure when you release you are on top of the existing restriction otherwise you will get 2 sets of conditions.
   You should have a single orange icon on the right stretching across both conditions
4. *Classify your ontology*
   What happens?

Open World Assumption

- The reasoner does not have enough information to classify pizzas under **VegetarianPizza**
- Typically several Existential restrictions on a single property with different fillers – like primitive pizzas
- Existential should be paraphrased by "amongst other things…"
- We need closure for the given 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**

Closing Pizza Descriptions

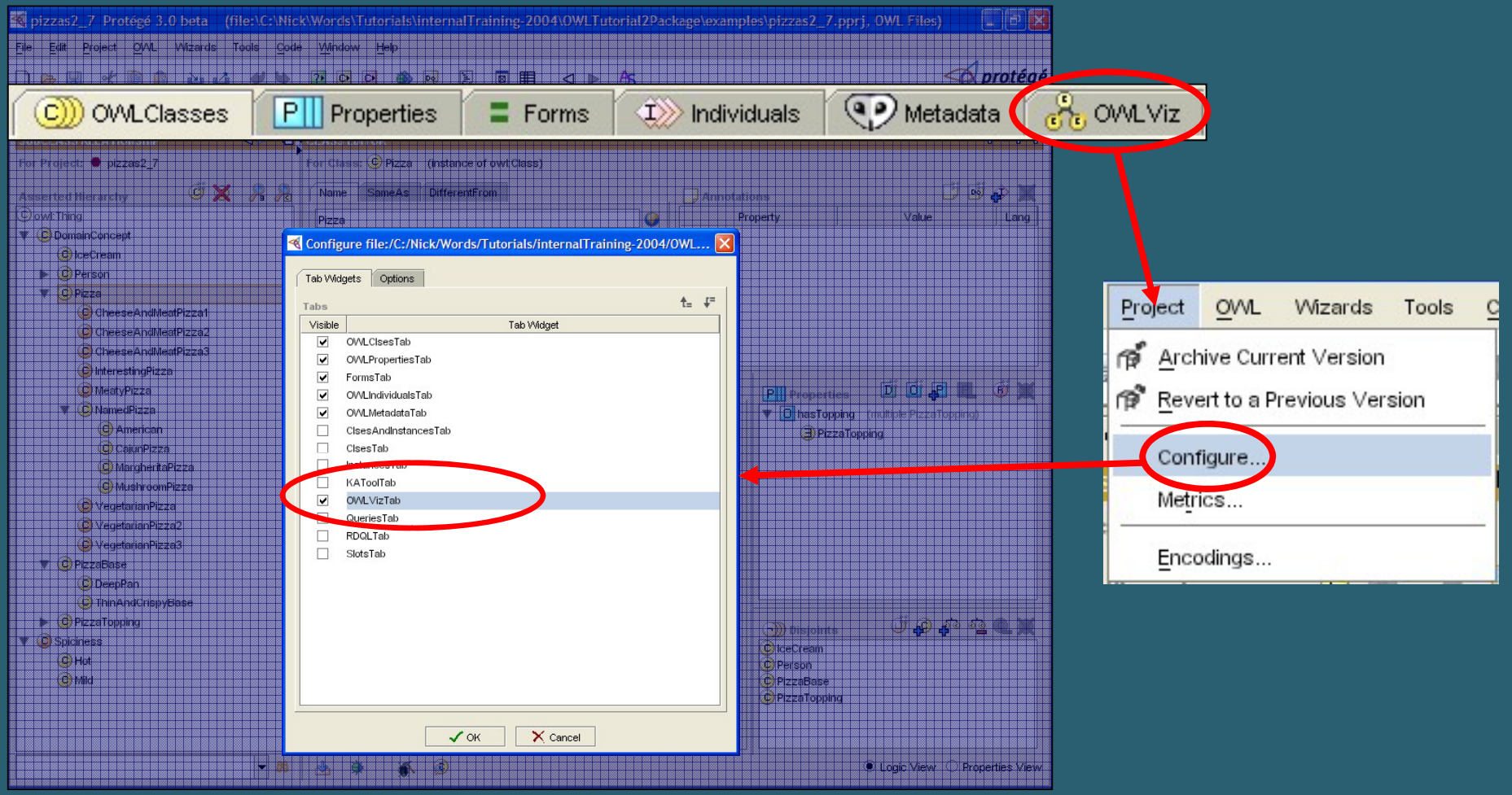
Start with pizzas2_7.owl

1. *Select* **MargheritaPizza**
2. *Create a Universal Restriction on the hasTopping property with a filler of "**TomatoTopping ⊔ MozzarellaTopping**"*
   Remember, you can type "or" to achieve this, or you can use the expression palette
3. *Close your other pizzas*
   Each time you need to create a filler with the union of all the classes used on the hasTopping property (ie all the toppings used on that pizza)
4. *Classify your ontology*
   Finally, the defined class **VegetarianPizza** *should subsume any classes that only have vegetarian toppings*

Viewing our Hierarchy Graphically

- Using OWLViz
- Requires Graphviz (available on the Labs page)
- Download it and install it (we will extra configurations in the next steps if required)

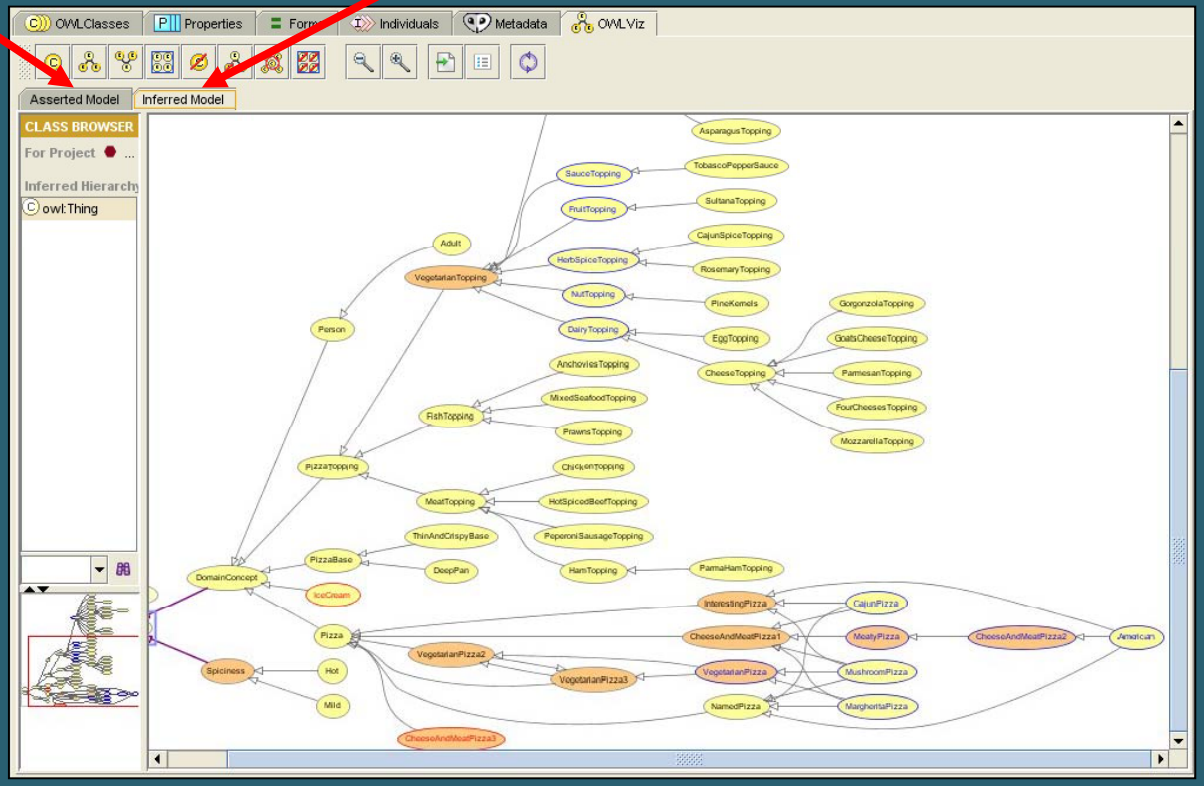
Viewing our Hierarchy Graphically



OWLViz Tab

View Asserted Model

View Inferred Model



Using the created Ontology

The pizza Finder application

Pizza Finder

- Download the **Pizza Finder Application**
- Run the application using your ontology

Exercise from the course

Exercise

Create an OWL ontology that models the following concepts:

1. There should be three classes: Customer, Shop and Product.
2. Customer and Shop should be equipped with properties name (xsd:string) and email (xsd:string), which are equivalent to foaf:name and foaf:mbox.
3. Each Product should have an order number (xsd:int). An order number can be unambiguously assigned to a Product.
4. A Shop should have a property sells (range: Product) and a Product should have a property soldBy (range: Shop) respectively.
5. Instances of class Shop that sell more than 100 products should belong to a new class BigShop.
6. A Product must not be a Customer.
7. Instances that are both, Shop and Customer should belong to a class PurchaseAndSale.

Note: Import the foaf ontology

References

- Slides based on the tutorial prepared by N. Drummond, M. Horridge
  http://www.cs.man.ac.uk/~drummond/cs646

- Software / resources / community at:
  - http://protege.stanford.edu/