Semantic Web Engineering

Gerald Reif
reif@ifi.unizh.ch

Fr. 10:15-12:00, Room 2.A.10

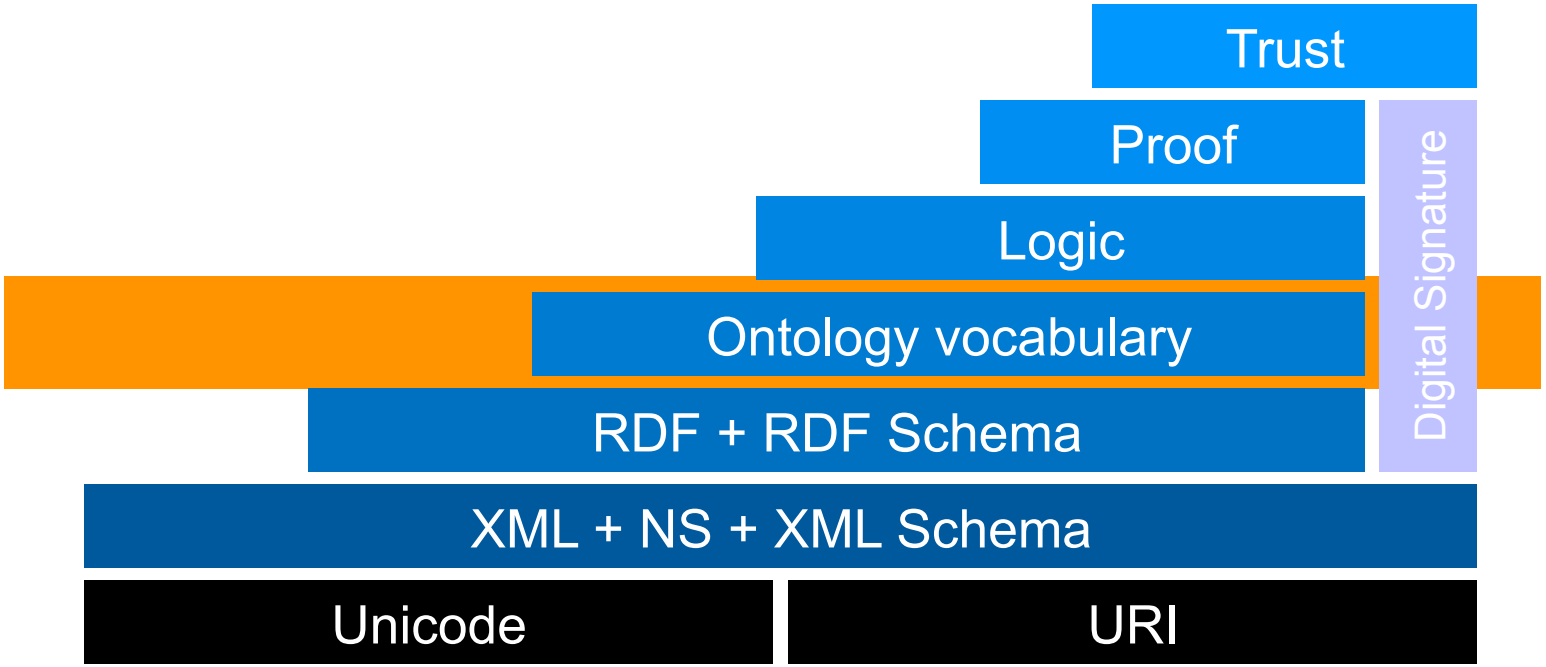
University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Organizational Stuff

- Today only ~45 Min lecture
- No lecture on November 12, 2010
- Lecture on November 29, 2010 takes place in Andreasstrasse AND 3-48

Web Ontology Language



Definition: Ontology

An ontology is an explicit and formal specification of a shared conceptualization.

- This definition includes:
  - The specified concepts must be defined explicitly.
  - The concepts are formally specified.
  - There must be a shared agreement on the ontology.
  - There might be more than one ontology for a domain.
- In general, an ontology describes formally the vocabulary to talk about a domain of discourse.
- The ontology consists of a finite list of terms and relationships between these terms.

Requirements for Ontology Languages 1/2

- Well-defined syntax
  - Necessary for machine processing of information; known from programming languages.
- Formal Semantics
  - A formal Semantics describes the meaning of knowledge precisely. Achieved by using logic.
- Convenience of expression
  - Provide a simple syntax to represent the logic constraints.

Requirements for Ontology Languages 2/2

- Efficient reasoning support
  - The use of formal semantics allows to reason about the knowledge:
    - Check the consistency of an ontology.
    - Check for unintended relationships.
    - Automatically classify instances in classes.
  - Checks like the preceding ones are valuable for
    - designing large ontologies, where multiple authors are involved
    - integrating and sharing ontologies from various sources
- Sufficient expressive power
  - Use as much logic to be able to express the wanted constraints, but still keep it computational complete.

Limitations of RDF and RDF Schema

- RDF/RDFS allow to define Classes, properties, class and property hierarchies, and domain and range restrictions.
- Several Features are missing:
  - *Cardinality constraints* on properties.
    - A `Person` has exactly two `Parents`.
  - *Local scope* of properties. In RDFS we cannot declare range restrictions that apply to some classes only.
    - A cow eats only plants, while other animals my eat meat, too.
  - Disjointness of classes.
    - `Male` and `Female` are *disjoint*.
  - Boolean combination of classes. New classes are defined by combining other classes using *union*, *intersection*, and *complement*.
    - The class `Person` is defined by the disjoint union of the classes `Male` and `Female`.
  - Special characteristics of properties.
    - To declare a property *transitive* ("greater than"), *unique* ("is mother of"), *inverse* ("eats" and "is eaten by").

The Web Ontology Language OWL

- The limitations of RDF lead to research on more advanced ontology languages.
  - DAML: DARPA Agent Markup Language; US initiative
  - OIL: Ontology Inference Layer; European initiative.
  - DAML+OIL joint US/European initiative.
- The W3C Web Ontology Working Group
  - DAML+OIL was the starting point for the group.
- Web Ontology Language (OWL)
  - W3C recommendation since 10 February 2004
  - http://www.w3.org/TR/owl-semantics/

OWL Related W3C Documents

- The OWL Overview gives a simple introduction to OWL by providing a language feature listing with very brief feature descriptions; http://www.w3.org/TR/owl-features/
- The OWL Guide demonstrates the use of the OWL language by providing an extended example. It also provides a glossary of the terminology used in these documents; http://www.w3.org/TR/owl-guide/
- The OWL Reference gives a systematic and compact (but still informally stated) description of all the modeling primitives of OWL; http://www.w3.org/TR/owl-ref/
- The OWL Semantics and Abstract Syntax document is the final and formally stated **normative** definition of the language; http://www.w3.org/TR/owl-semantics/
- The OWL Web Ontology Language Test Cases document contains a large set of test cases for the language; http://www.w3.org/TR/owl-test/
- The OWL Use Cases and Requirements document contains a set of use cases for a web ontology language and compiles a set of requirements for OWL. http://www.w3.org/TR/webont-req/

Combining OWL with RDF Schema

- Ideally, OWL would extend RDF Schema
  - Consistent with the layered architecture of the Semantic Web
- **But** simply extending RDF Schema would work against obtaining expressive power and efficient reasoning
  - Combining RDF Schema with logic leads to uncontrollable computational properties

Three Species of OWL

- W3C'sWeb Ontology Working Group defined OWL as three different sublanguages:
  - OWL Full
  - OWL DL
  - OWL Lite
- Each sublanguage geared toward fulfilling different aspects of requirements

OWL Full

- It uses all the OWL languages primitives
- It allows the combination of these primitives in arbitrary ways with RDF and RDF Schema
  - e.g. impose a cardinality constraint on the class of all classes, essentially limiting the number of classes that can be described in an ontology.
- OWL Full is fully upward-compatible with RDF, both syntactically and semantically
- OWL Full is so powerful that it is undecidable
  - No complete (or efficient) reasoning support

OWL DL

- OWL DL (Description Logic) is a sublanguage of OWL Full that restricts application of the constructors from OWL and RDF
  - Application of OWL's constructors' to each other is disallowed
  - Therefore it corresponds to a well studied description logic
- OWL DL permits efficient reasoning support
- **But** we lose full compatibility with RDF:
  - Not every RDF document is a legal OWL DL document.
  - Every legal OWL DL document is a legal RDF document.

OWL Lite

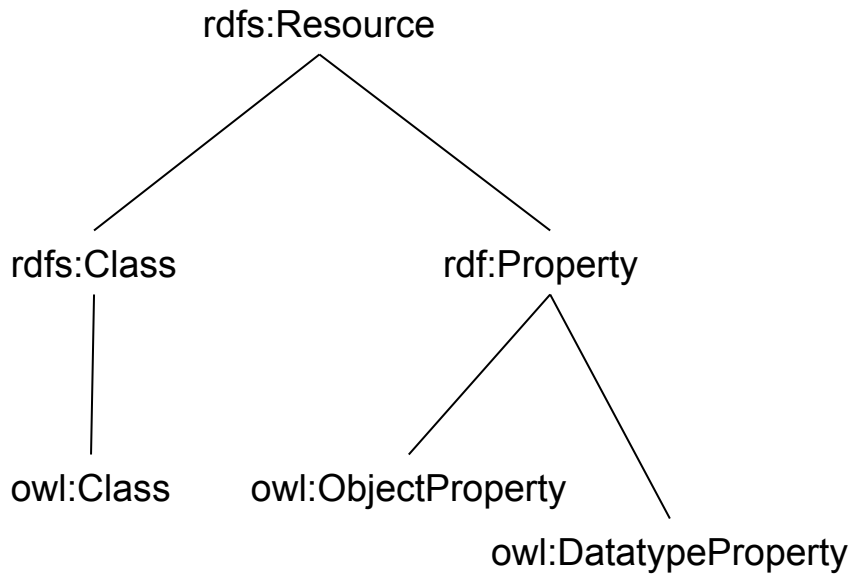
- An even further restriction limits OWL DL to a subset of the language constructors
  - E.g., OWL Lite excludes enumerated classes, disjointness statements, and arbitrary cardinality.
- The advantage of this is a language that is easier to
  - grasp, for users
  - implement, for tool builders
- The disadvantage is restricted expressivity

Upward Compatibility between OWL Species

- Every legal OWL Lite ontology is a legal OWL DL ontology.
- Every legal OWL DL ontology is a legal OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

OWL Compatibility with RDF Schema

- All varieties of OWL use RDF for their syntax
- Instances are declared as in RDF, using RDF descriptions
- and typing information OWL constructors are specialisations of their RDF counterparts



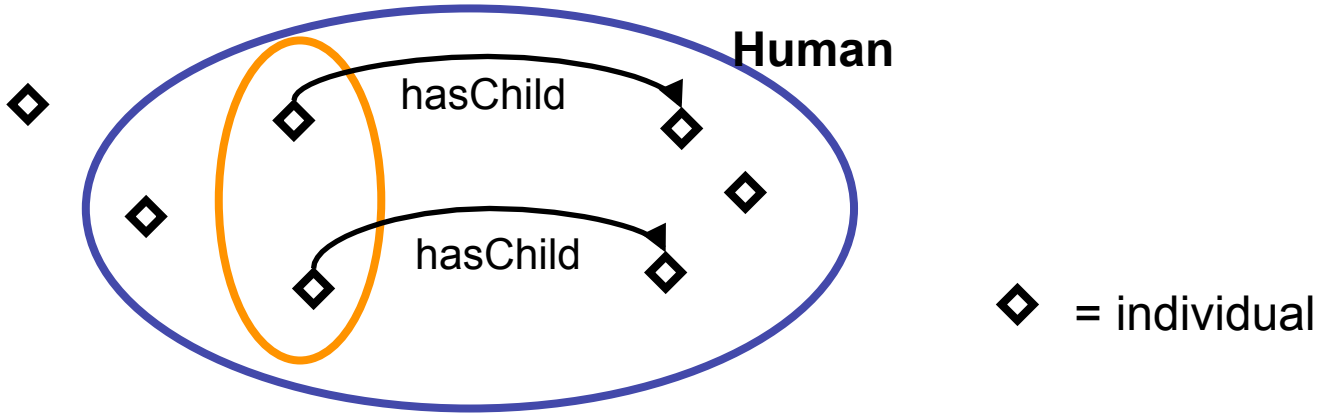
What are Description Logics?

- A family of logic based Knowledge Representation formalisms
- Historic background
  - Practical: Semantic networks, frame systems
  - Formal: First Order Logic (FOL)
- Decidable fragment of FOL
- Describe domain in terms of concepts (classes), roles (relationships) and individuals
- Provision of inference services
  - Sound and complete decision procedures for key problems
  - Implemented systems (highly optimized)
- OWL Web ontology language based on SHIQ DL

DL Example

- Every human has human children:

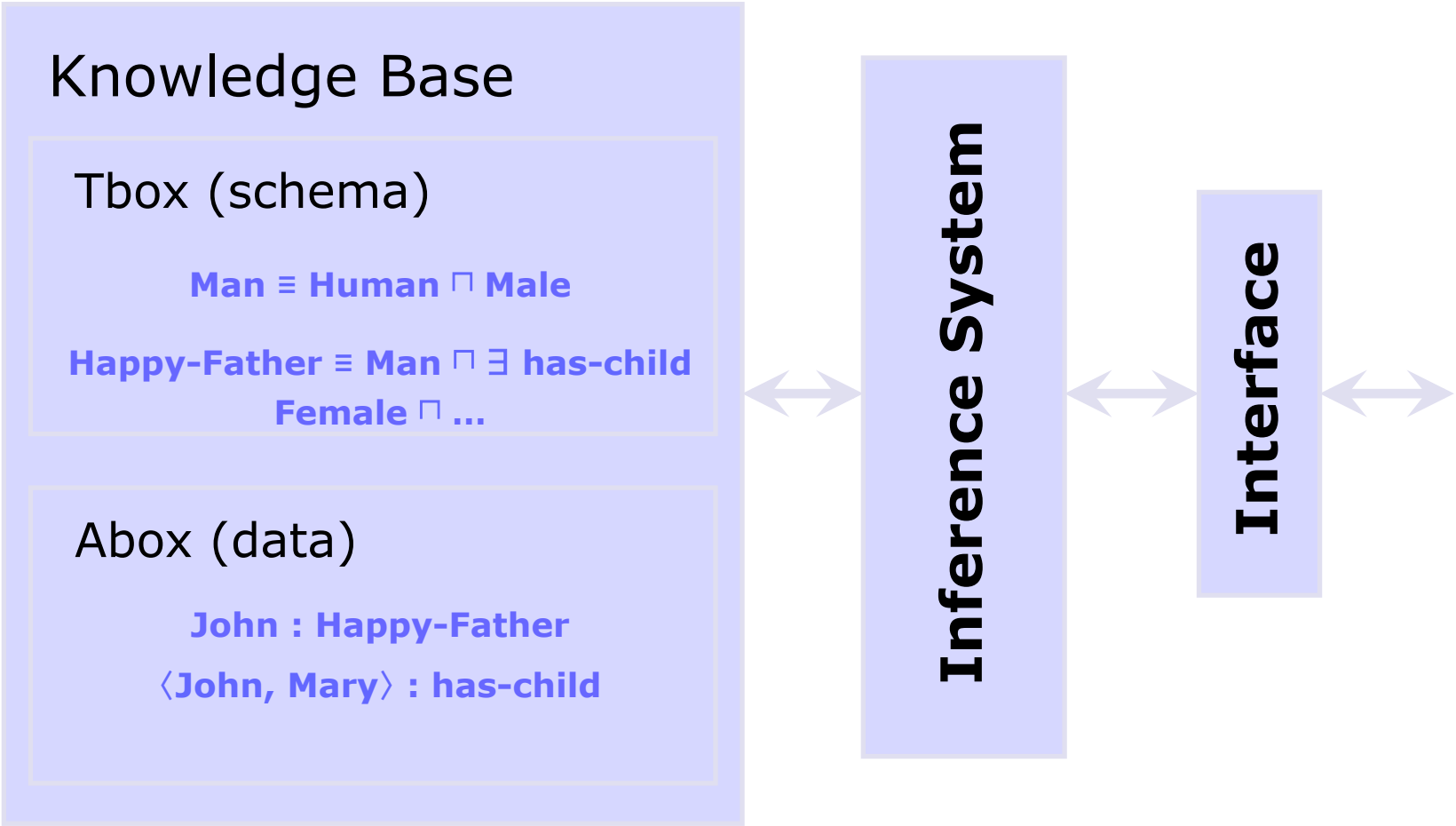
  $\exists hasChild.Human \sqsubseteq Human$



- A man that is married to a doctor and has at least five children, all of whom are professors:

  $Human \sqcap \neg Female \sqcap \exists married.Doctor \sqcap (\geq 5\ hasChild) \sqcap \forall hasChild.Professor$

DL Architecture



Necessary and Sufficient Condition

- Necessary condition:

  $\mathtt{Dog} \sqsubseteq \mathtt{Animal}$ (Dog is a subclass of Animal)

  - Being a *Animal* is a necessary condition of being a *Dog*, but is not sufficient.

- Sufficient condition:

  $\mathtt{CarOwner} \equiv (\mathtt{Person} \sqcap \exists \mathtt{owns.Car})$ (equivalent class)

  - *CarOwners* must be a *Person* who *owns* a *Car* (necessary conditions as above), but in addition, any *Person* who *owns* a *Car* must also be a *CarOwner*.

OWL as DL: Class Constructors

Constructor	DL Syntax	Example
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$	Human $\sqcap$ Male
unionOf	$C_1 \sqcup \ldots \sqcup C_n$	Cat $\sqcup$ Dog
complementOf	$\neg C$	$\neg$ Male
oneOf	$\{x_1\} \sqcup \ldots \sqcup \{x_n\}$	{john} $\sqcup$ {mary}
allValuesFrom	$\forall P.C$	$\forall$ hasSon.Male
someValuesFrom	$\exists P.C$	$\exists$ owns.Car
maxCardinality	$\leq n\ P$	$\leq$ 2 hasChild
minCardinality	$\geq n\ P$	$\geq$ 1 hasChild

- XML Schema datatypes

OWL Syntax

E.g., Person ⊓ ∀ hasChild.Doctor ⊓ ∃ hasChild.Doctor

DL Syntax

intersectionOf(Person
restriction(hasChild allValluesFrom Doctor)
(hasChild someValuesFrom Doctor))

Abstract Syntax

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Person"/>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#Doctor"/>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasChild"/>
      </owl:onProperty>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasChild"/>
      </owl:onProperty>
      <owl:someValuesFrom rdf:resource="#Doctor"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

OWL/XML Syntax

OWL 2 Syntaxes

Functional-Style Syntax

```
SubClassOf( :Woman :Person )
```

RDF/XML Syntax

```
<owl:Class rdf:about="Woman">
  <rdfs:subClassOf rdf:resource="Person"/>
</owl:Class>
```

Turtle Syntax

```
:Woman rdfs:subClassOf :Person .
```

Manchester Syntax

```
Class: Woman
  SubClassOf: Person
```

OWL/XML Syntax

```
<SubClassOf>
  <Class IRI="Woman"/>
  <Class IRI="Person"/>
</SubClassOf>
```

There are several syntaxes to serialize OWL 2 ontologies.

The RDF/XML syntax is the only syntax that is mandatory to be supported by all OWL 2 tools.

More examples:

http://www.w3.org/TR/owl2-primer/

Open and Closed Worlds

- Open World Assumption (OWA)
  - We cannot assume that all information is known about all the individuals in the domain.
  - Being unable to prove that an individual a is an instance of *X* does not justify our concluding that a is not an instance of *X*.
- Closed World Assumption (CWA)
  - If we cannot deduce that an individual a is an instance of *X* then we can assume that a is an instance of *(complementOf X)*.
- Drawback

  unionOf (A complementOf (A))
  - It may be the case that we cannot determine (given the information at our disposal) exactly which of the two it is an instance of. Although an OWL reasoner should always answer yes to the question "is a an instance of (unionOf (A (complementOf A)))", it may answer no to both "is a an instance of A" *and* "is a an instance of (complementOf A)".

OWL as DL: Axioms

Axiom	DL Syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human $\sqsubseteq$ Animal $\sqcap$ Biped
equivalentClass	$C_1 \equiv C_2$	Man $\equiv$ Human $\sqcap$ Male
disjointWith	$C_1 \sqsubseteq \neg C_2$	Female $\sqsubseteq \neg$ Male
sameAs	$\{x_1\} \equiv \{x_2\}$	{President} $\equiv$ {G.W.Bush}
differentFrom	$\{x_1\} \sqsubseteq \neg \{x_2\}$	{john} $\sqsubseteq \neg$ {peter}
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter $\sqsubseteq$ hasChild
equivalentProperty	$P_1 \equiv P_2$	cost $\equiv$ prise
inverseOf	$P_1 \equiv P_2^-$	hasChild $\equiv$ hasParent$^-$
transitiveProperty	$P^+ \sqsubseteq P$	ancestor$^+$ $\sqsubseteq$ ancestor
functionalProperty	$T \sqsubseteq \leq 1 P$	T $\sqsubseteq \leq$ 1 hasMother
inverseFunctionalProperty	$T \sqsubseteq \leq 1 P^-$	T $\sqsubseteq \leq$ 1 SSN$^-$
reflexive		knows
irreflexible		isMotherOf
symetric		isSibling
antisymetic		isChildOf

Unique Name Assumption (UNA)

- The Unique Name Assumption (UNA) says that any two individuals with different names are different individuals.
- OWL semantics does **not** make the UNA
  - There are mechanisms in the language (`owl:differentFrom` and `owl:AllDifferent`) that allow us to assert that individuals are different.

Reasoning

- Modern reasoners use the *Tableau Algorithm*
  - Try to build a tree like model by decomposition uses tableau rules corresponding to constructors in logic (e.g., $\sqcap$, $\exists$)
  - Some steps are nonterministic (e.g. $\sqcup$)
  - In practice, this means search
- Examples use the Pellet or Fact++ reasoner and the Protègè interface

OWL Pizza Example

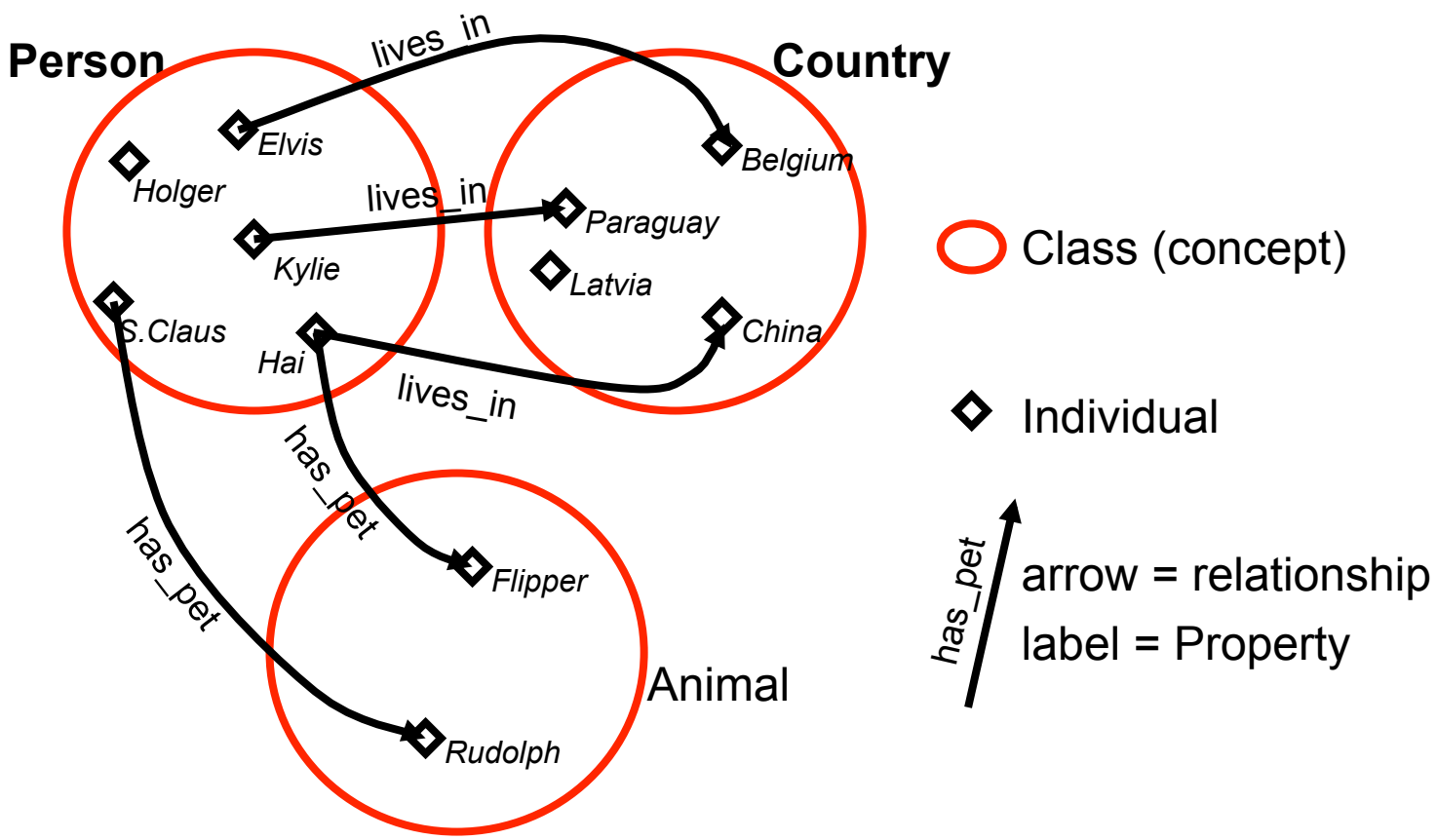
- Explaning OWL and DL with the help of the "Pizza" ontology.
- Full description of the example can be found in "The Practical Guide To Building OWL Ontologies Using The Protègè-OWL Plugin and CO-ODE Tools"

  http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf
- Slices are partly (or mostly) taken from "A Practical Introduction to Ontologies & OWL" from Nick Drummond and Matthew Horridge University of Manchester

Ontology Engineering

- Ontology Engineering stands for the process of modeling an ontology for the domain of discourse.
- Most often it is **not** the domain expert that formalises their knowledge – because of the complexity of the modelling 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.
- Our domain of discourse is Pizza making.
  - Luckily, we are all experts in Pizzas, so we just need some material to verify our knowledge…

OWL Constructs



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 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)
- 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
- N-ary relationships can be modelled with workarounds in OWL

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

protégé ...

- Is a knowledge modelling environment
- Is free, open source software
- Is developed by Stanford Medical Informatics
- 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 plug-ins to allow backend / interface extensions

Protégé-OWL

swrc_v0.3.owl (file:/Users/reif/Desktop/swrc_v0.3.owl) - [/Users/reif/Documents/Teaching/Semantic Web Examples/OWL Examples/Pizza.owl]

swrc_v0.3.owl (file:/Users/reif/Desktop/swrc_v0.3.owl)

Active Ontology | Entities | Classes | Object Properties | Data Properties | Individuals | OWLViz | DL Query

Asserted class hierarchy | Inferred class hierarchy

Asserted class hierarchy: CheeseyPizza

- Thing
  - DomainConcept
    - Country
    - IceCream
    - Pizza
      - CheeseyPizza
      - InterestingPizza
      - MeatyPizza
      - NamedPizza
      - NonVegetarianPizza
      - RealItalianPizza
      - SpicyPizza
      - SpicyPizzaEquivalent
      - VegetarianPizza
      - VegetarianPizzaEquivalent1
      - VegetarianPizzaEquivalent2
    - PizzaBase
      - DeepPanBase
      - ThinAndCrispyBase
    - PizzaTopping
      - CheeseTopping
      - FishTopping
      - FruitTopping
      - HerbSpiceTopping
      - MeatTopping
      - NutTopping
      - SauceTopping
      - SpicyTopping
      - VegetableTopping
      - VegetarianTopping
  - ValuePartition

Class Annotations | Class Usage

Annotations: CheeseyPizza

Annotations

comment
"Any pizza that has at least 1 cheese topping."@en

label
"PizzaComQueijo"@pt

Description: CheeseyPizza

Equivalent classes

Pizza
and hasTopping some CheeseTopping

Superclasses

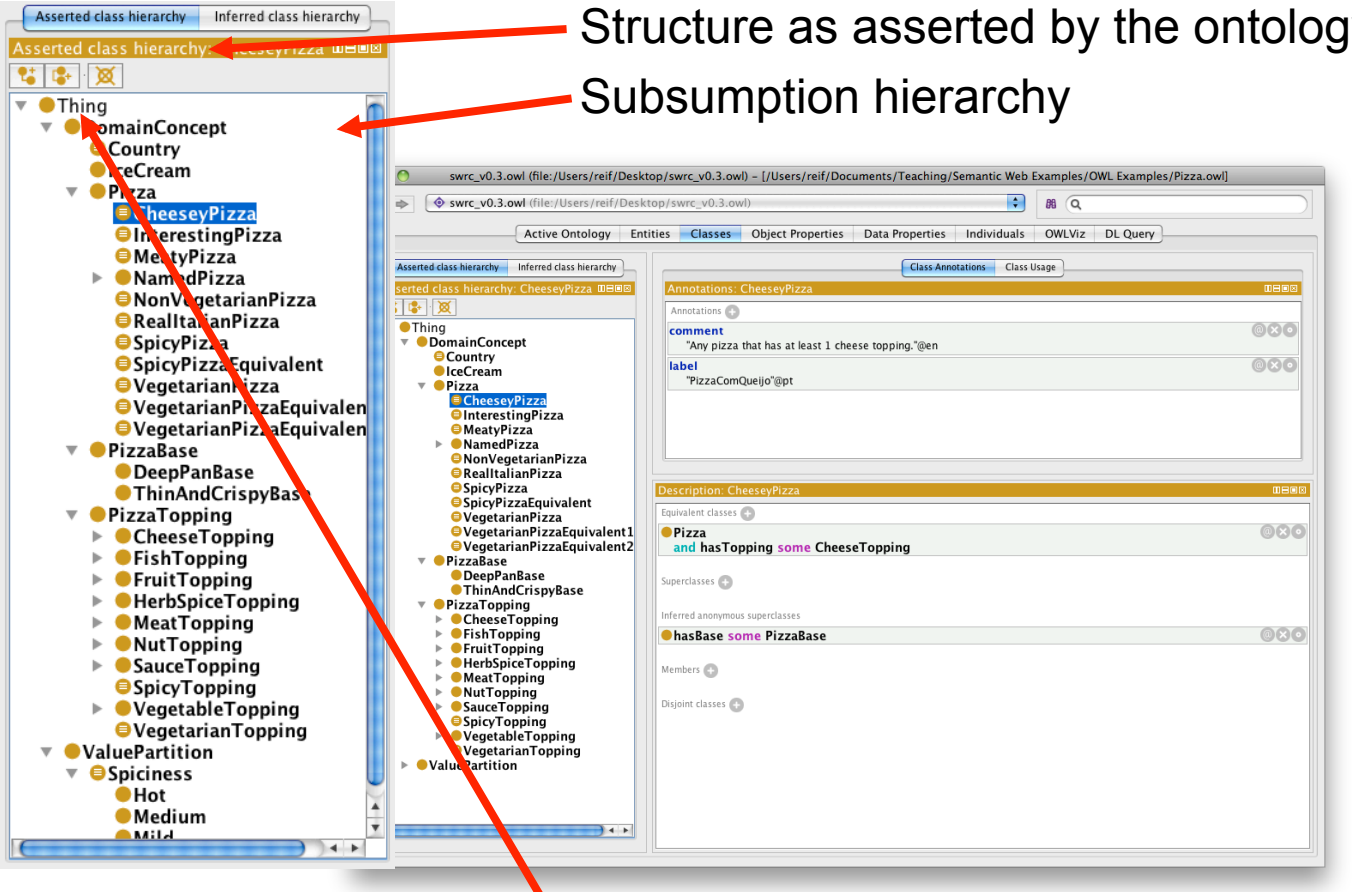
Inferred anonymous superclasses

hasBase some PizzaBase

Members

Disjoint classes

Class Hierarchy



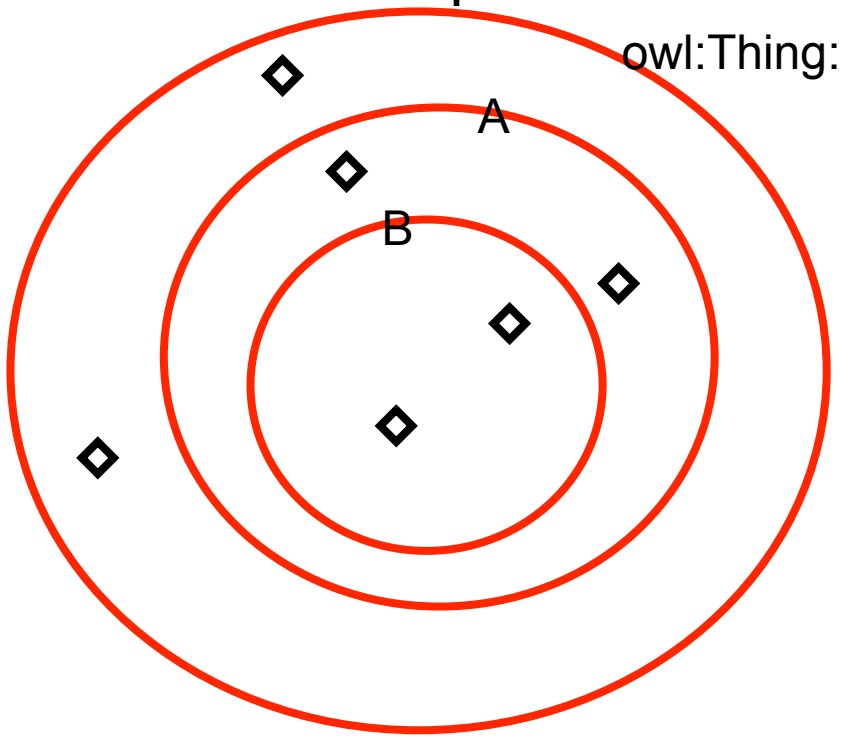
Structure as asserted by the ontology engineer

Subsumption hierarchy

owl:Thing is the root class

Subsumption

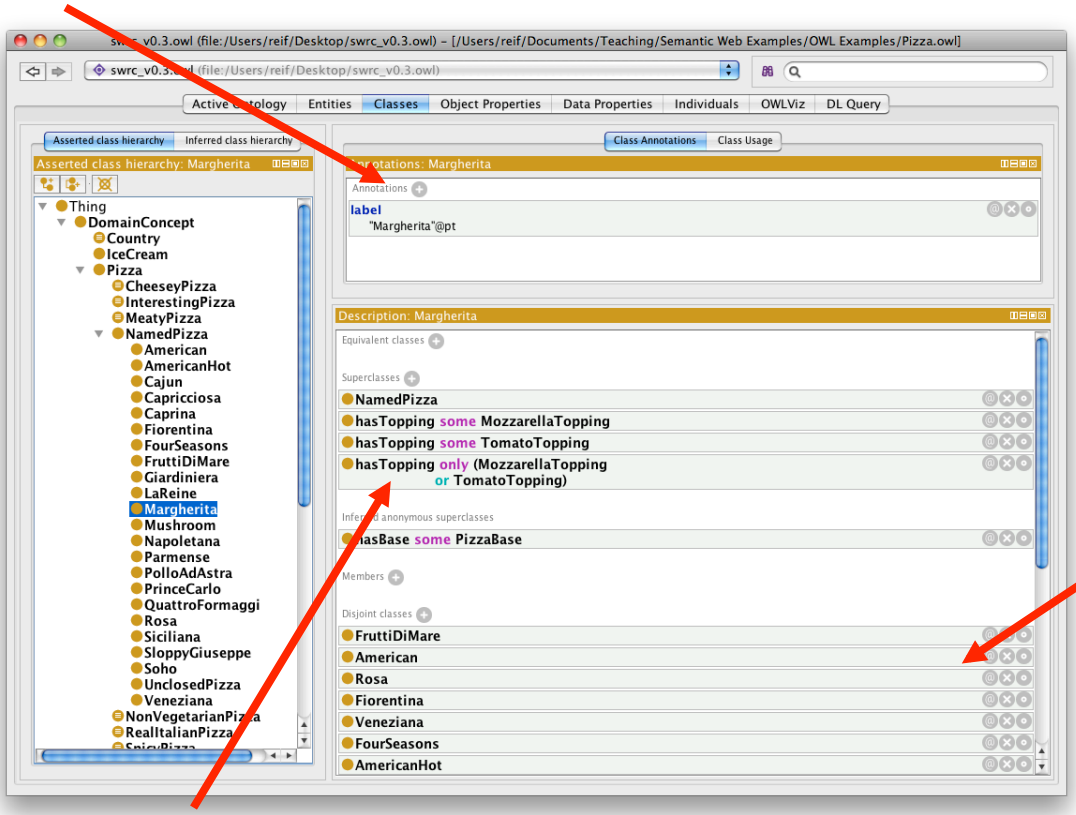
- Superclass/subclass relationship, “isa”
- **All** members of a subclass can be inferred to be members of its superclasses



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

Class Editor

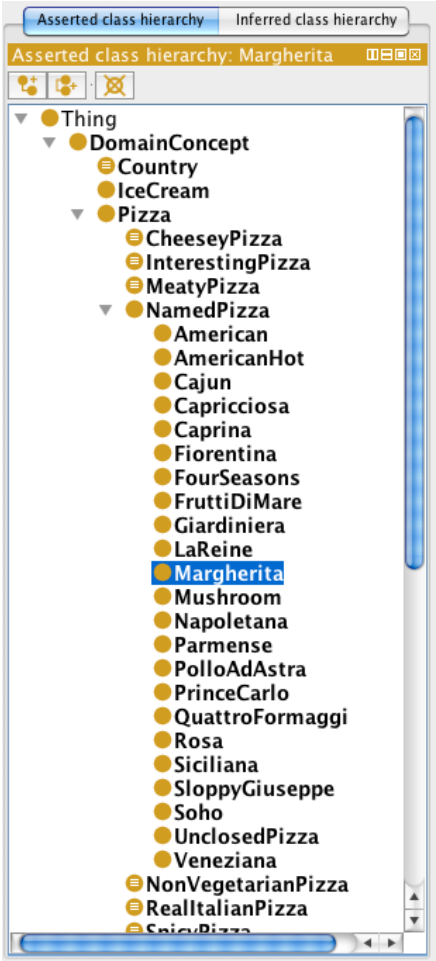
Class documentation, and annotation (for class meta-data)



Disjoints

Conditions Widget

Creating Disjunct Classes



- Create classes for Pizzas, Bases and Toppings
- Put Toppings into several subclasses
- Make classes disjoint with siblings
- Wizard to create class hierarchies:

  Toos ➔ Create Class hierarchies…

Meaning Disjunct Classes

- OWL Classes are assumed to *overlap*.
- We therefore cannot assume that an individual is not a member of a particular class simply because it has not been asserted to be a member of that class.
- In order to *separate* a group of classes we must make them disjoint from one another.
  - In our above example Pizza, PizzaTopping and PizzaBase have been made disjoint from one another.
  - This means that it is not possible for an individual to be a member of a combination of these classes - it would not make sense for an individual to be a Pizza and a PizzaBase!

Consistency Checking

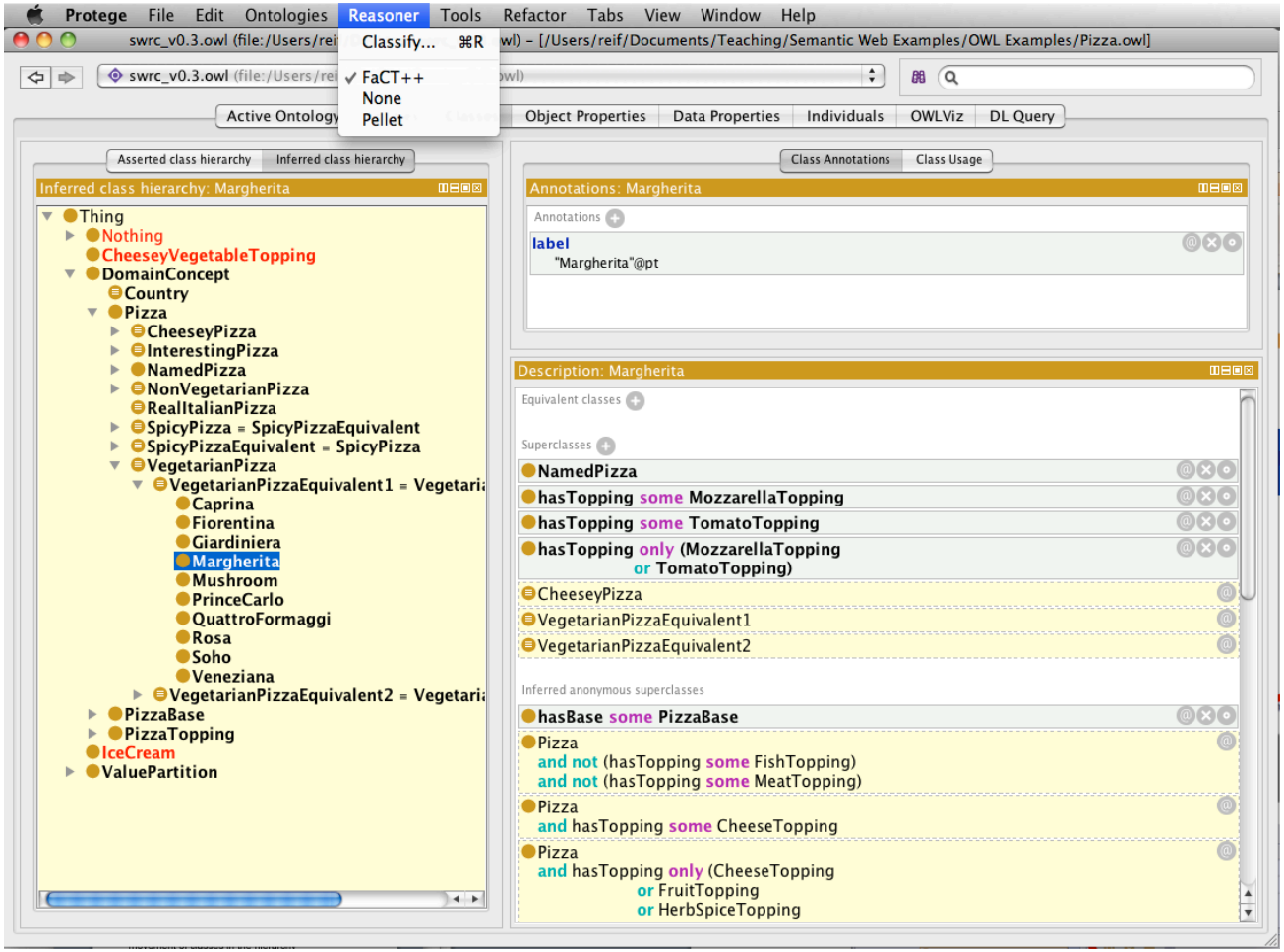
- We've just created a class that doesn't really make sense
  - MeatyVegetableTopping subclass of MeatTopping and VegetableTopping
  - What is a MeatyVegetableTopping?
- We'd like to be able to check the logical of our model
- This is one of the tasks that can be done automatically by software known 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 and to check its consistency.
- You may also hear them being referred to as Classifiers
- Standard reasoner services are:
  - Consistency Checking
  - Subsumption Checking
  - Equivalence Checking
  - Instantiation Checking

- 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". A well designed ontology can be compiled to check its meaning is that intended.

Accessing the Reasoner

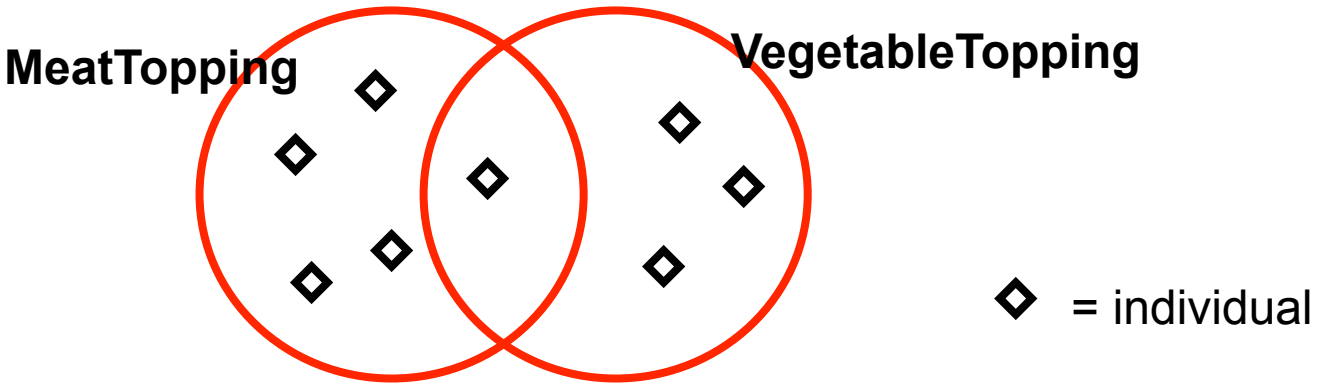


Reasoning about our Pizzas

- 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

Disjointness

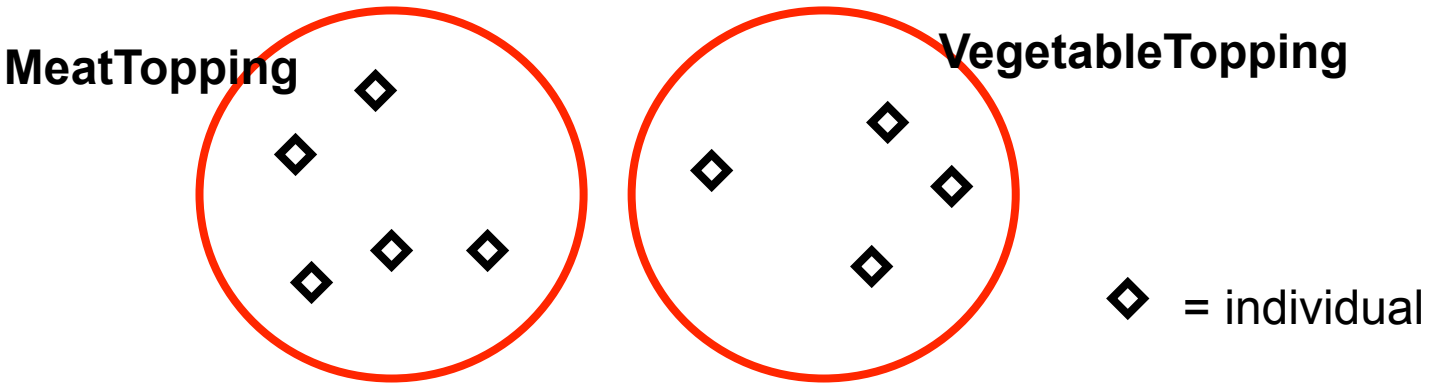
- OWL assumes that classes overlap



  - 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

Why is MeatyVegetableTopping Inconsistent?

- We have asserted that a **MeatyVegetableTopping** is a subclass of two classes we have stated are disjoint

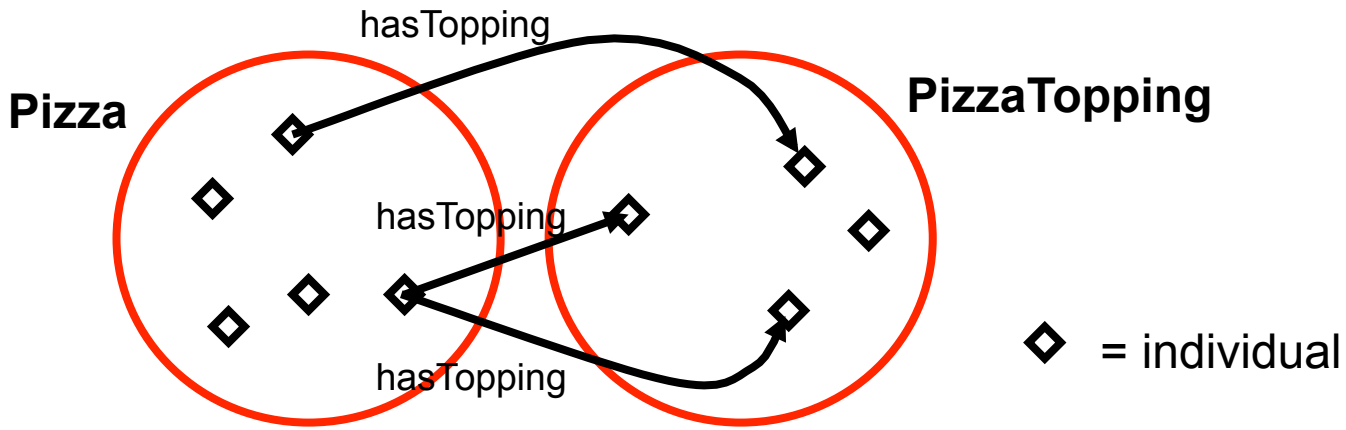
- The disjoint means nothing can be a **MeatTopping** and a **VegetableTopping** at the same time
- This means that **MeatyVegetableTopping** can never contain any individuals

- The class is therefore inconsistent
- This is what we expect!

- It can be useful to create classes we expect to be inconsistent to "test" your model – often we refer to these classes as "probes" – generally it is a good idea to document them as such to avoid later confusion

What are we missing?

- This is not a semantically rich model
- Apart from “is kind of” (subsumption) and “is not kind of” (disjoint, 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



Relationships in OWL

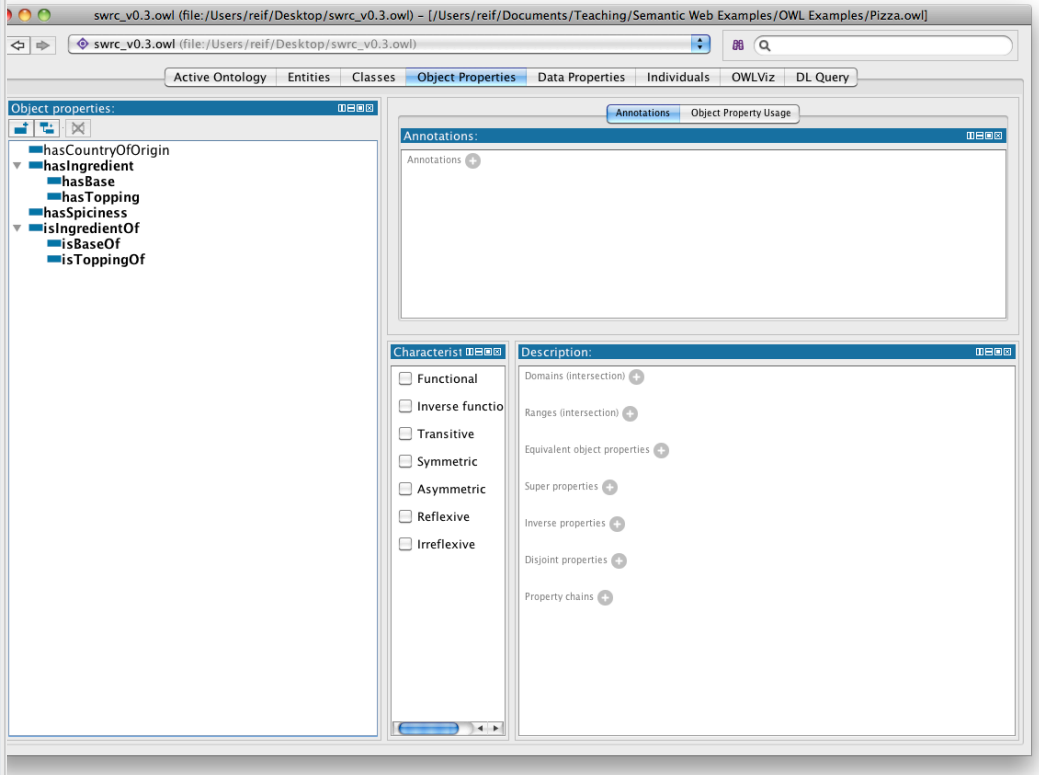
- In OWL-DL, relationships can only be formed between Individuals or between an Individual and a data value.
  (In OWL-Full, Classes can be related, but this cannot be reasoned with)
- Relationships are formed along Properties
- We can restrict how these Properties are used:
  - Globally – by stating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

OWL Properties

- Object Property – relates Individuals
- Datatype Property – relates Individuals to data (int, string, float etc)
- Annotation Property – for attaching metadata to classes, individuals or properties

Properties Tab: Property Browser

Note that Properties can be in a hierarchy



Creating Properties

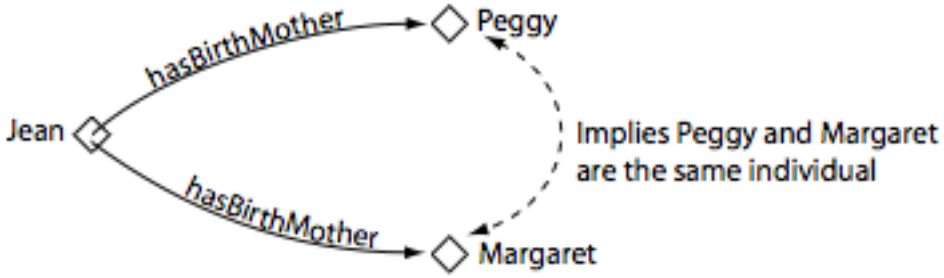
- We tend to name properties using camelNotation with a lowercase letter to begin

- We often create properties using 2 standard naming patterns:
  - has… (eg hasColour)
  - is…Of (eg isTeacherOf) or other suffixes
- This has several advantages:
  - It is easier to find properties
  - It is easier for tools to generate a more readable form (see tooltips on the classes in the hierarchy later)
  - Inverses properties typically follow this pattern eg hasPart, isPartOf

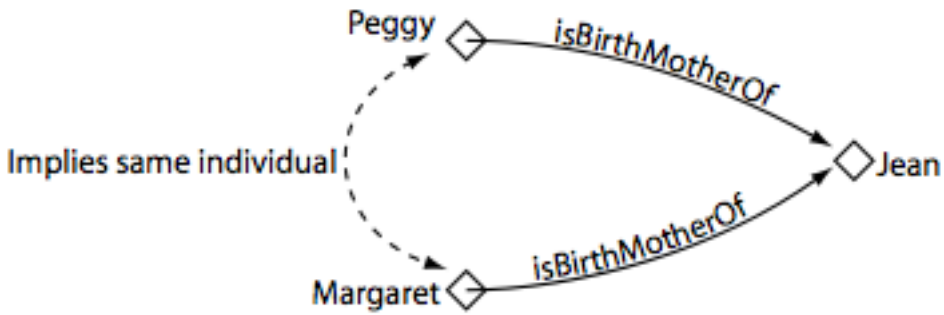
- Our example hasTopping fits into this

Property Characteristics 1/4

- Functional: at most individual as object

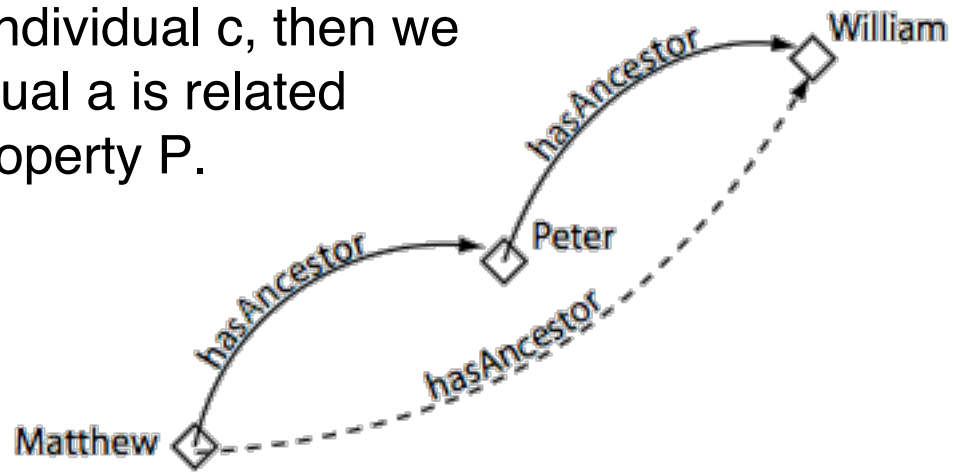


- Inverse functional: at most one individual as subject

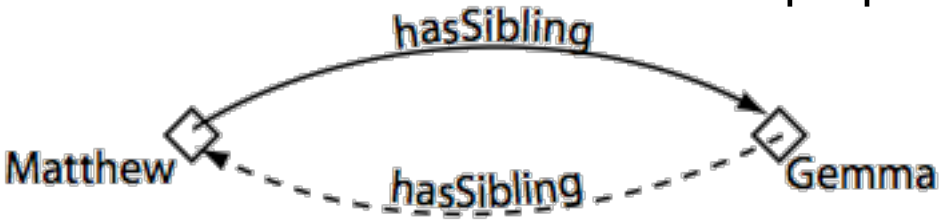


Property Characteristics 2/4

- Transitive: If the property P relates individual a to individual b, and also individual b to individual c, then we can infer that individual a is related to individual c via property P.

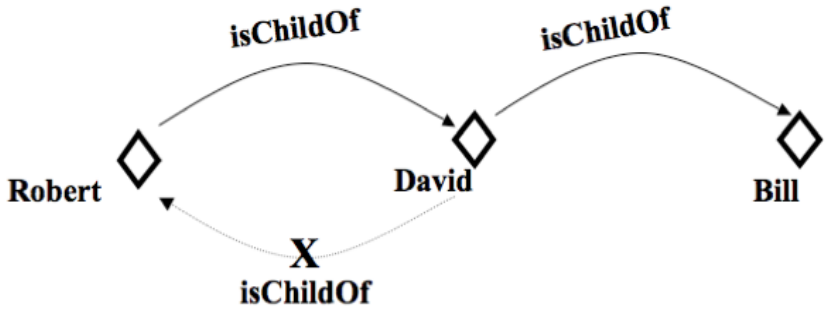


- Symmetric: The property P relates individual a to individual b then individual b is also related to individual a via property P.

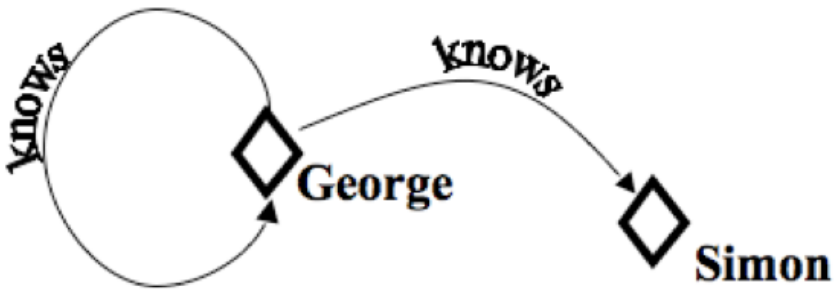


Property Characteristics 3/4

- antisymetric: If a property P is antisymmetric, and the property relates individual a to individual b then individual b cannot be related to individual a via property P.

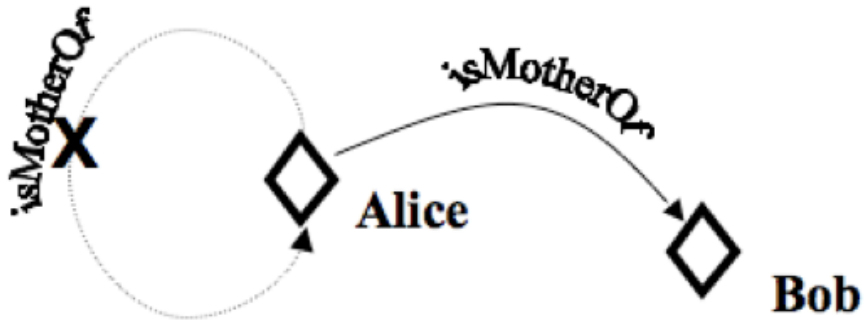


- reflexive: A property P is said to be reflexive when the property must relate individual a to itself.



Property Characteristics 4/4

- irreflexive: If a property P is irreflexive, it can be described as a property that relates an individual a to individualb, where individual a and individualb are not the same.



Domain and Range - Observation 1/2

- OWL domains and ranges should not be viewed as constraints to be checked.
- They are used as “axioms” in reasoning.
  - For example if the property hasTopping has the domain set as Pizza and we then applied the hasTopping property to IceCream (individuals that are members of the class IceCream), this would generally not result in an error. It would be used to infer that the class IceCream must be a subclass of Pizza!
  - An error will only be generated (by a reasoner) if Pizza is disjoint to IceCream

Domain and Range - Observation 2/2

- It is possible to specify multiple classes as the range for a property.
- If multiple classes are specified in Protege the range of the property is interpreted to be the intersection of the classes.
  - For example, if the range of a property has the classes Man and Woman listed in the range widget, the range of the property will be interpreted as intersection of Man and Woman.
- The same observation holds for the specified domain.

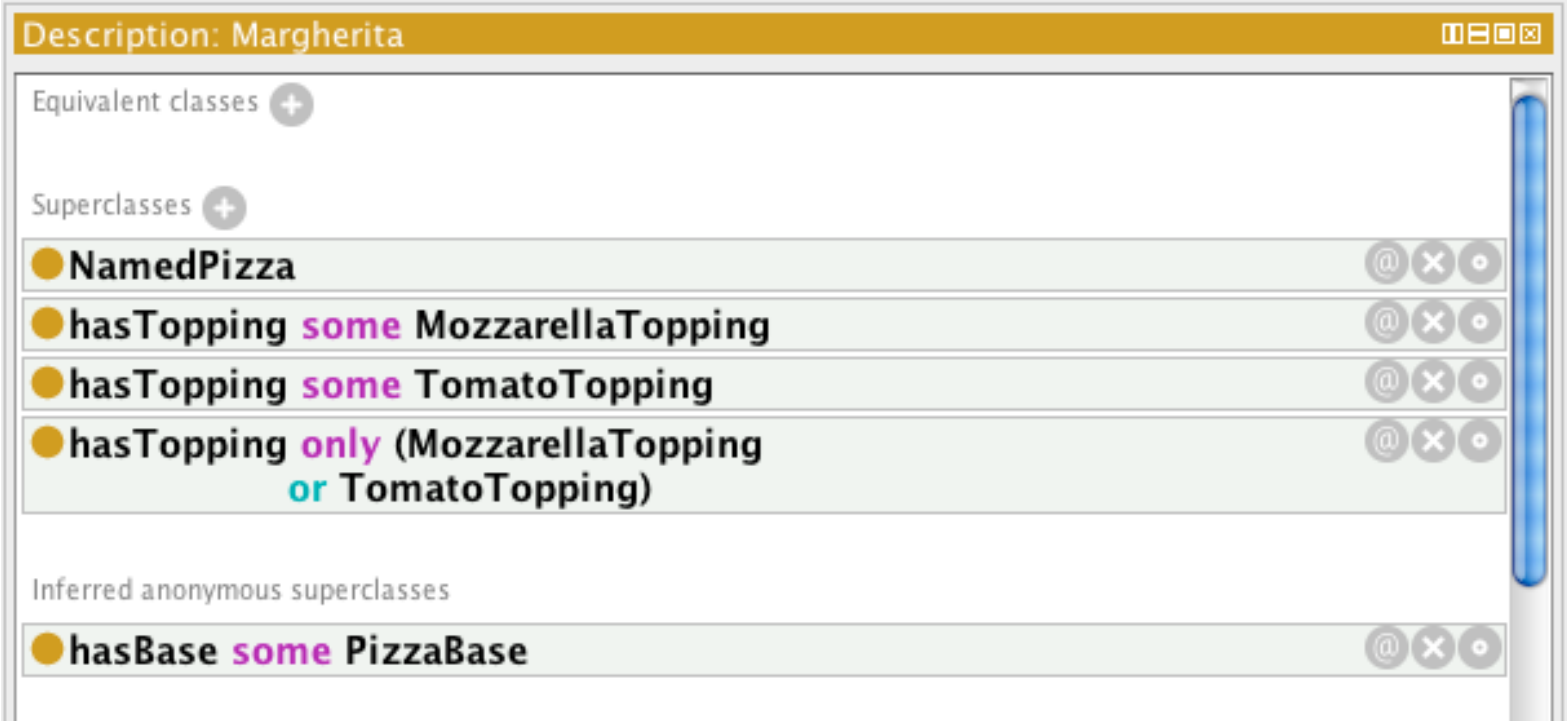
Associating Properties with Classes

- We now have 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
- We create Restrictions using the Conditions widget

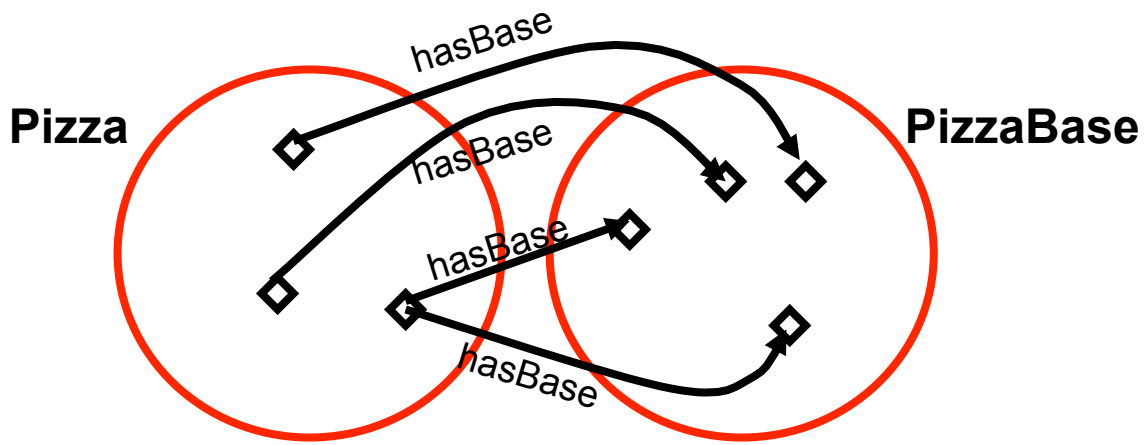
- Conditions can be any kind of Class – you have already added Named superclasses in the Conditions Widget. Restrictions are a type of Anonymous Class

Conditions Widget



What does this mean?

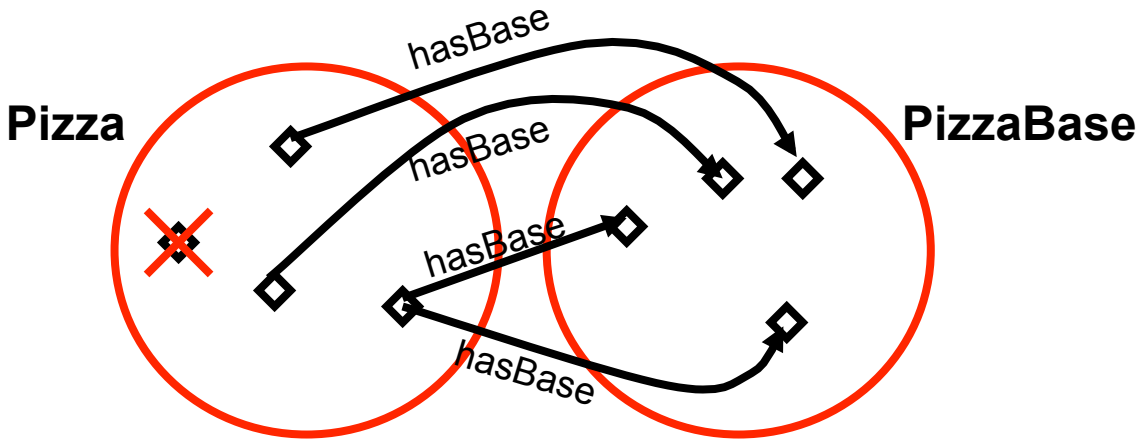
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



  - ► "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?

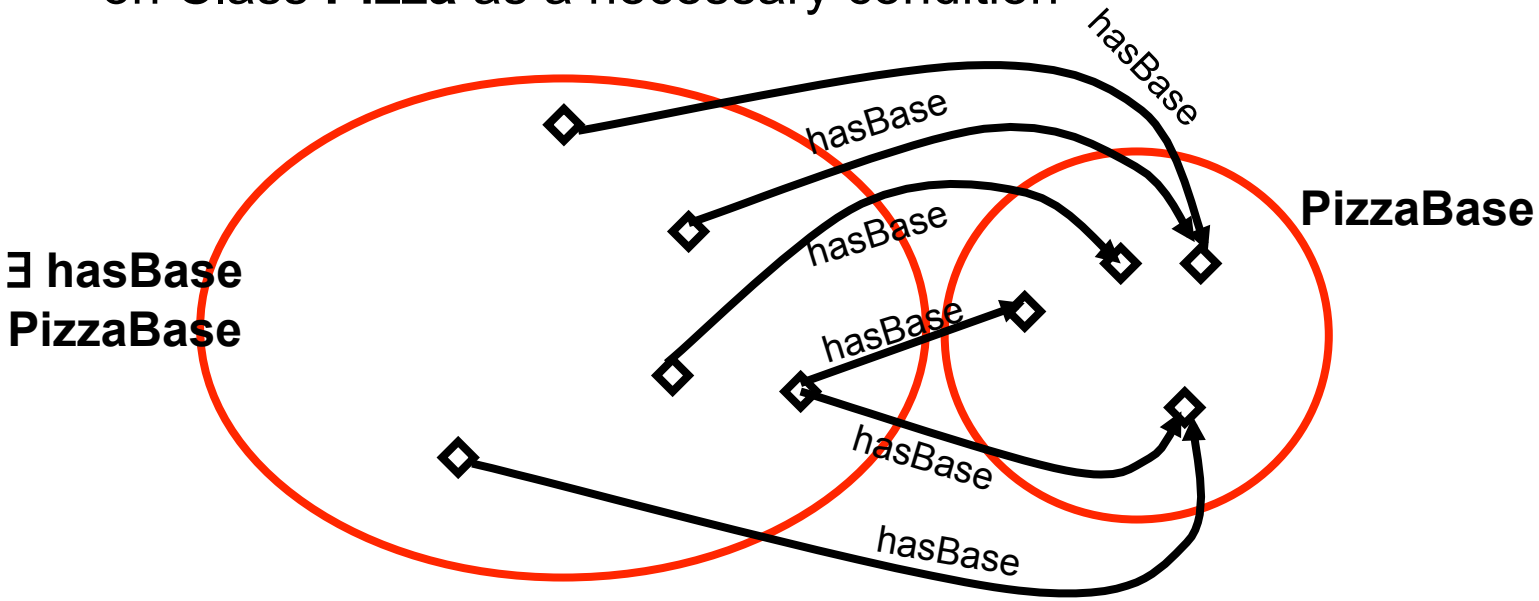
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



► "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**"

Why?

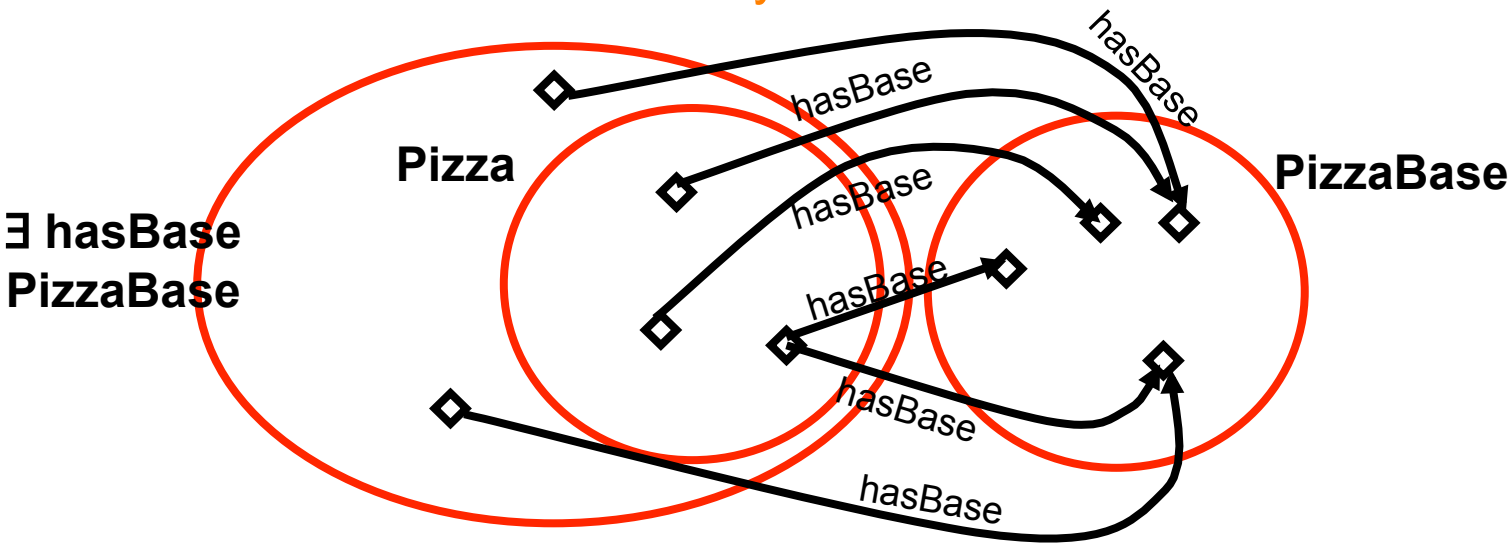
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



- 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**

Creating Some Tasty Pizzas

- MargaritaPizza: Tomato, Mozzarella
- AmericanPizza: Tomato, Mozzarella, Pepperoni
- AmericanHotPizza: Tomato, Mozzarella, Pepperoni, JalapenoPepper
- SohoPizza: Tomato, Mozzarella, Olive, Parmezan

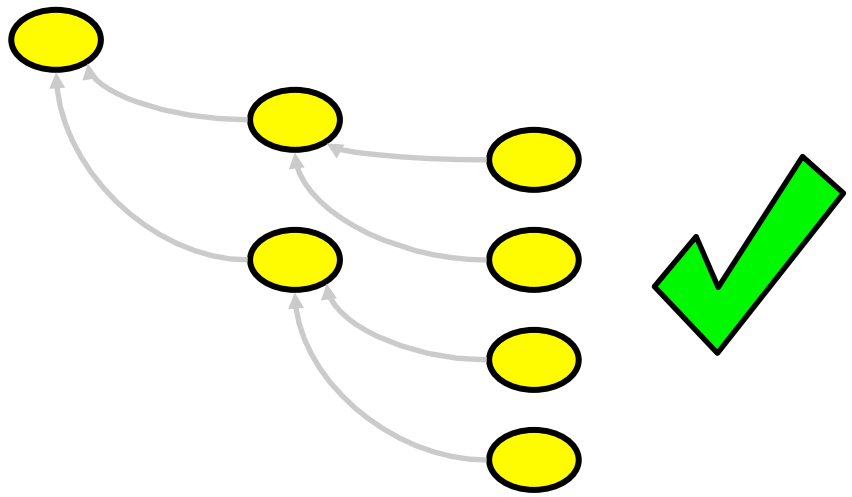
Restriction Types

∃	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”

Primitive Classes

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

disjoint tree



Polyhierarchies

- By the end of this tutorial we intent 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)

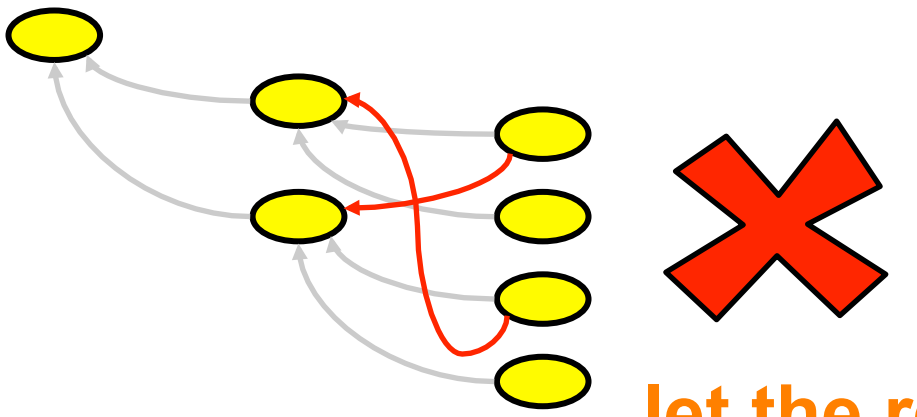
BUT…

Asserted Polyhierarchies

We believe asserting polyhierarchies is bad

- 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 *Superclass* block to the *Equivalent* block which changes the meaning

- And classify again…

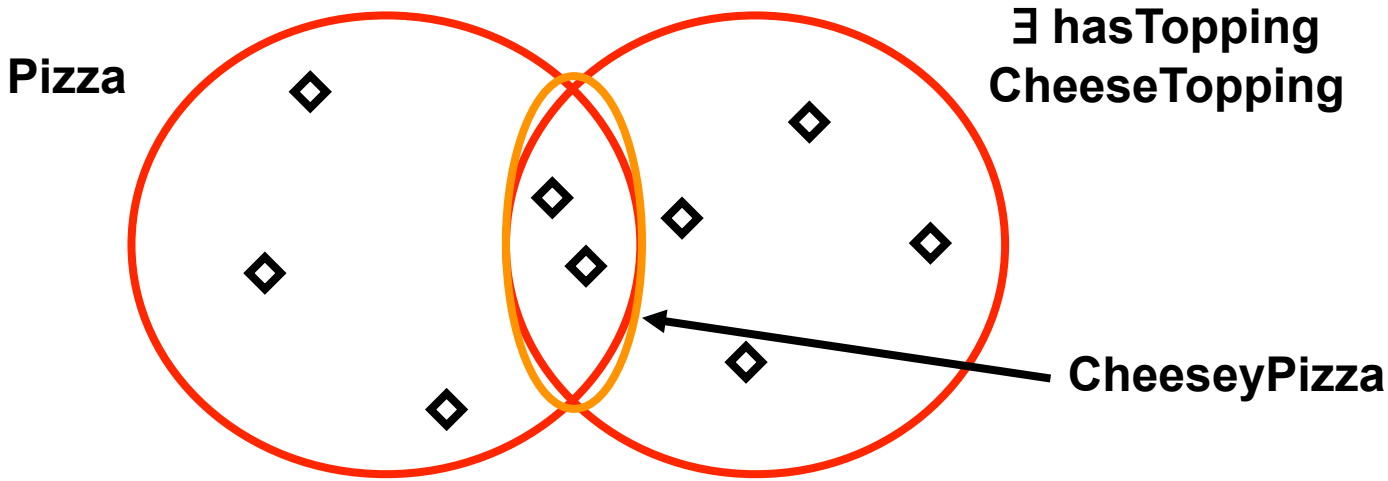
Reasoner Classification

- 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

Why?

Necessary & Sufficient Conditions

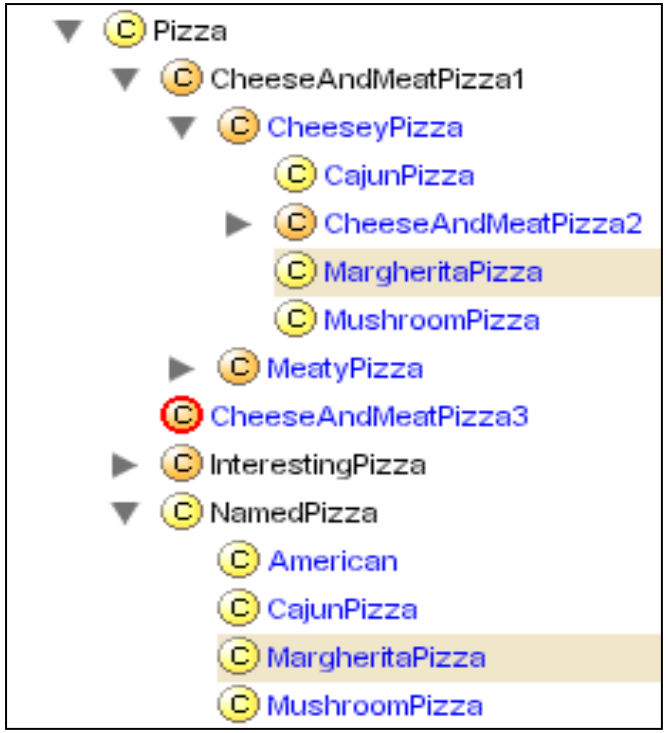
- Each set of necessary & sufficient conditions is an Equivalent Class



- **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

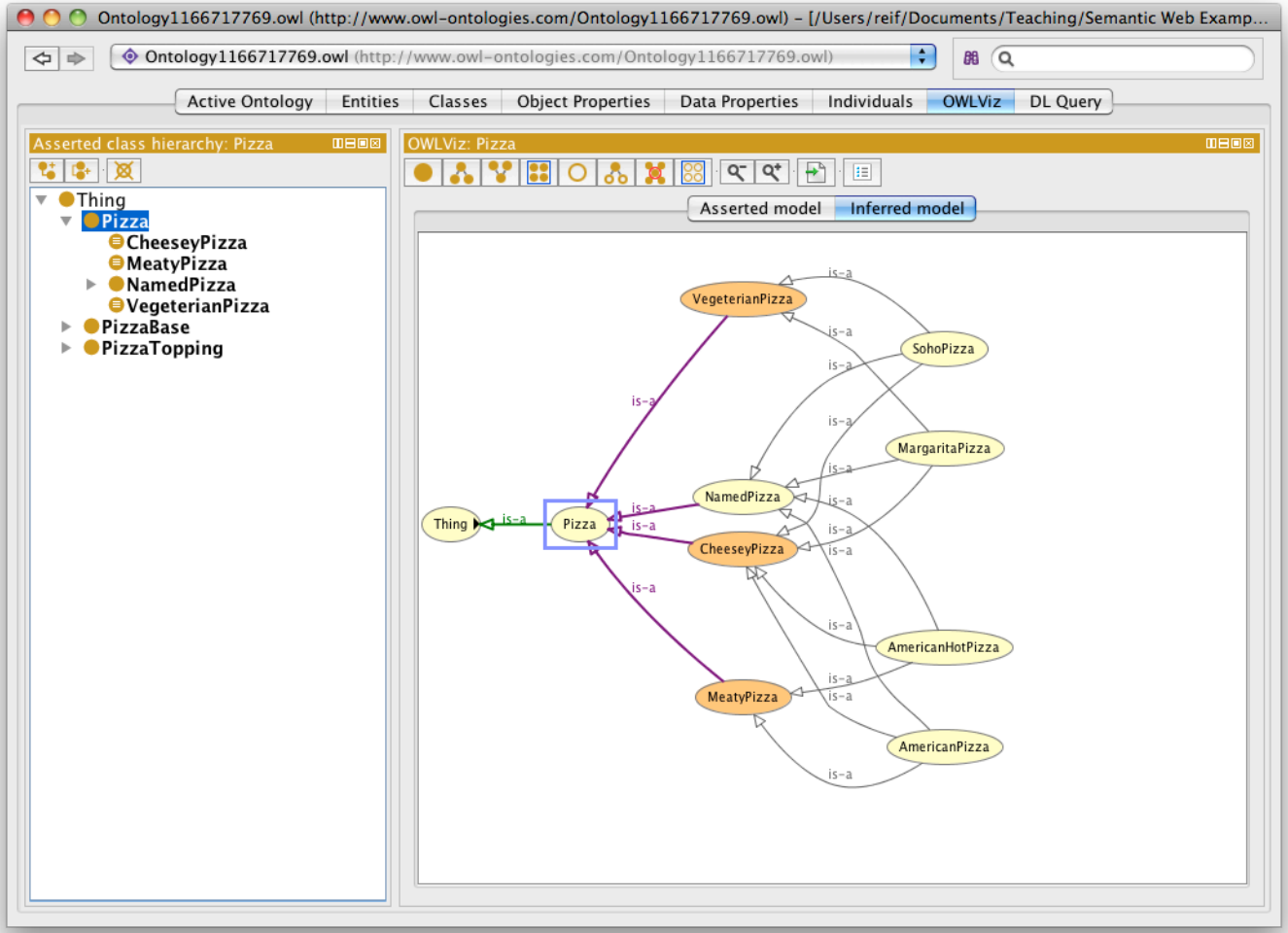
- We don't currently have many kinds of primitive pizza but its easy to see that if we had, it would have been a substantial task to assert **CheeseyPizza** as a parent of lots, if not all, of them

- 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"

OWLViz Tab


Ontology1166717769.owl (http://www.owl-ontologies.com/Ontology1166717769.owl) - [/Users/reif/Documents/Teaching/Semantic Web Examp...
Ontology1166717769.owl (http://www.owl-ontologies.com/Ontology1166717769.owl)
Active Ontology
Entities
Classes
Object Properties
Data Properties
Individuals
OWLViz
DL Query
Asserted class hierarchy: Pizza
Thing
Pizza
CheeseyPizza
MeatyPizza
NamedPizza
VegeterianPizza
PizzaBase
PizzaTopping
OWLViz: Pizza
Asserted model
Inferred model
VegeterianPizza
SohoPizza
MargaritaPizza
NamedPizza
Thing
Pizza
CheeseyPizza
AmericanHotPizza
MeatyPizza
AmericanPizza
is-a

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**
  - Opposite to the Primitive Classes defined before.
  - 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:

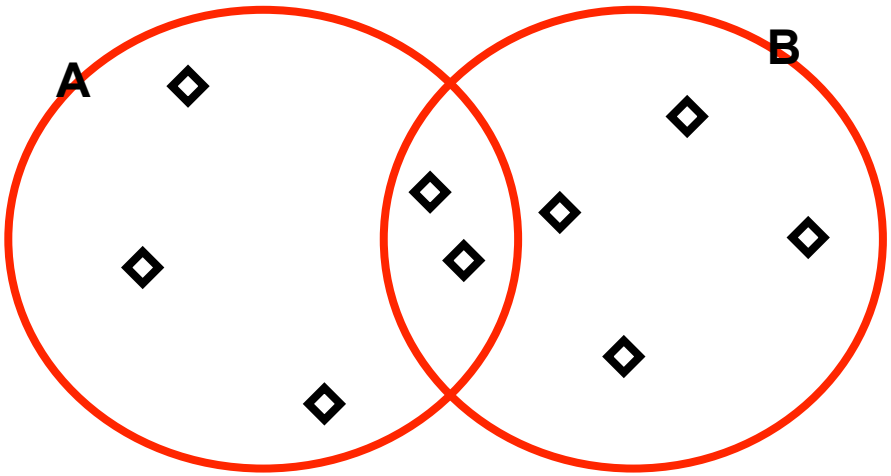
1. To be able to create a vegetarian topping
   This requires a Union Class

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)



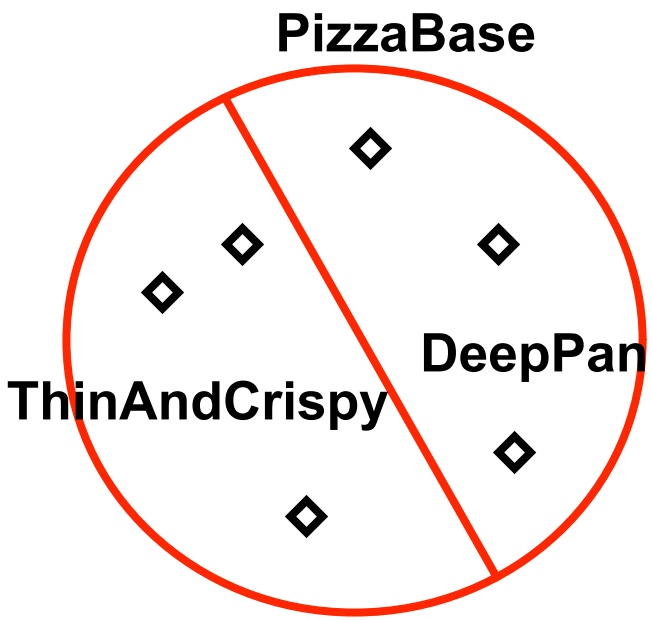
Covering Axioms

- 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.

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

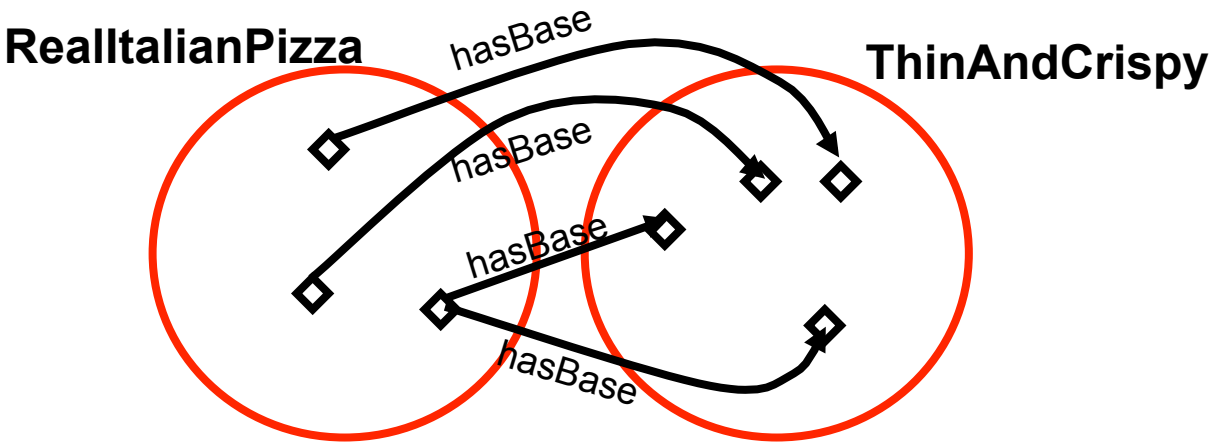
- We need to say our **VegetarianPizza** can only have toppings that are vegetarian toppings
- We can do this by creating a Universal or AllValuesFrom restriction
- We'll first look at an example…

Real Italian Pizzas

- “RealItalianPizzas only have bases that are ThinAndCrispy”
- A Universal Restriction (“only”) 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?

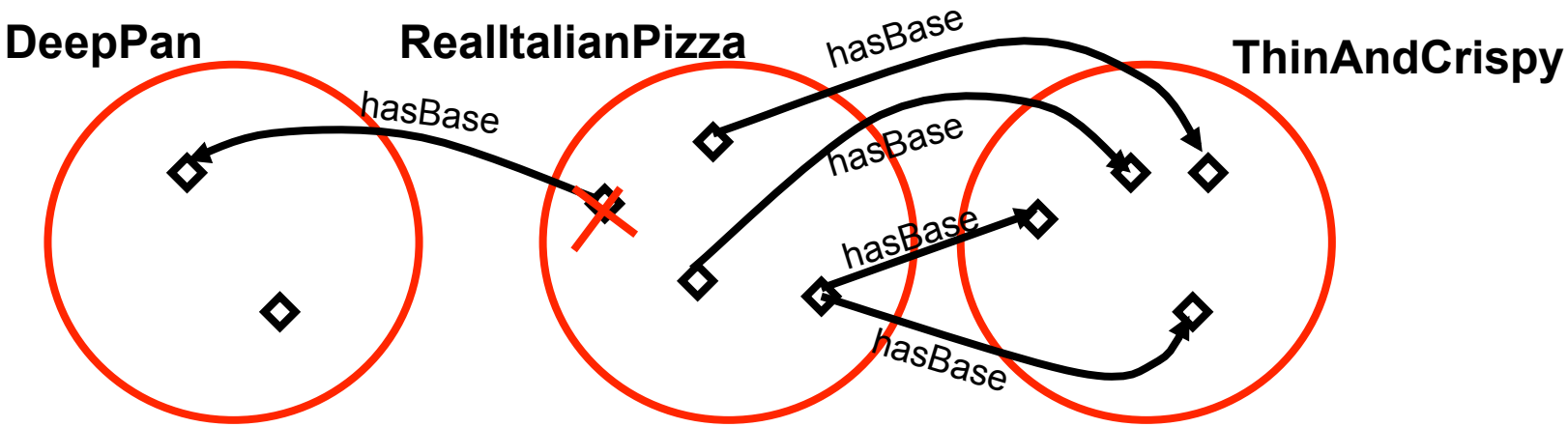
- We have created a restriction: ∀ hasBase **ThinAndCrispy** on Class **RealItalianPizza** as a necessary condition



► "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?

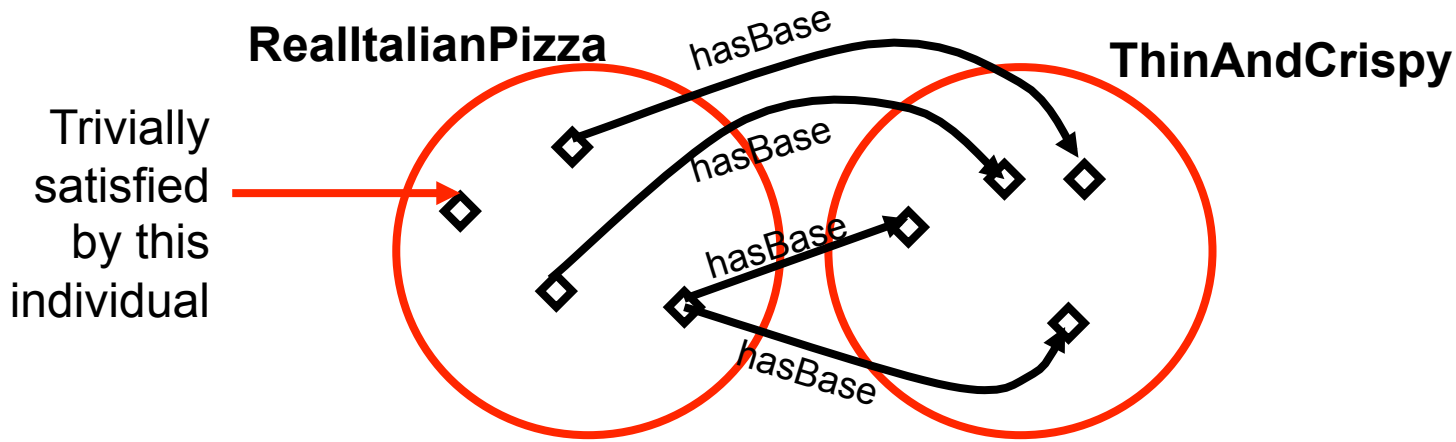
- We have created a restriction: ∀ hasBase **ThinAndCrispy** on Class **RealItalianPizza** as a necessary condition



- ► "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

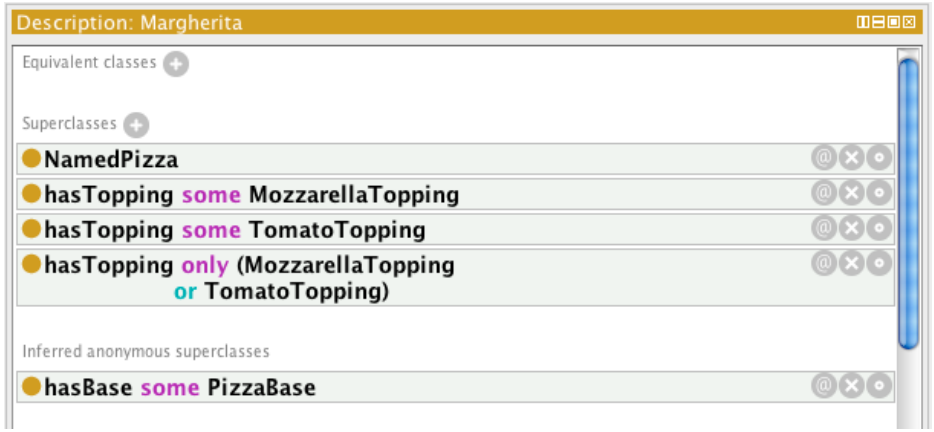
- This is in the form of a Universal Restriction with a filler that is the Union of the other fillers for that property

- 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**
only toppings from **MozzarellaTopping** or **TomatoTopping**



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

Pizza Example Summary

You should now be able to:

- Create Defined Classes and classify using a Reasoner to check expected results
- Create Covering Axioms
- Close Class Descriptions and understand the Open World Assumption

Ontology Engineering

- Developing an ontology for a domain is a complex task.
- Ontology engineering provides methodologies to systematically define ontologies.
- Guidelines when to use a class or a property and how to structure the classes in a subclass hierarchy.
- Introduction in ontology engineering:

  http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness.pdf

Why to develop an Ontology?

- To share common understanding of the structure of information among people or software agents
- To enable reuse of domain knowledge
- To make domain assumptions explicit
- To separate domain knowledge from the operational knowledge
- To analyze domain knowledge

Developing an Ontology includes

- defining classes in the ontology
- arranging the classes in a taxonomic (subclass–superclass) hierarchy
- defining properties and describing allowed domains and ranges for these properties
- filling in the values for properties for instances

Rules when defining an Ontology

- There is no one correct way to model a domain
  - There are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.
- Ontology development is necessarily an iterative process.
- Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.

Defining the Skope of an Ontology

- Questions you should answer:
  - What is the domain that the ontology will cover?
  - For what we are going to use the ontology?
  - For what types of questions the information in the ontology should provide answers?
  - Who will use and maintain the ontology?
- Define a set of competency questions
  - A list of questions that a knowledge base based on the ontology should be able to answer.
  - These questions will serve as the litmus test later
    - Does the ontology contain enough information to answer these types of questions?
    - Do the answers require a particular level of detail or representation of a particular area?
    - These competency questions are just a sketch and do not need to be exhaustive.

Steps to the Ontology

- Consider reusing existing ontologies
- Enumerate important terms in the ontology
- Define the classes and the class hierarchy
- Define the properties, their domain and range
- Define the restrictions of the classes
- Create instances
- Check whether the ontology answers the competency questions