UNIVERSITY OF
Southampton

UNIVERSITY OF Southampton

Ontology Engineering and Design Patterns

COMP6215 Semantic Web Technologies

Dr Nicholas Gibbins - nmg@ecs.soton.ac.uk

Ontologies

"a formal, explicit specification of a shared conceptualisation" (Gruber)

https://userpages.uni-koblenz.de/~staab/Research/Publications/2009/handbookEdition2/what-is-an-ontology.pdf

Ontologies

Machine understandable

ontology should represent a shared view of the domain

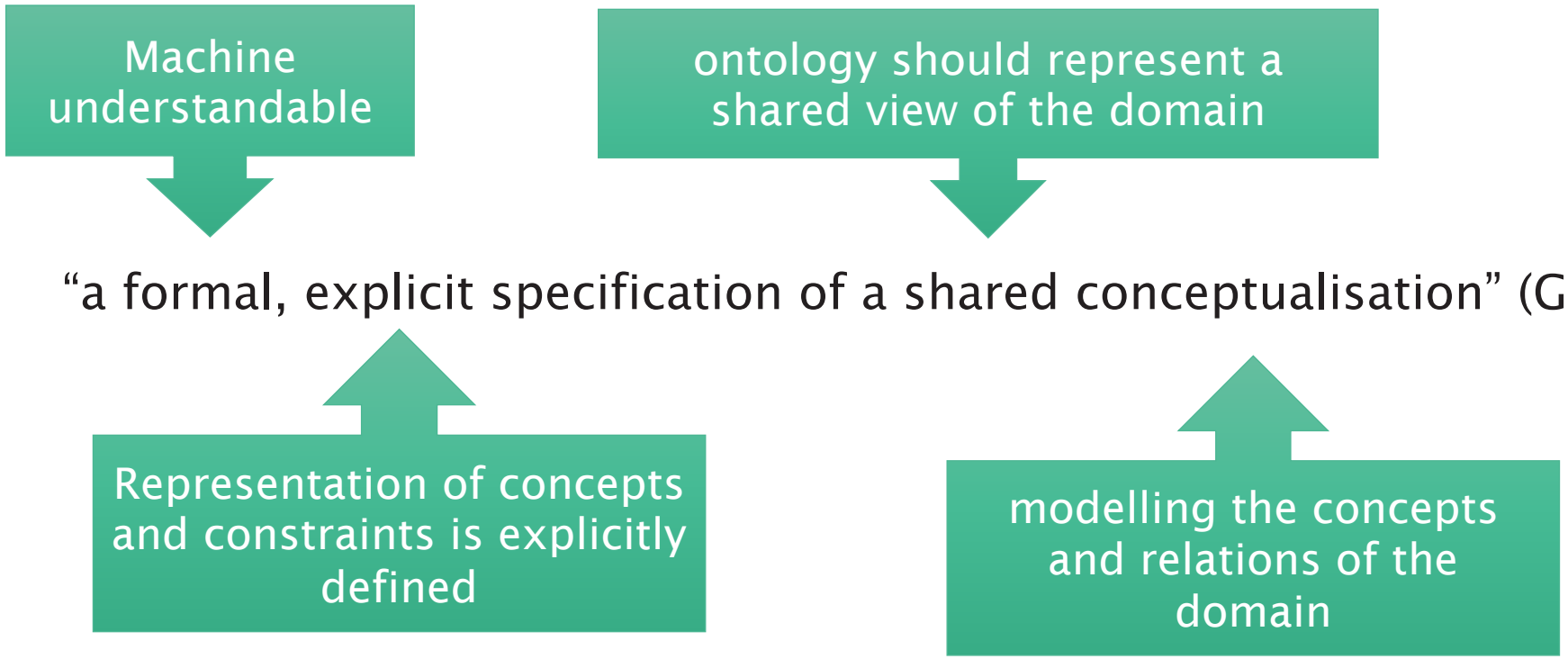
"a formal, explicit specification of a shared conceptualisation" (Gruber)

Representation of concepts and constraints is explicitly defined

modelling the concepts and relations of the domain

https://userpages.uni-koblenz.de/~staab/Research/Publications/2009/handbookEdition2/what-is-an-ontology.pdf

Ontologies



The combination of concepts and relationships required to model a knowledge domain in a human and machine understandable format

https://userpages.uni-koblenz.de/~staab/Research/Publications/2009/handbookEdition2/what-is-an-ontology.pdf

Type of Ontologies

There are four main types of ontologies:

- Representation ontologies
- General or upper-level ontologies
- Domain ontologies
- Application ontologies

Representation ontologies

Describe low level primitive representations

- Such as semantic web languages

Example ontologies:

- OWL, RDF, RDFS

Usual size: small, a few dozens of concepts and relations

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://www.w3.org/2000/01/rdf-schema#> a owl:Ontology ;
  dc:title "The RDF Schema vocabulary (RDFS)" .

rdfs:Resource a rdfs:Class ;
  rdfs:isDefinedBy <http://www.w3.org/2000/01/rdf-schema#> ;
  rdfs:label "Resource" ;
  rdfs:comment "The class resource, everything." .

rdfs:Class a rdfs:Class ;
  rdfs:isDefinedBy <http://www.w3.org/2000/01/rdf-schema#> ;
  rdfs:label "Class" ;
  rdfs:comment "The class of classes." ;
  rdfs:subClassOf rdfs:Resource .

rdfs:subClassOf a rdf:Property ;
  rdfs:isDefinedBy <http://www.w3.org/2000/01/rdf-schema#> ;
  rdfs:label "subClassOf" ;
  rdfs:comment "The subject is a subclass of a class." ;
```

Upper-level ontologies

Describe high-level, abstract, concepts

Usually domain independent

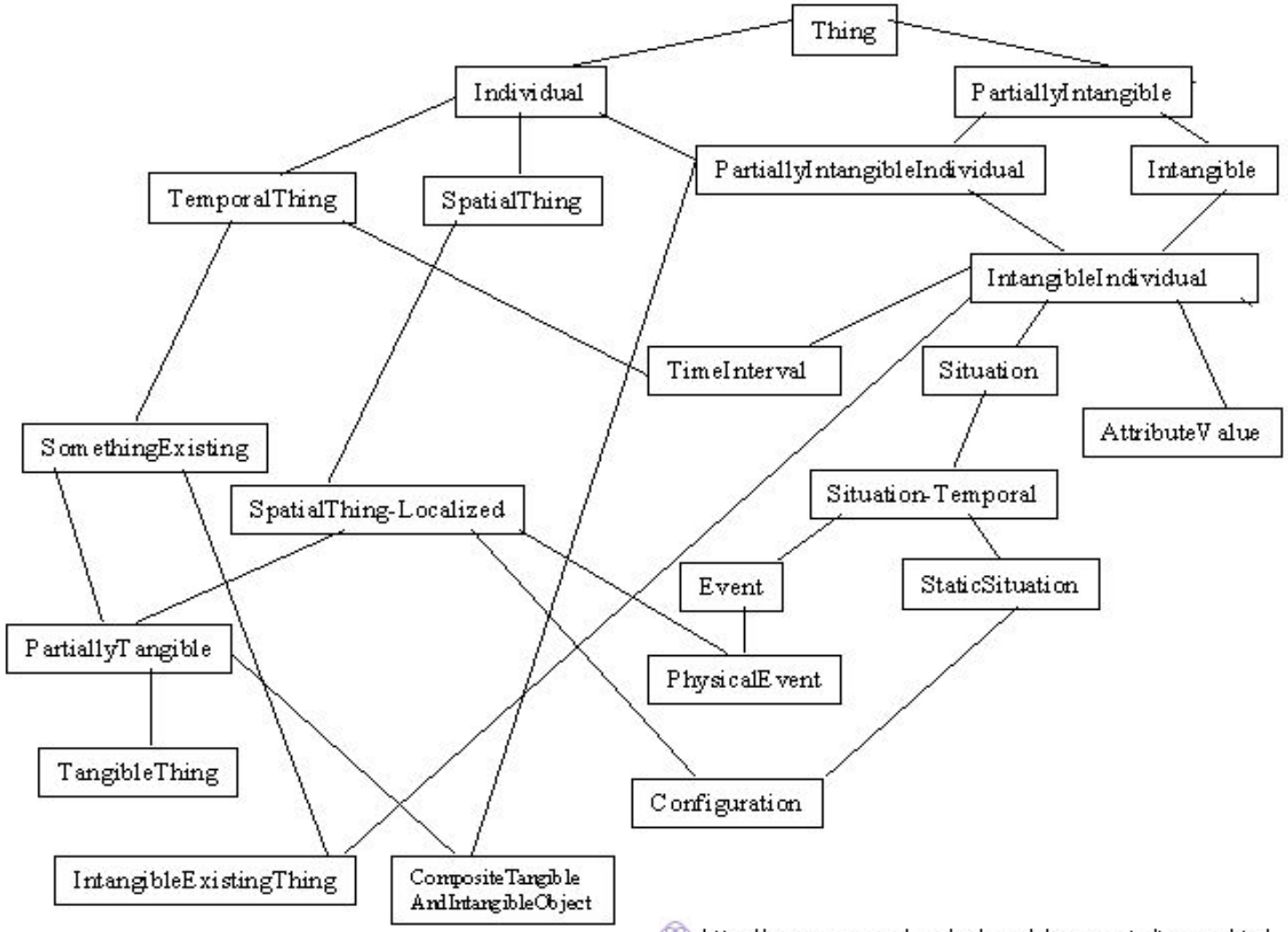
- Can be used as part of other ontologies

Sometimes part of broad ontologies

Examples:

- DOLCE
  (small upper level ontology)
- Cyc: commonsense ontology
  - Hundreds of thousands of concepts
- WordNet: English lexicon
  - Over 150K concepts
- SUMO: Suggested Upper Merged Ontology
  - Around 10K concepts

Example: A (tiny) fragment of Cyc



http://www.cyc.com/cycdoc/vocab/upperont-diagram.html

Domain ontologies

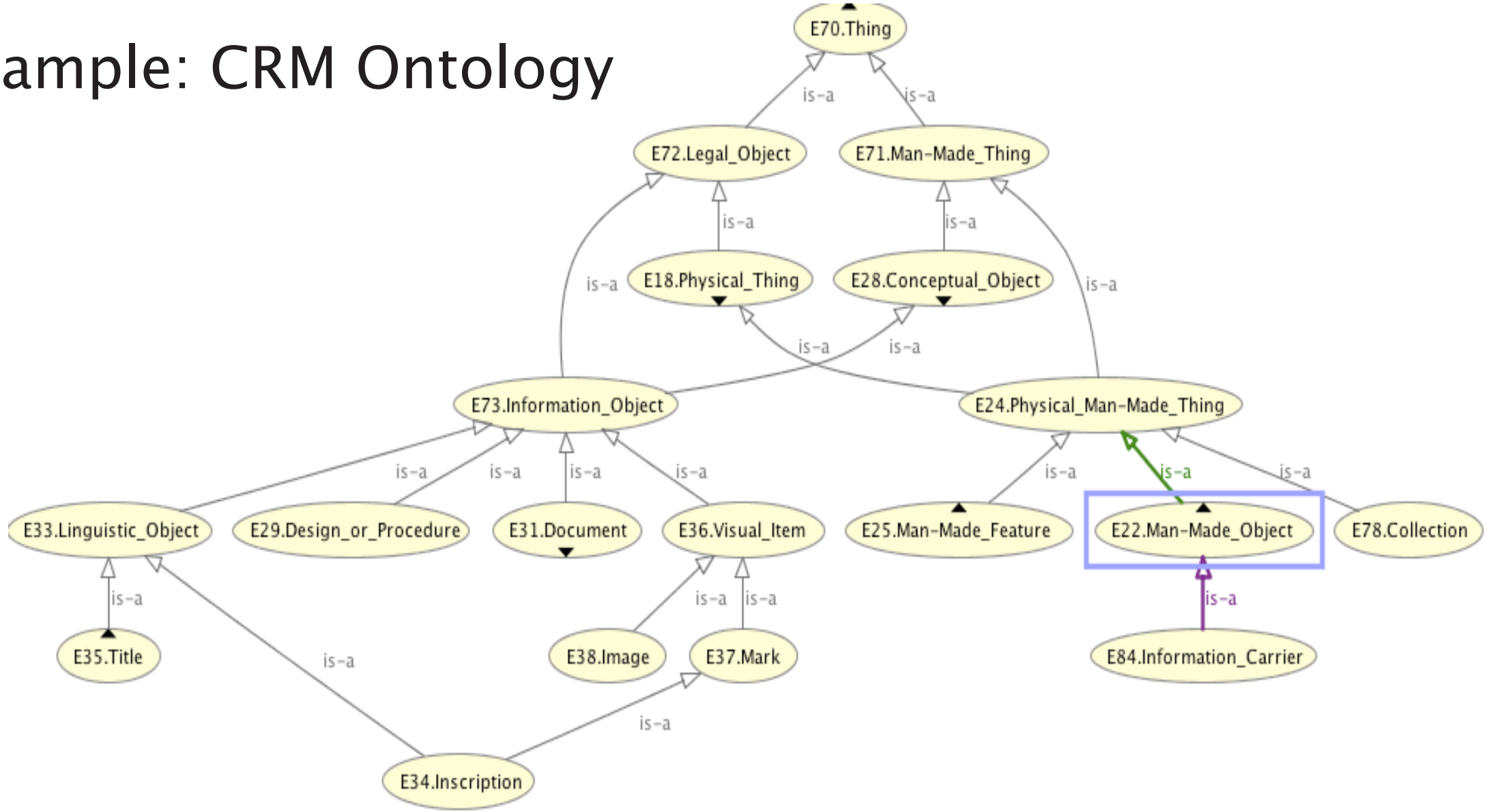
Describe a particular domain extensively

Domain dependent by definition

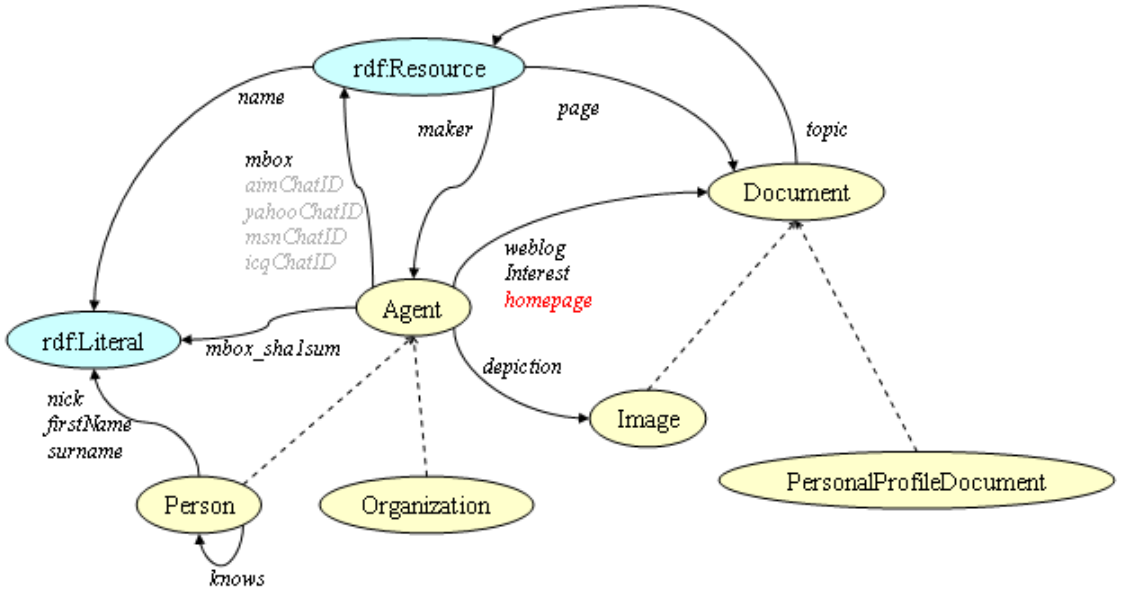
Examples:

- GO: Gene Ontology
  - Roughly 25K concepts
- CIDOC CRM: cultural heritage
  - Roughly 100 concepts
- FMA: Foundational Model of Anatomy
  - Around 75K concepts

Example: CRM Ontology



Application ontologies



Mainly designed to answer to the needs of a particular application

Scaled and focused to fit the application domain requirements

Examples:

- FOAF: Friend of a Friend ontology
  - about a dozen concepts
- ESWC06: for conference metadata
  - about 80 concepts, including FOAF

UNIVERSITY OF Southampton

Ontology Building Methodologies

Ontology Building Methodologies

No standard methodology for ontology construction

There are a number of methodologies and best practices

The following life cycle stages are usually shared by the methodologies:

- Specification - scope and purpose
- Conceptualisation - determining the concepts and relations
- Formalisation - axioms, restrictions
- Implementation - using some ontology editing tool
- Evaluation - measure how well you did
- Documentation - document what you did

Specification

Specifying the ontology's purpose and scope

- Why are you building this ontology?
- What will this ontology be used for?
- What is the domain of interest?
  - An ontology for car sales probably doesn't need to know much about types and prices of engine oil
- How much detail do you need?

Specification: Competency Questions

What are the questions you need the ontology to answer?

- These are competency questions
- Make a list of such questions and use as a check list when designing the ontology
- Helps to define scope, level of detail, evaluation, etc.

Specification: Competency Questions

The questions that you REALLY need

- You probably don't need to worry about the questions that "perhaps someone might need to ask someday"

The questions that CAN BE answered

- Can you get the necessary data to answer those questions?
- Permanent lack of some data may render parts of the ontology useless!

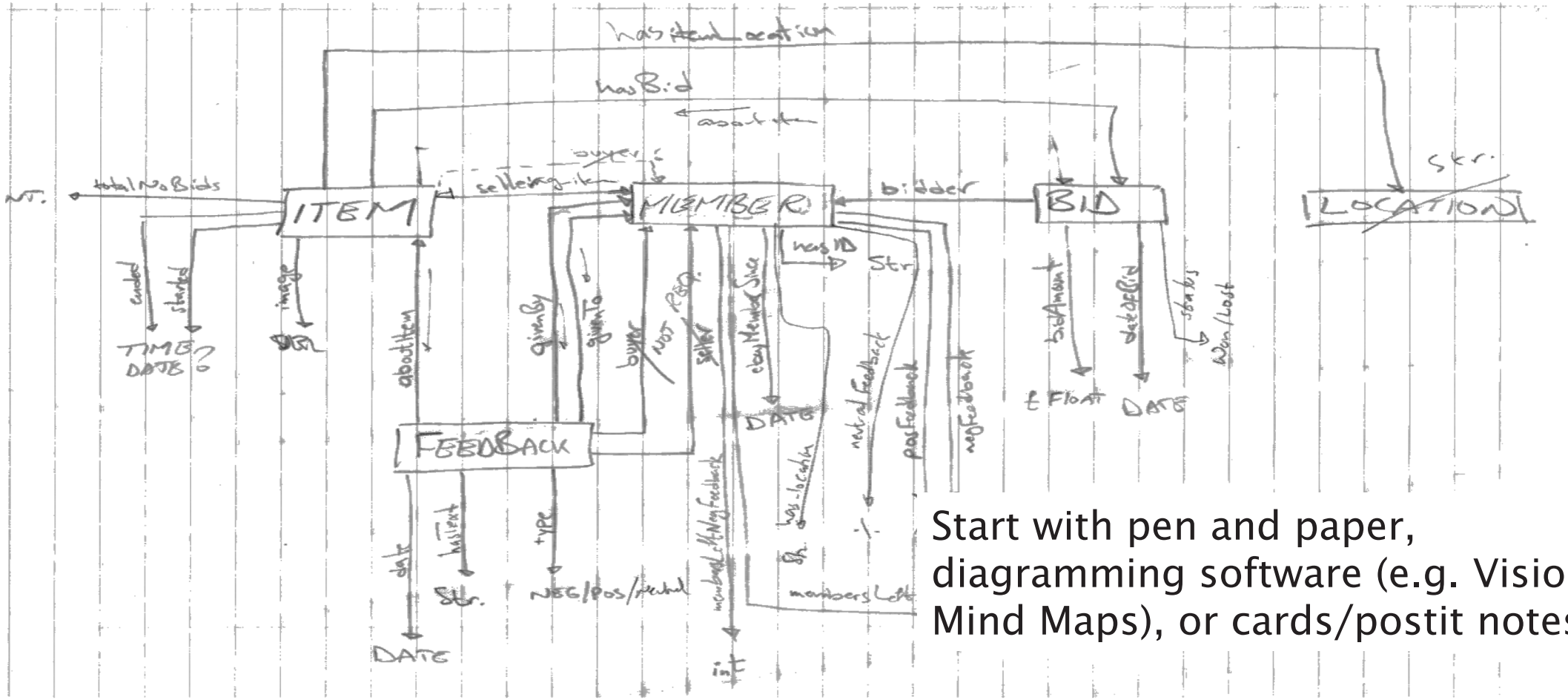
Conceptualisation

Identify the concepts to include in your ontology, and how they relate to each other

- Depends on your defined scope and competency questions
- Define unambiguous names and descriptions for classes and properties (more on this in Documentation)
- Reach agreement (the hard part!)

The best tool to use:

Conceptualisation



Start with pen and paper, diagramming software (e.g. Visio, Mind Maps), or cards/postit notes

Conceptualisation: Reuse

Ontologies are meant to be reusable!

- Technology for reusing ontologies is still limited

Always a good idea to check any existing models or ontologies

- Check your database models or off-the-shelf ontologies

Check existing ontologies

- No need to reinvent the wheel, unless it is easier to do so!
- Ontology search engines
  - Swoogle, Watson, lodlaundromat

What can you reuse?

- Databases
- Vocabularies
- Ontologies
  - Some much re-used ontologies
    - For describing persons: FOAF
    - For describing documents: Dublin Core
    - For describing social media: SIOC
    - For describing vocabulary hierarchies: SKOS
    - For describing e-commerce: Good Relations
    - For Web metadata: schema.org
    - …

Formalisation

- Moving from a list of concepts to a formal model
- Define the hierarchy of concepts and relations
- Also note down any restrictions
  - E.g. NonProfitOrg isDisjoint from ProfitOrg
  - An email address is unique

Formalisation: Building the Class Hierarchy

Top-down

- Start with the most general classes and finish with the most detailed classes

Bottom-up

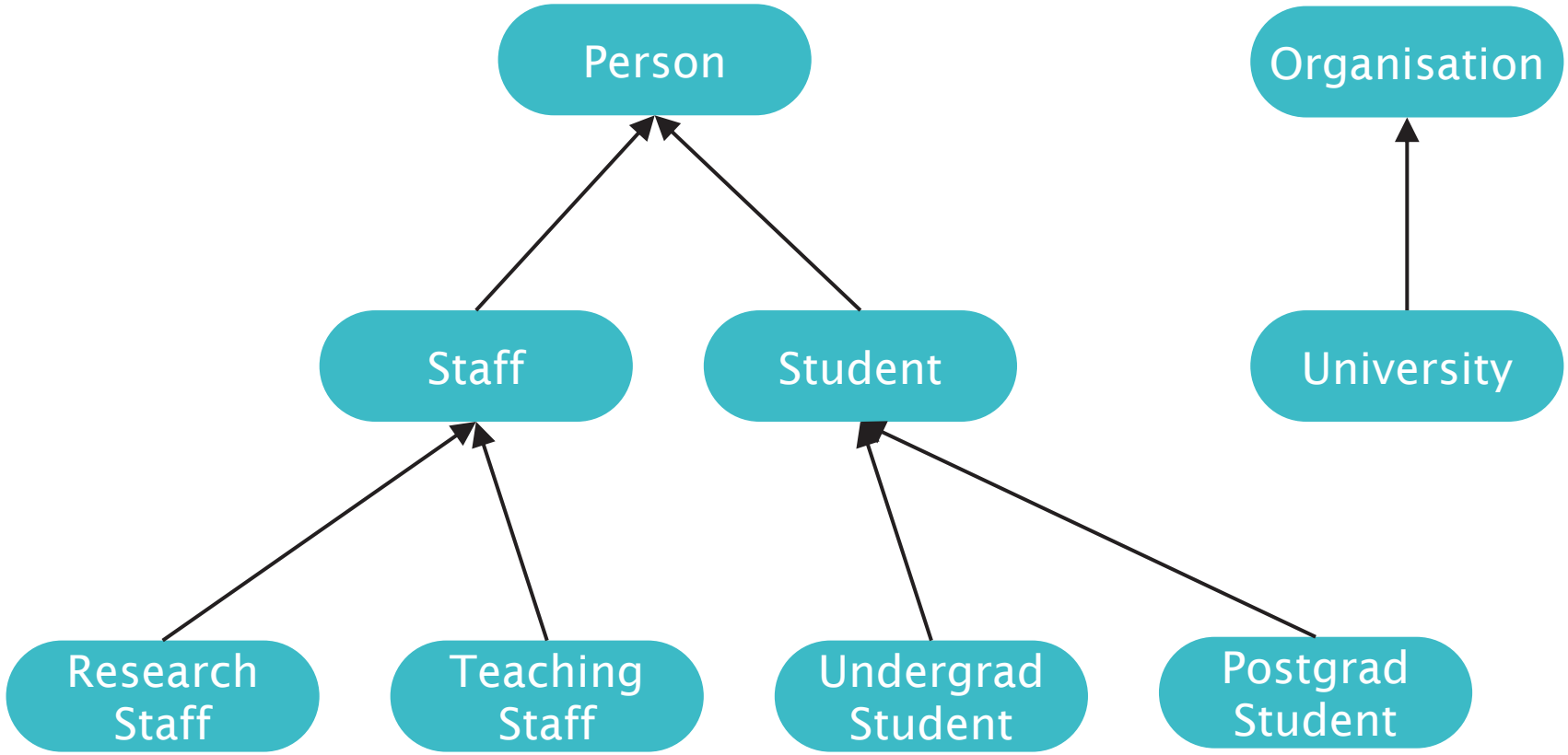
- Start with the most detailed classes and finish with the most general ones

Middle-out

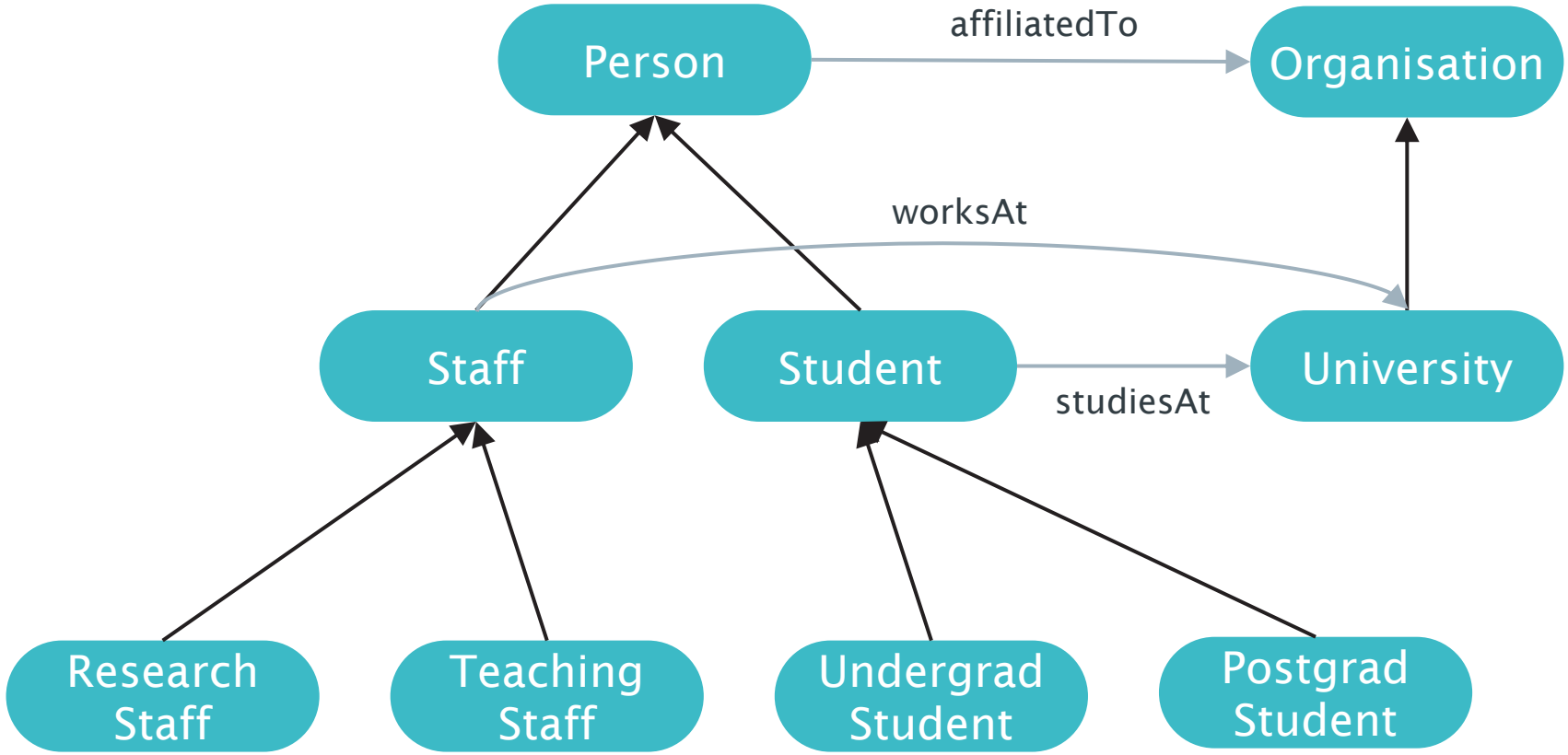
- Start with the most obvious classes
- Group as required
- Then go upwards and downwards to the more general and more detailed classes respectively
- Good for controlling scope and detail

Formalisation: Middle-Out Approach

Formalisation: Middle-Out Approach



Formalisation: Middle-Out Approach

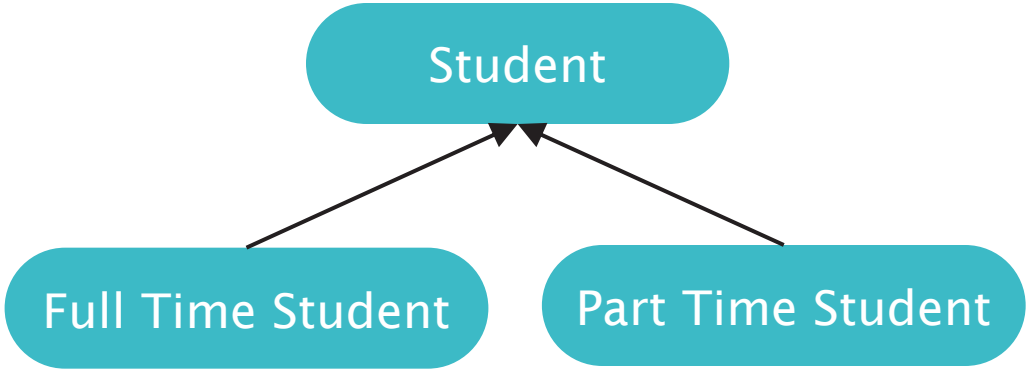


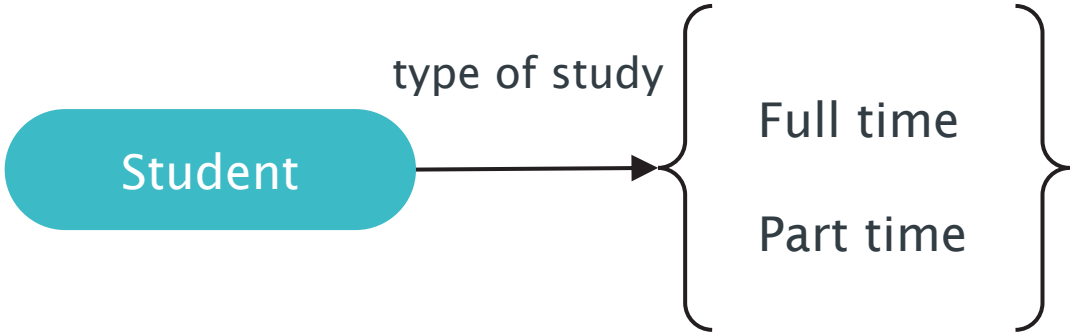
Formalisation: Naming Conventions

- Not rules, but conventions
- Avoid spaces and uncommon delimiters in class and relation names
  - E.g. use PetFood or Pet_Food instead of Pet Food or Pet*Food
- Capitalise each word in a class name
  - E.g. PetFood instead of Petfood or even petfood
- Start names of relations with a lowercase letter
  - E.g. pet_owner, petOwner
- Use singular nouns for classes
  - E.g. Pet, Person, Shop

Formalisation: Class or Relation?

Is it a class or a relation?



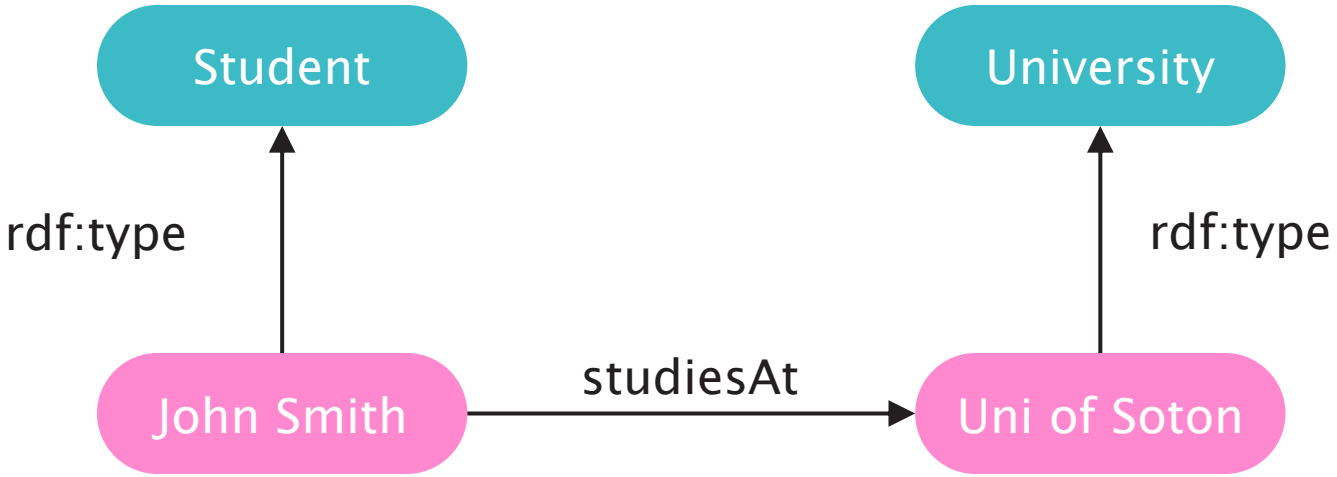


It depends!

If the subclass doesn't need any new relations (or restrictions), then consider making it a relation

Formalisation: Class or Instance?

Is it a class or an instance?

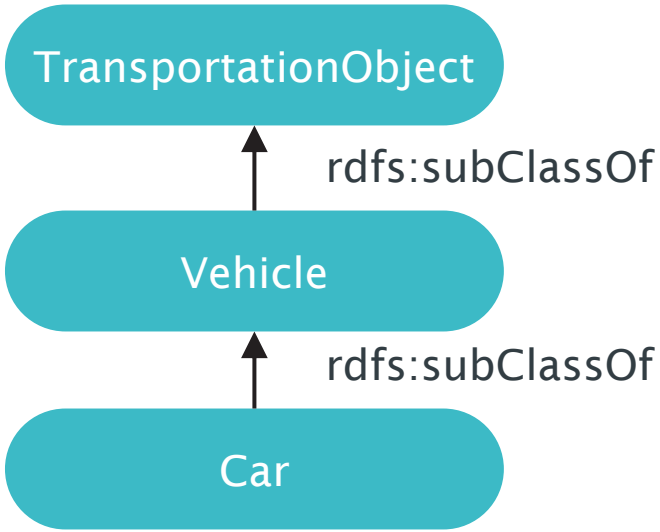


- If it can have its own instances, then it should be a class
- If it can have its own subclasses, then it should be a class

Formalisation: Transitivity of Class Hierarchy

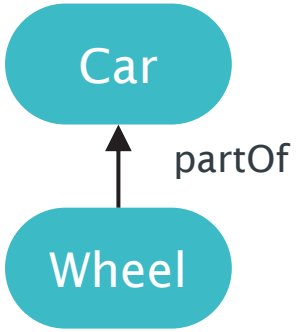
subClassOf relation is always transitive

- Car is a subclass of Vehicle
- Vehicle is a subclass of TransportationObject
- Any instance of Car is also a TransportationObject



subClassOf is not the same as "part of"

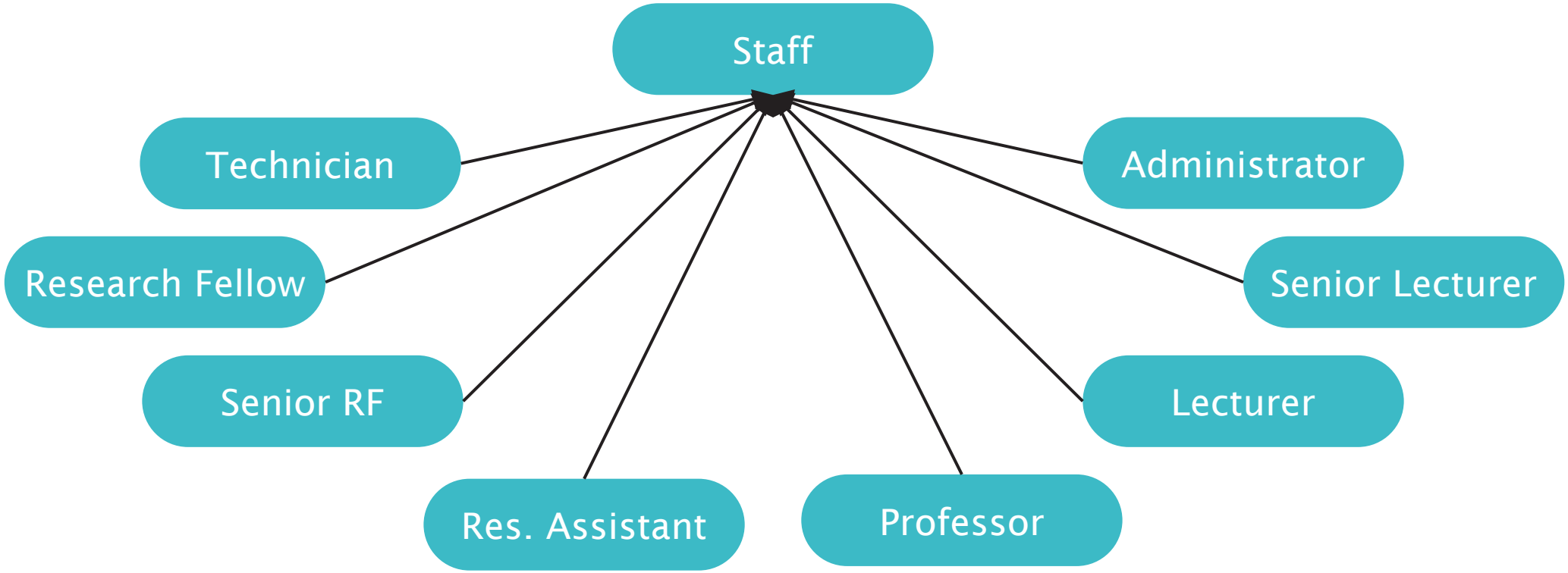
- (see meronymy pattern later this lecture)



Formalisation: Tidy Your Hierarchy

Avoid subClassOf clutter!

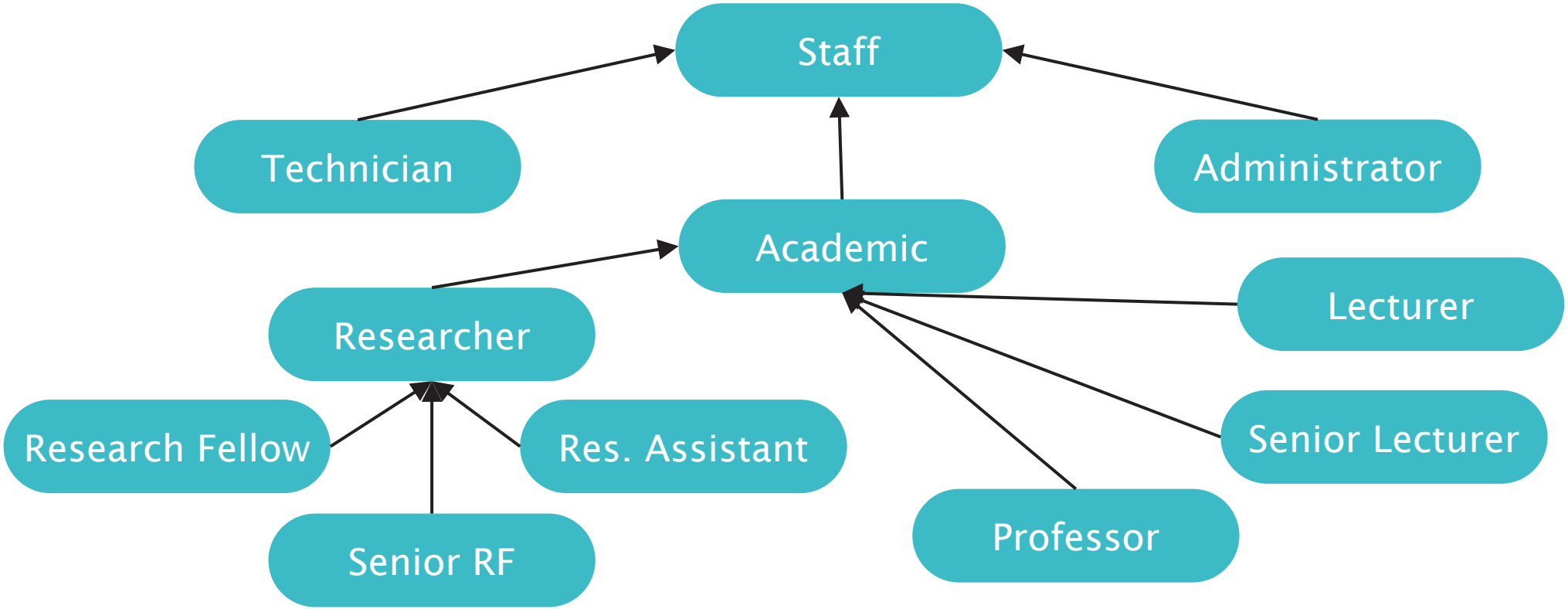
- Break down your hierarchy further if you have too many direct subclasses of a class



Formalisation: Tidy Your Hierarchy

Avoid subClassOf clutter!

- Break down your hierarchy further if you have too many direct subclasses of a class



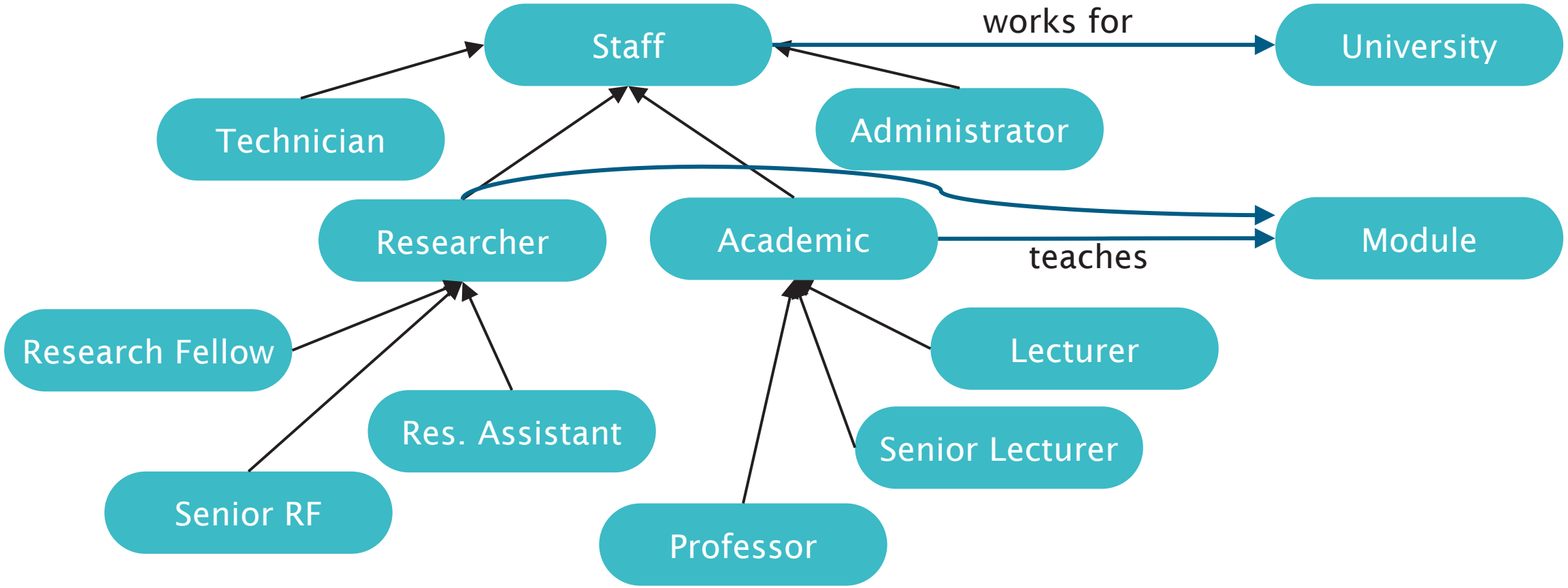
Formalisation: Where to Point my Relation?

Relations should point to the most general class

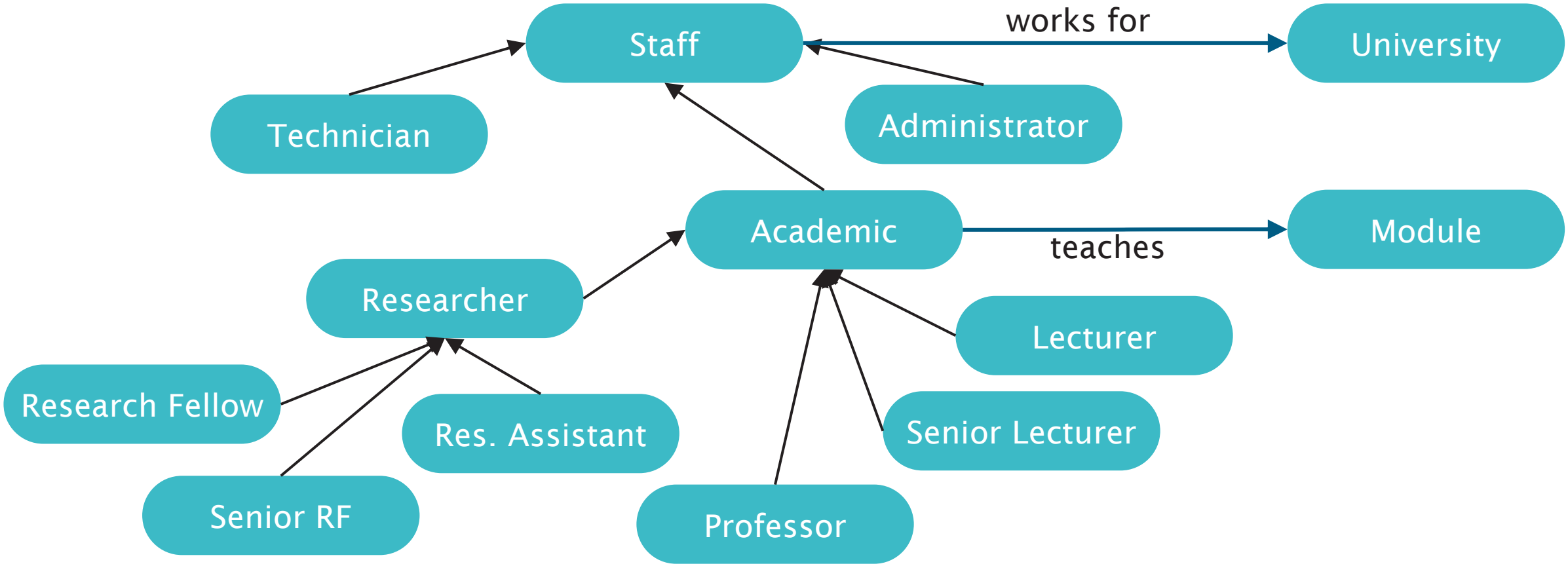
- But not too general
  - e.g relations pointing to Thing!
- And not too specific
  - e.g. relations pointing to the bottom of the hierarchy

As a rule of thumb, if the domain or range of a relation is a disjunction (union) of classes, some refactoring is probably required

Formalisation: Where to Point my Relation?



Formalisation: Where to Point my Relation?



Implementation

- Choose a language
  - e.g. RDFS, OWL...
- Implement it with an ontology editor
  - e.g. Protégé, SWOOP, TopQuadrant
- Edit the class hierarchy
- Add relationships
- Add restrictions
- Select appropriate value types, cardinality, etc
- Use a reasoner to check the consistency of your ontology
  - e.g. Racer, Pellet, Fact++, HermiT
  - Best to do this as you go along – easier to trace bugs in your modelling

Evaluation: Verification

Is your ontology correct?

- Is it syntactically correct?
- Is it consistent?

Implementing the ontology in an ontology editor helps to get the syntax correct

Using a reasoner helps you check that it's consistent

You can also validate your OWL ontology online:

- http://visualdataweb.de/validator/

Evaluation: Validation

Does your ontology successfully do what you set out to do?

Check the ontology against your competency questions

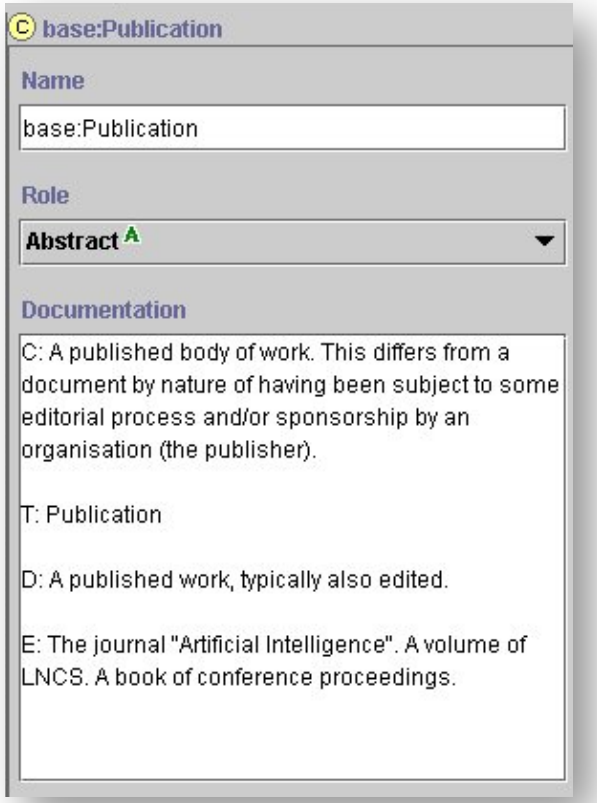
- Write the questions in SPARQL or in similar query languages
- Can you get the answers you need?
- Is it quick enough?
- Add additional properties or restructure the ontology to increase efficiency?

Documentation

Documenting the design and implementation rational is crucial for future usability and understanding of the ontology

- Rational, design options, assumptions, decisions, examples, etc.

Structured documentation may clarify these assumptions



Douglas Skuce proposed a convention for structured documentation of ontological assumptions in 1995

- Conceptual assumptions (C)
  (long definition, comparing with other classes/properties)
- Terminological assumptions (T) (alternative terms used)
- Definitional assumption (D) (short definition)
- Examples (E)

Structured documentation

Instead of putting C/T/D/E into a single rdfs:comment, structure the metadata using appropriate properties from RDFS and SKOS (import SKOS into your ontology)

Conceptual assumptions (C)

- skos:scopeNote, rdfs:comment

Terminological assumptions (T)

- skos:prefLabel, skos:altLabel, rdfs:label

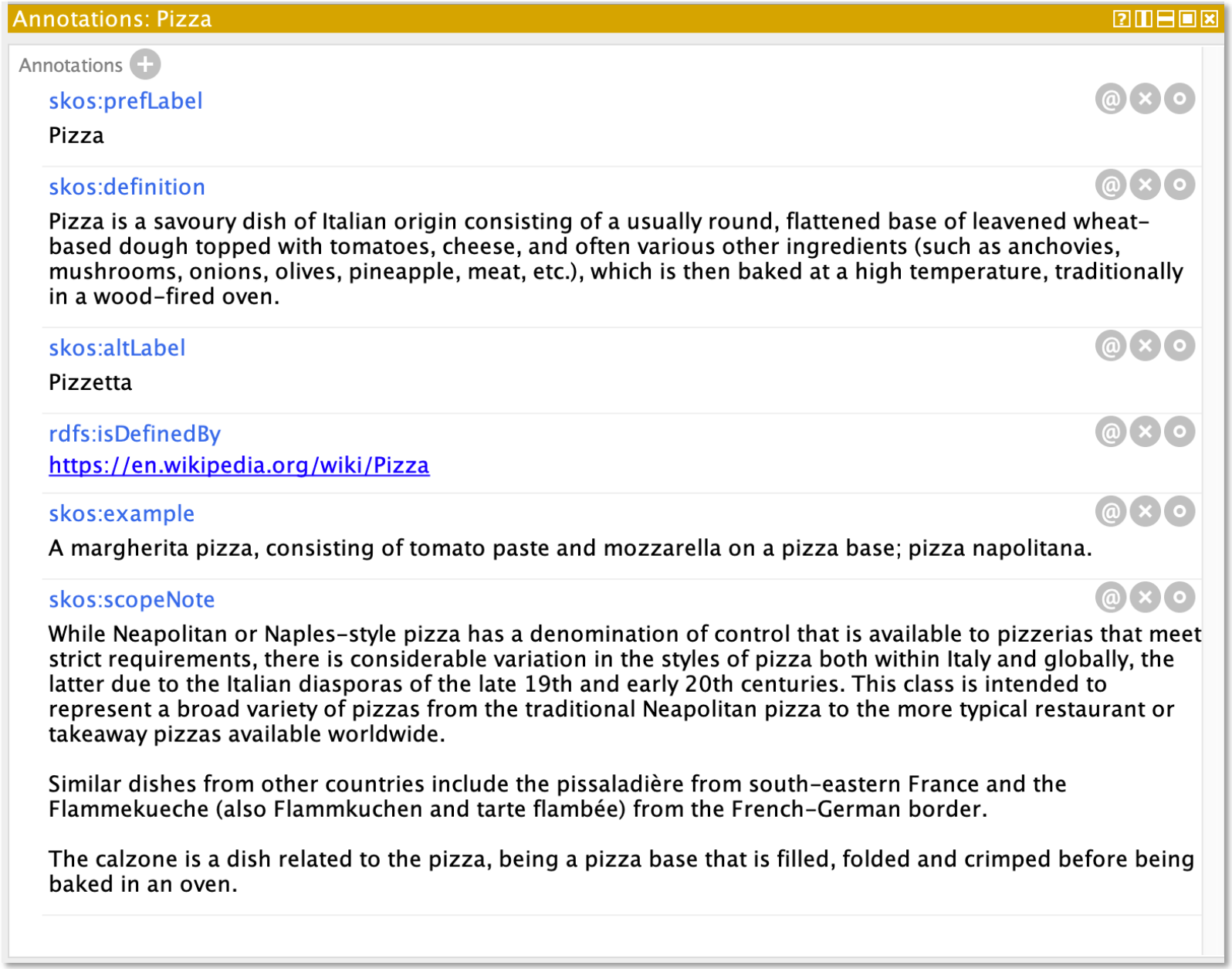
Definitional assumptions (D)

- skos:definition

Examples (E)

- skos:example

Use rdfs:isDefinedBy to indicate if definition is taken from an external source


Annotations: Pizza
Annotations
skos:prefLabel
Pizza
skos:definition
Pizza is a savoury dish of Italian origin consisting of a usually round, flattened base of leavened wheat-based dough topped with tomatoes, cheese, and often various other ingredients (such as anchovies, mushrooms, onions, olives, pineapple, meat, etc.), which is then baked at a high temperature, traditionally in a wood-fired oven.
skos:altLabel
Pizzetta
rdfs:isDefinedBy
https://en.wikipedia.org/wiki/Pizza
skos:example
A margherita pizza, consisting of tomato paste and mozzarella on a pizza base; pizza napolitana.
skos:scopeNote
While Neapolitan or Naples-style pizza has a denomination of control that is available to pizzerias that meet strict requirements, there is considerable variation in the styles of pizza both within Italy and globally, the latter due to the Italian diasporas of the late 19th and early 20th centuries. This class is intended to represent a broad variety of pizzas from the traditional Neapolitan pizza to the more typical restaurant or takeaway pizzas available worldwide.
Similar dishes from other countries include the pissaladière from south-eastern France and the Flammekueche (also Flammkuchen and tarte flambée) from the French-German border.
The calzone is a dish related to the pizza, being a pizza base that is filled, folded and crimped before being baked in an oven.

Summary

Ontology construction is an iterative process

- Build ontology, try to use it, fix errors, extend, use again, and repeat

There is no single correct model for your domain

- The same domain may be modelled in several ways

Following best practices helps to build good ontologies

- Well scoped, documented, structured

Reuse existing ontologies if possible

- Check your database models and existing ontologies
- Reuse or learn from existing representations
- (most ontology editing tools don't yet provide good support for reuse)

Common Pitfalls

Over scaling and complicating your ontology

- Need to learn when to stop expanding the ontology

Lack of documentation

- For the design rationale, vocabulary and structure decisions, intended use, etc.

Redundancy

- Increase chances of inconsistencies and maintenance cost

Using ambiguous terminology

- Others might misinterpret your ontology
- Mapping to other ontologies will be more difficult

UNIVERSITY OF Southampton

Ontology Design Patterns

Design Patterns

Patterns are general, reusable solution to commonly occurring problems

- Concept originated with Christopher Alexander's work on architecture
- Popularised in software engineering by the "gang of four"
- Subject of study by the knowledge engineering community

Design Patterns for the Semantic Web

N-ary relations

- How can we say more about a relation instance?
- How do we represent an ordered sequence of relations?

Value partitions and value sets

- How do we represent a fixed list of values?

Part-whole hierarchies

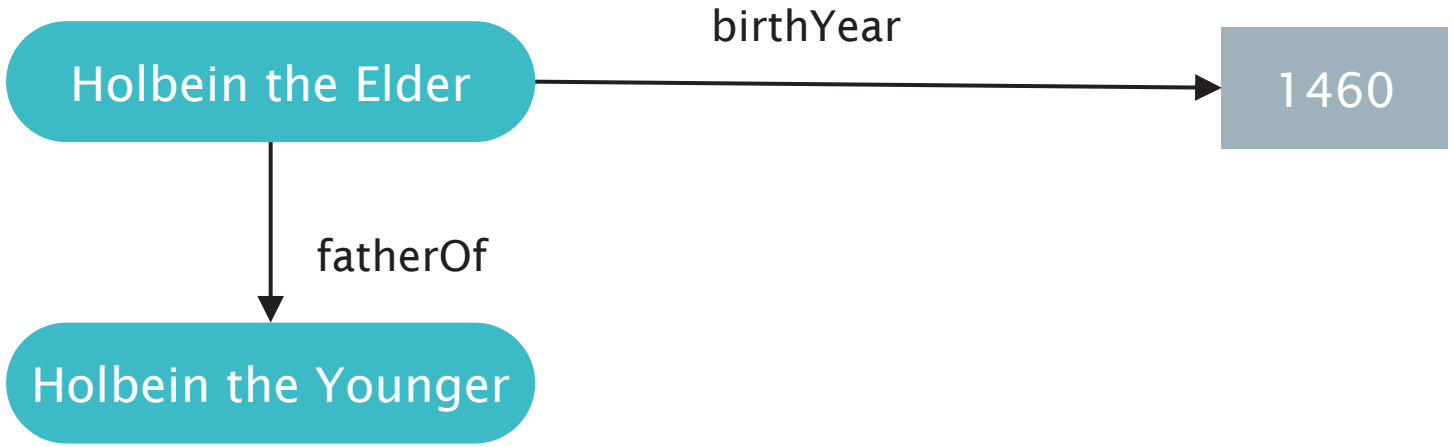
- How do we represent hierarchies other than the subclass hierarchy?

UNIVERSITY OF Southampton

N-ary Relations

Binary Relations

In RDF and OWL, binary relations link two individuals, or an individual and a value



The properties birthYear and fatherOf are binary relations

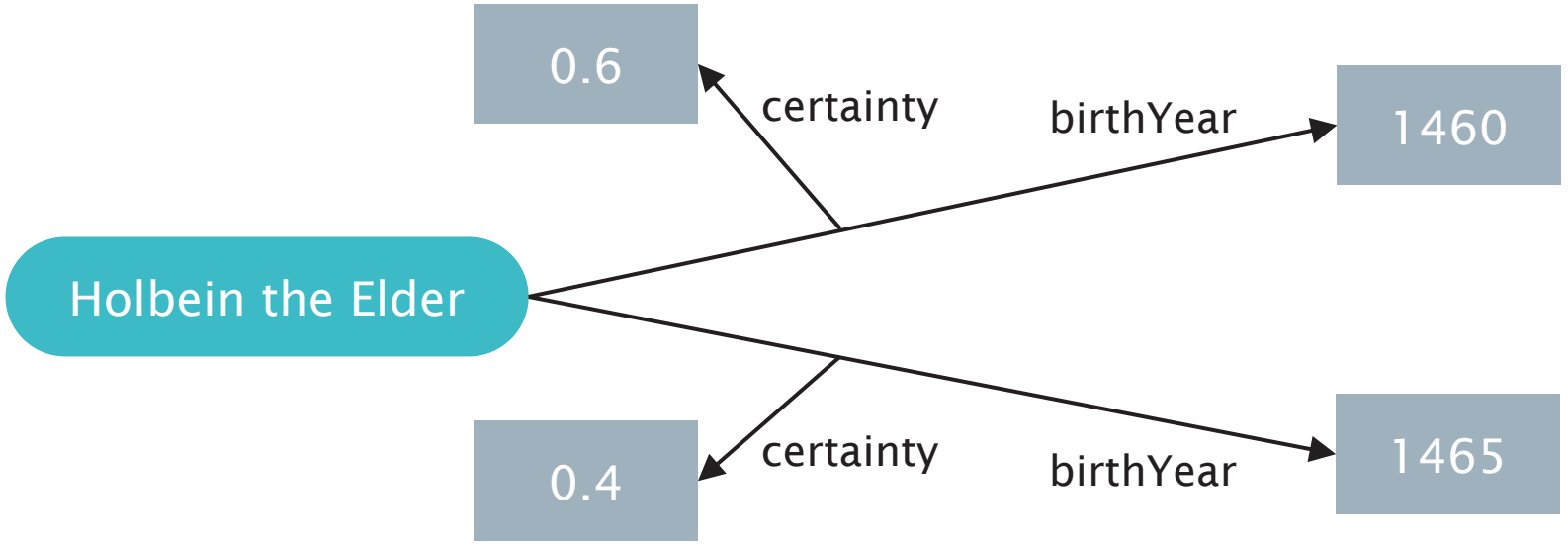
Relations with Additional Information

In some cases, we need to associate additional info with a binary relation

- e.g. certainty, strength, dates

For example, Holbein the Elder's date of birth is unconfirmed

- He was born in either 1460 or 1465
- How can we represent this uncertainty?



N-ary Relations

N-ary relations link an individual to more than a one value

Possible use cases:

1. A relation needs additional info
   e.g. a relation with a rating value
2. Two binary relations are related to each other
   e.g. body_temp (high, normal, low), and trend (rising, falling)
3. A relation between several individuals
   e.g. someone buys a book from a bookstore
4. Linking from, or to, an ordered list of individuals
   e.g. an airline flight visiting a sequence of airports

N-ary Relation Patterns

Pattern 1: Reified relation

- Use for cases 1, 2, and 3 above

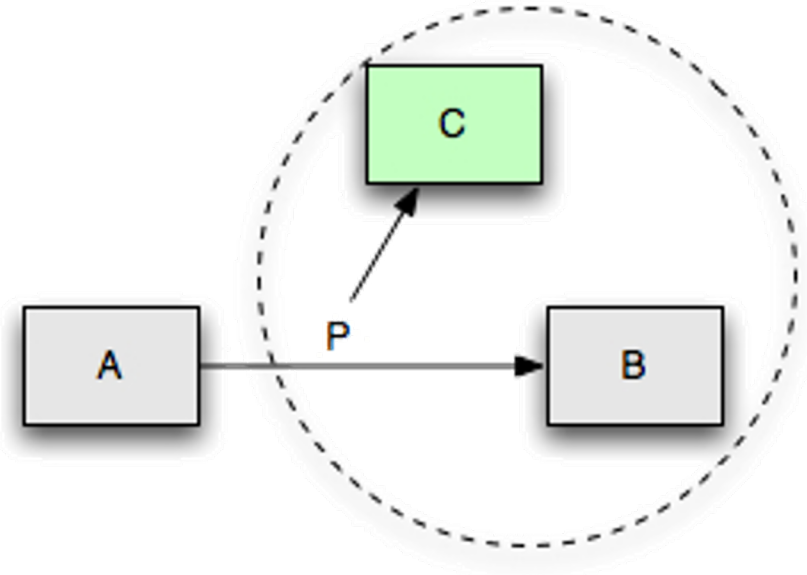
Pattern 2: Sequence of arguments

- For case 4

Pattern 1: Reified Relation

To represent additional information about a relation:

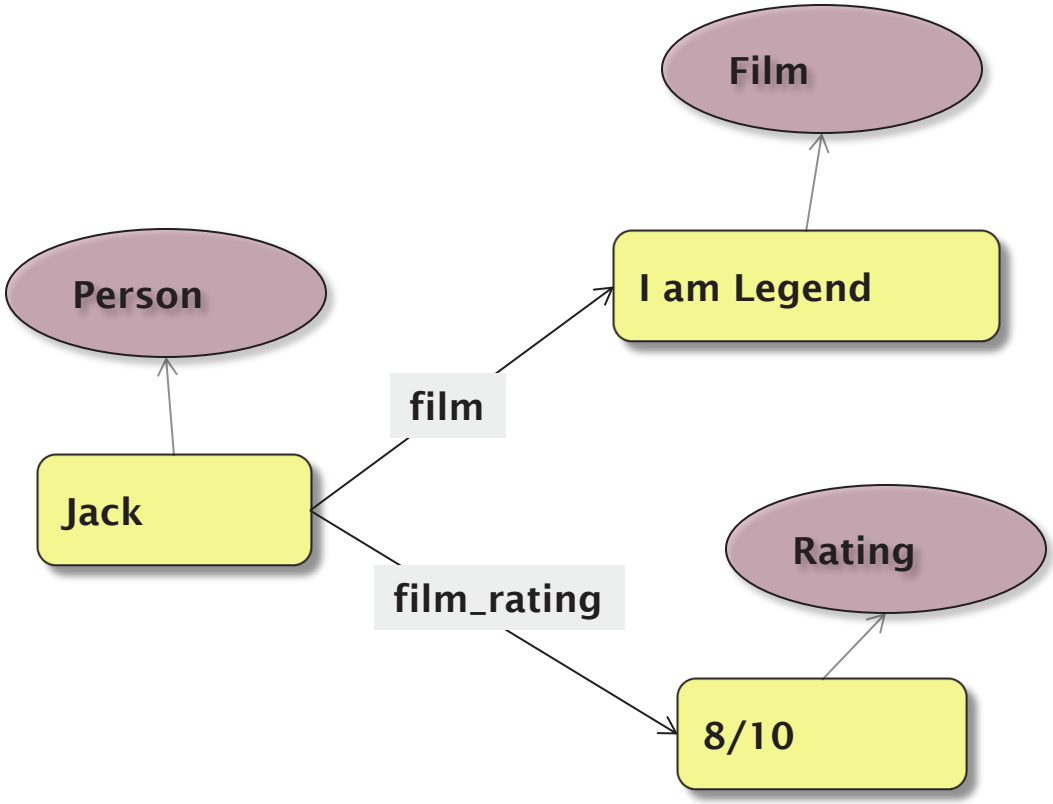
- Create a new class to represent the relation
- Individuals of this class are instances of the relation
- Relation class can have additional properties to describe more information about the relation



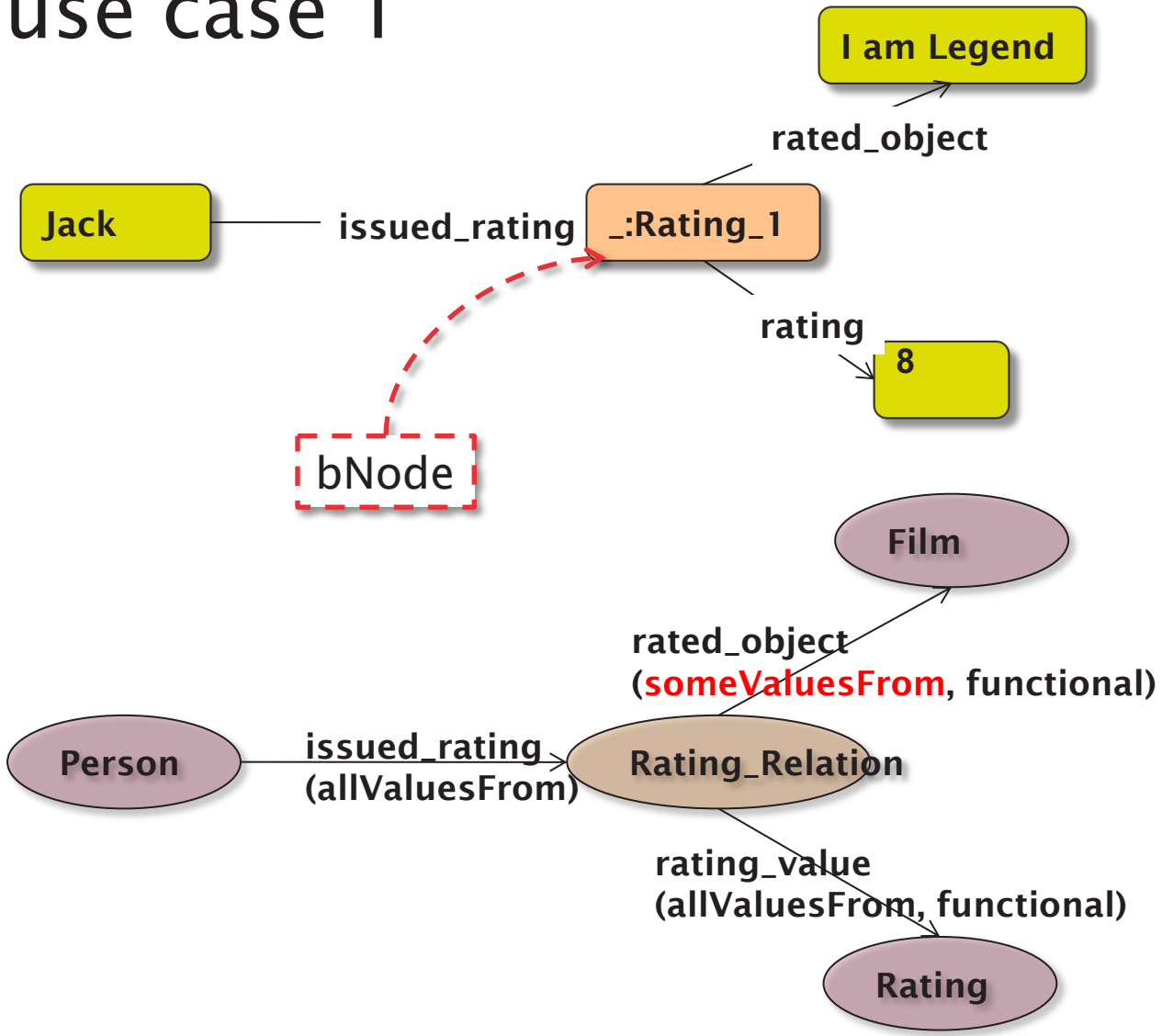
Use case 1: additional information

Jack has given the film ‘I Am Legend’ a four-star rating

- We need to represent a quantitative value to describe the rating relation



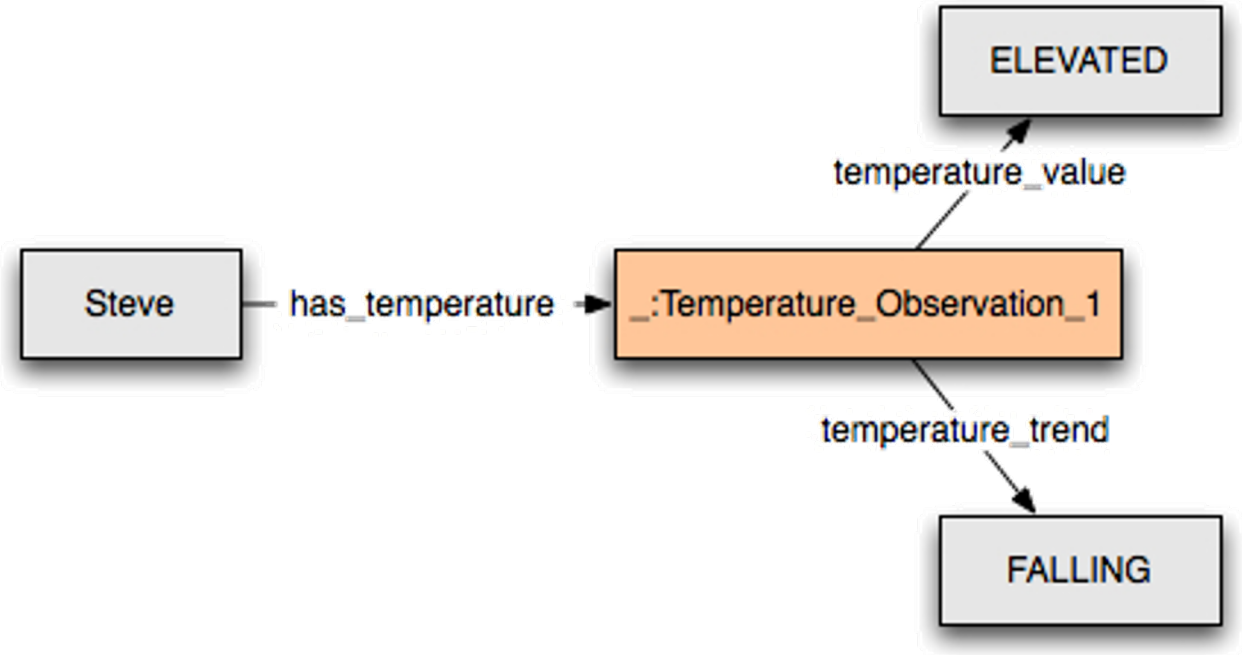
Solution for use case 1



Use case 2: different aspects of a relation

Steve has a temperature which is high, but falling

- We need to represent different aspects of the temperature that Steve has

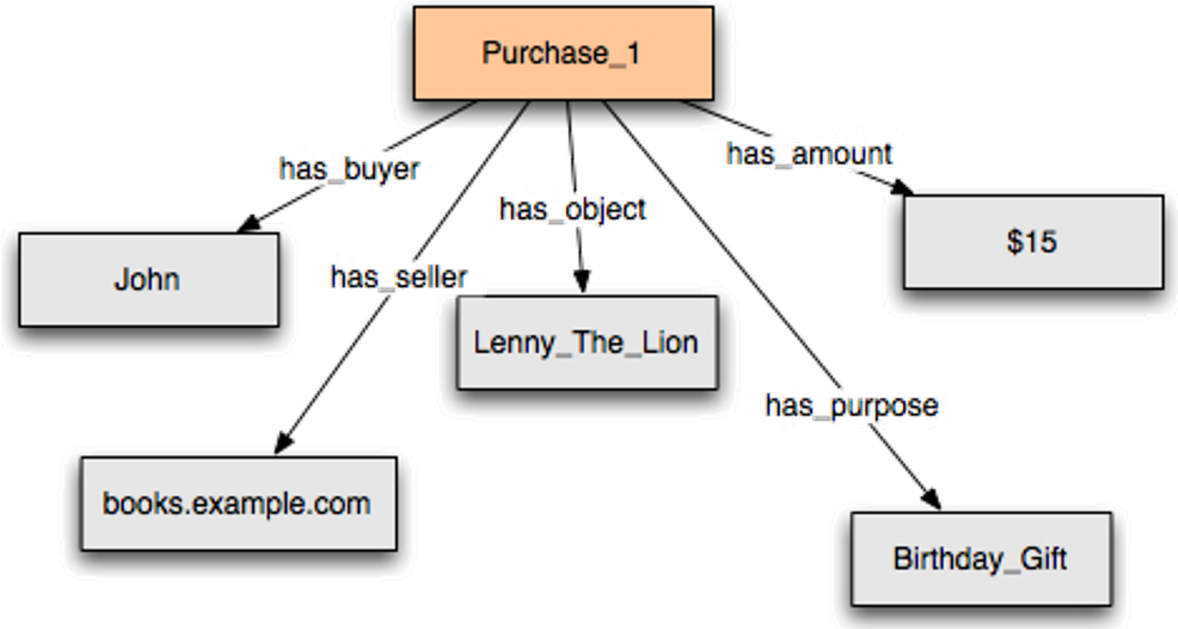


Source: W3C

Use case 3: no distinguished participant

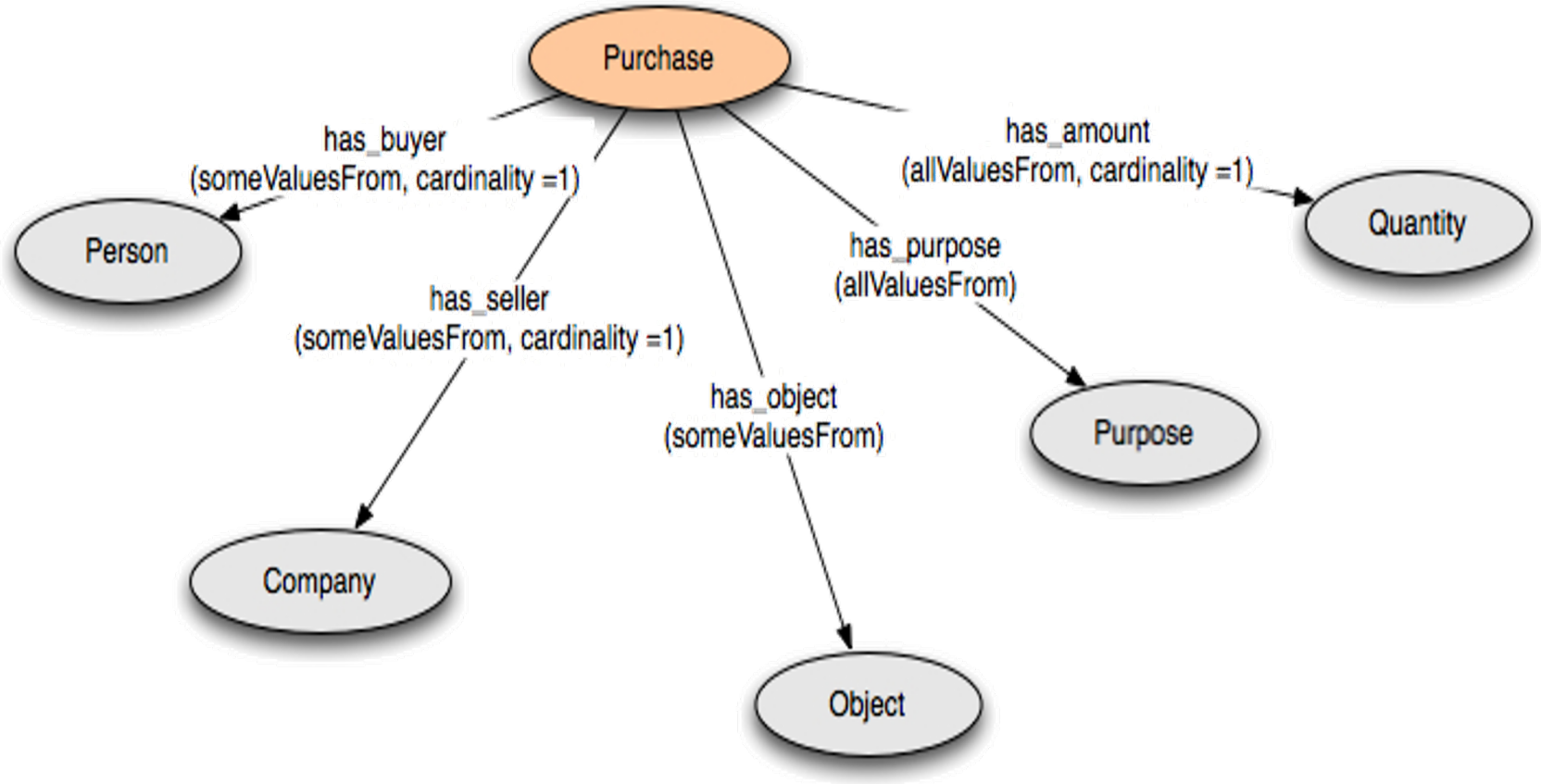
John buys a “Lenny the Lion” book from books.example.com for $15 as a birthday gift

- No distinguished subject for the relation
- i.e. no primary relation to convert into a Relation Class as in cases 1 and 2



Source: W3C

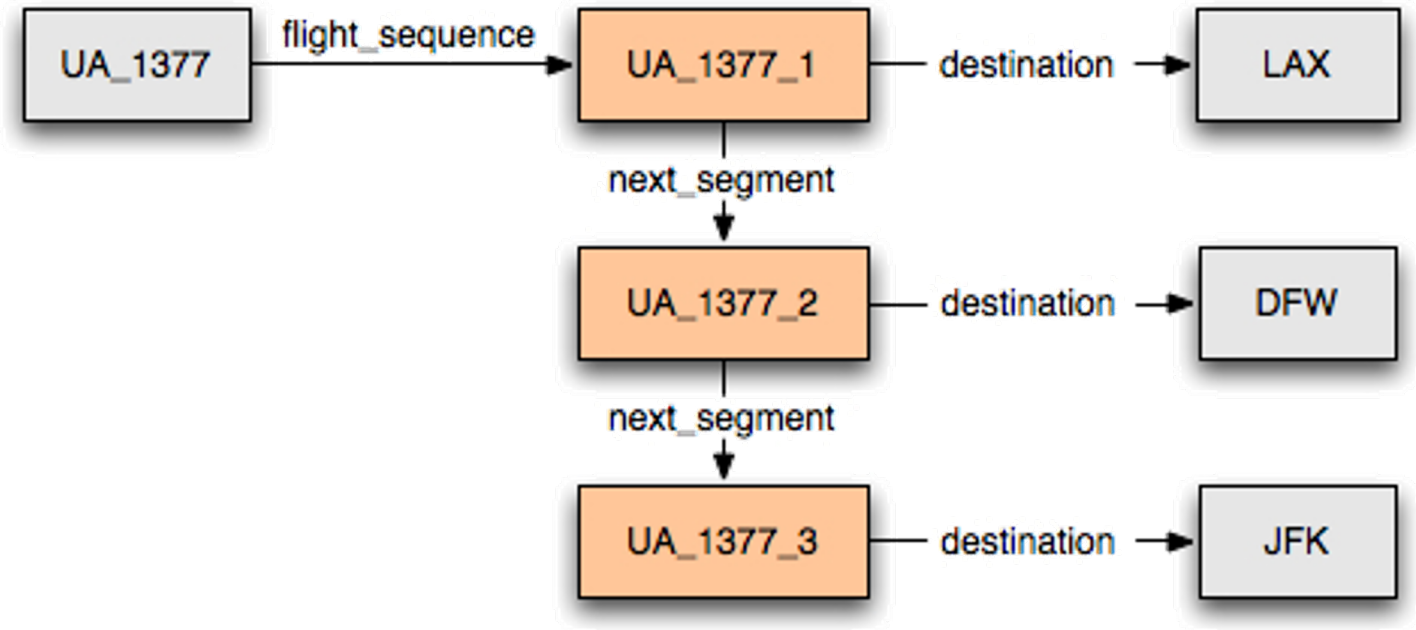
Solution for use case 3



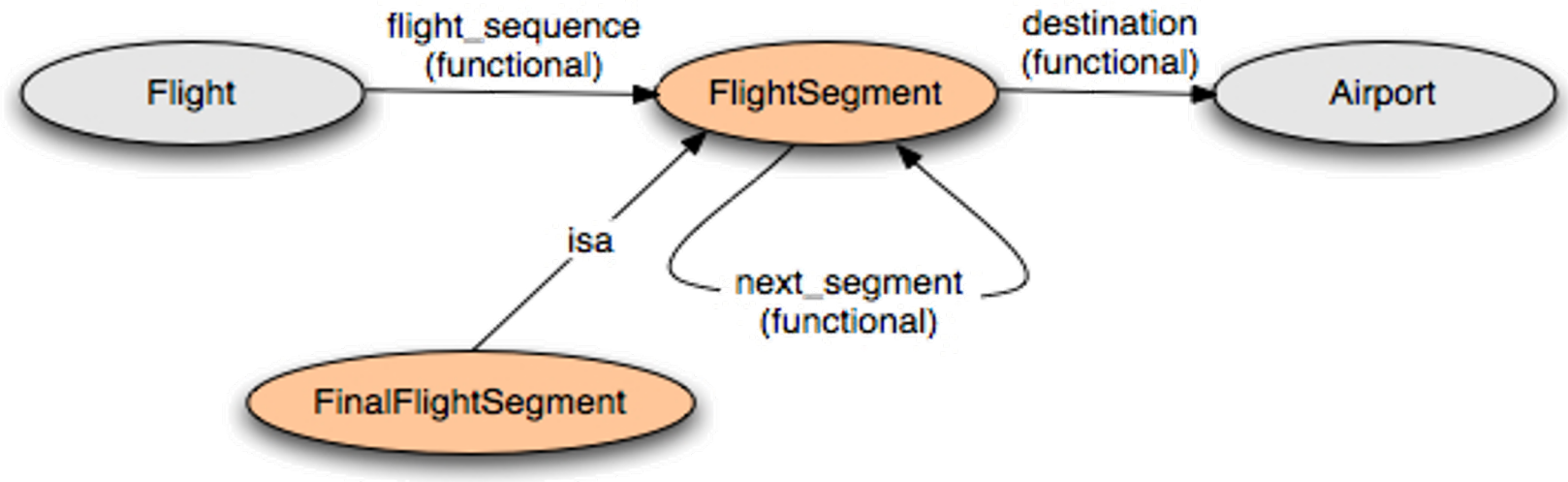
Pattern 2: Sequence of arguments

United Airlines, flight 1377 visits the following airports: LAX, DFW, and JFK

- For such an example, we need to represent a sequence of arguments



Pattern 2: Sequence of arguments




```
:FinalFlightSegment a owl:Class ;
        rdfs:comment "The last flight segment has no next_segment";
        rdfs:subClassOf :FlightSegment ;
        rdfs:subClassOf
                [ a owl:Restriction ; owl:maxCardinality "0";
                    owl:onProperty :next_segment] .
```

UNIVERSITY OF Southampton

Value Partitions and Value Sets

Descriptive Features

Descriptive features are quite common in ontologies:

- Size = {small, medium, large}
- Risk = {dangerous, risky, safe}
- Health status = {good health, medium health, poor health}

Also called “qualities”, “modifiers” and “attributes”

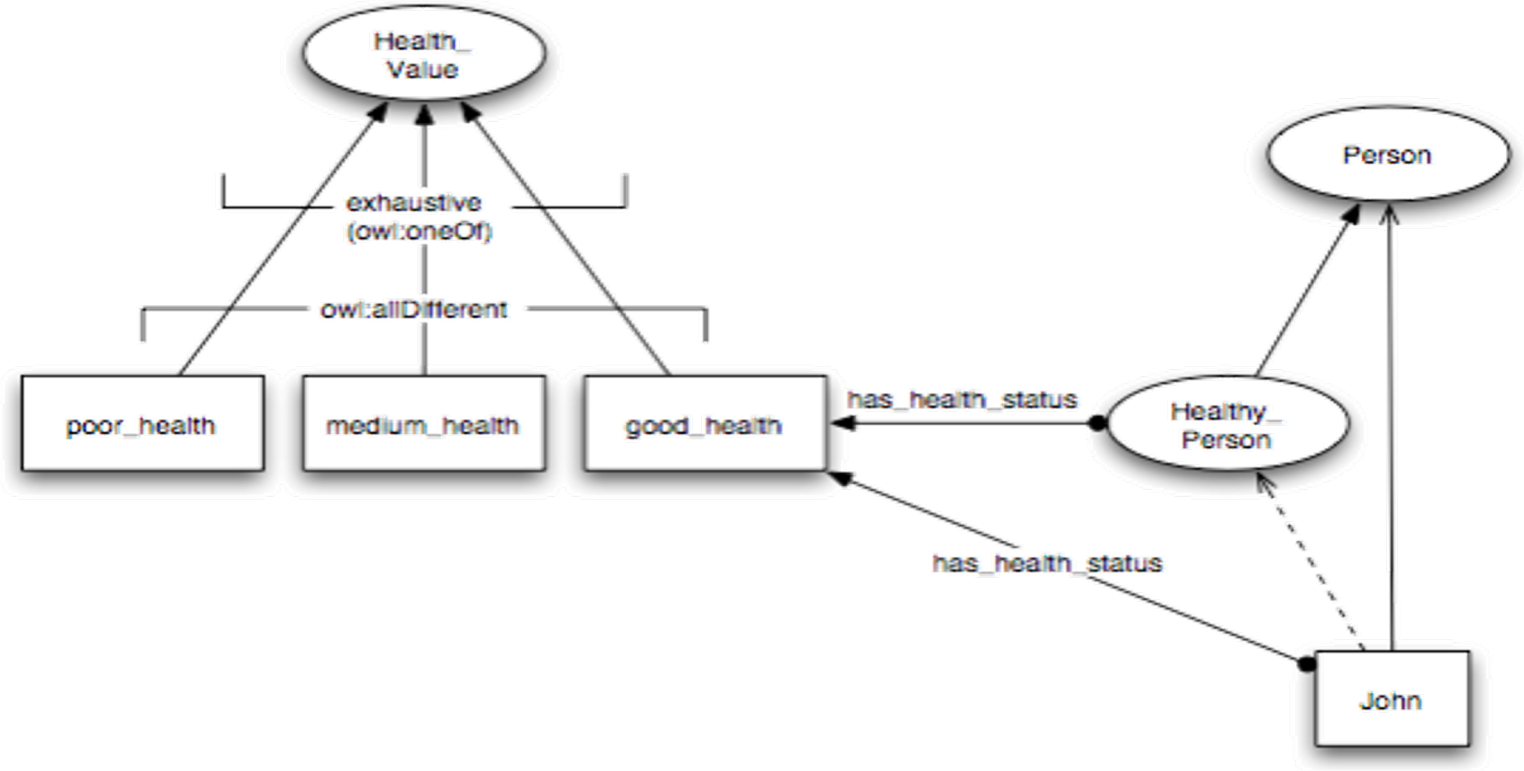
- A property can have only one value for each feature to ensure consistency

Three main approaches:

- Enumerated individuals (a value set)
- Disjoint classes (a value partition)
- Datatype values (not considered in this lecture)

Value Sets

Values of descriptive feature are individuals



Value Sets

A health value can be either poor, medium or good:

$$HealthValue \equiv \{ poorHealth, mediumHealth, goodHealth \}$$

Poor, medium and good are all different from each other:

$$poorHealth \neq mediumHealth$$
$$poorHealth \neq goodHealth$$
$$mediumHealth \neq goodHealth$$

A healthy person is a person who has some health status which is the value good:

$$HealthyPerson \equiv Person \sqcap \exists hasHealthStatus.\{ goodHealth \}$$

Notes on Value Sets

Need axioms to set the three health values to be different from each other

- This way, a person cannot have more than one health value at a time

Values cannot be further partitioned

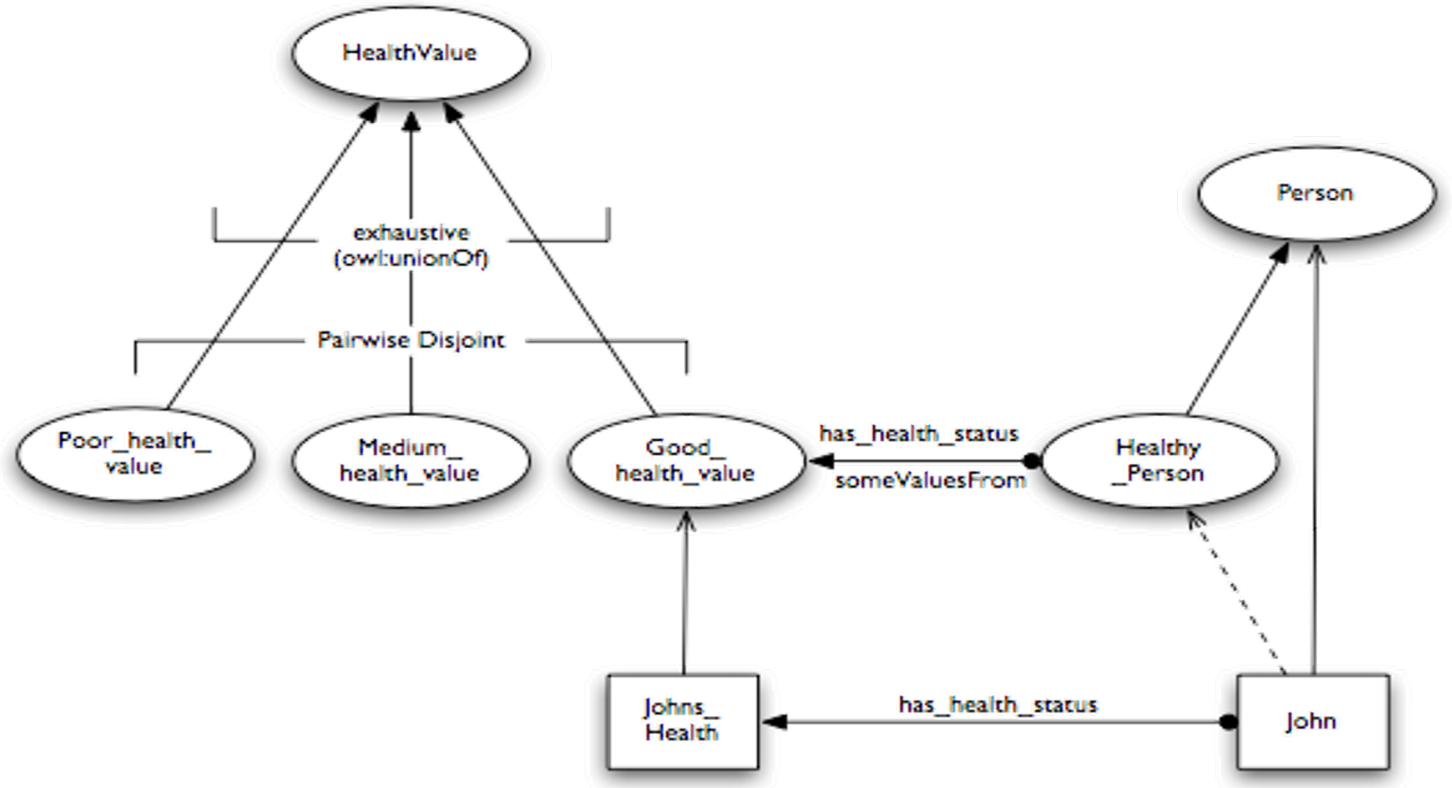
- e.g. cannot have fairly_good_health as a subtype of good_health

Only one set of values is allowed for a feature

- The class HealthValue cannot be equivalent to more than one set of distinct values
- Doing so will cause inconsistencies

Value Partitions

Values of descriptive features are disjoint subclasses:



Value Partitions

Poor, medium and good are types of health value:

$$PoorHealth \sqsubseteq HealthValue$$
$$MediumHealth \sqsubseteq HealthValue$$
$$GoodHealth \sqsubseteq HealthValue$$

Covering axiom (the only types of health value are poor, medium and good):

$$HealthValue \equiv PoorHealth \sqcap MediumHealth \sqcap GoodHealth$$

Poor, medium and good are pairwise disjoint:

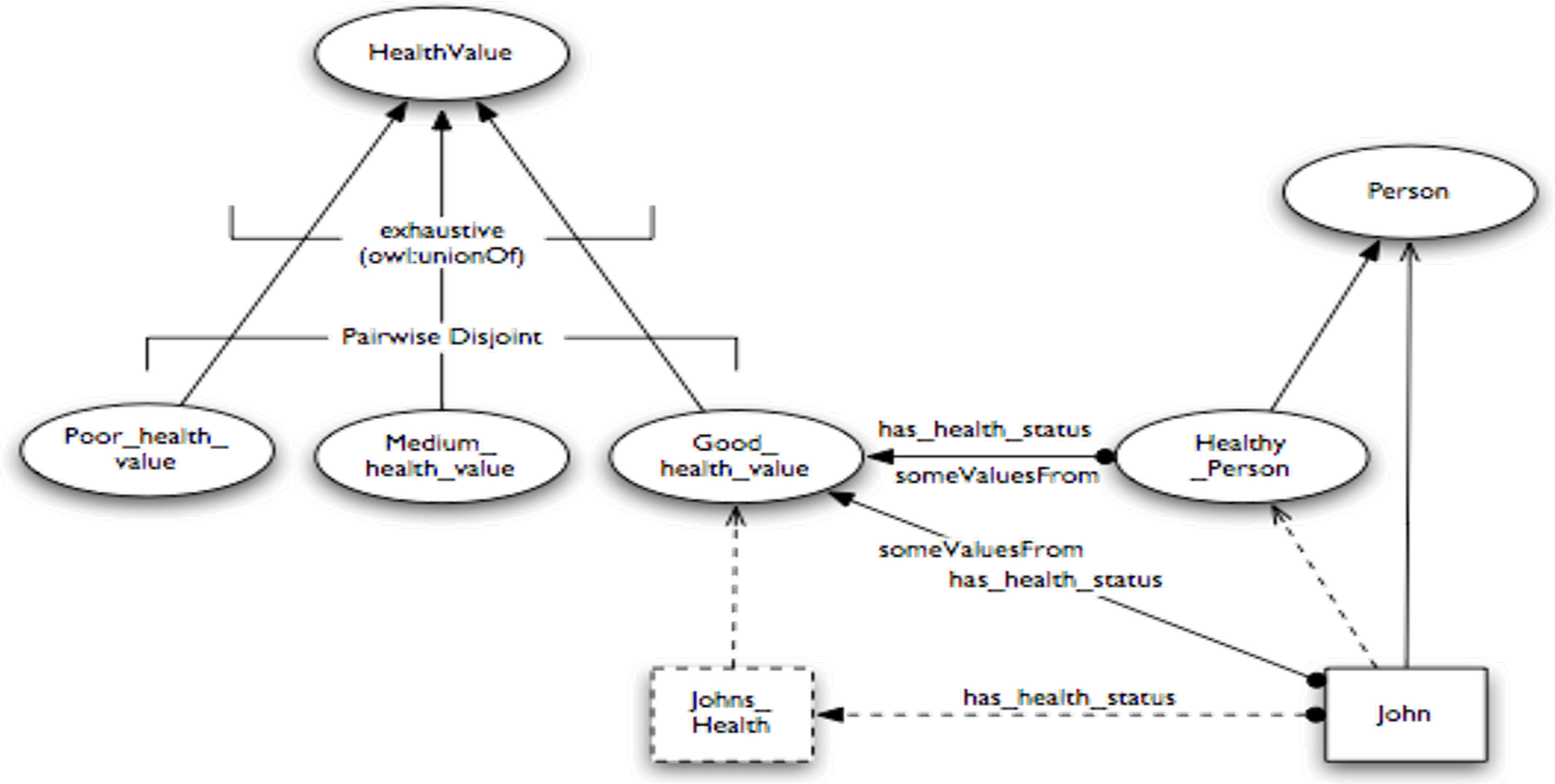
$$PoorHealth \sqcap MediumHealth \equiv \bot$$
$$PoorHealth \sqcap GoodHealth \equiv \bot$$
$$MediumHealth \sqcap GoodHealth \equiv \bot$$

A healthy person is a person who has some health status which is an instance of good

$$HealthyPerson \equiv Person \sqcap \exists hasHealthStatus.GoodHealth$$

Value Partitions

The instance JohnsHealth can be made anonymous



Notes on Value Partitions

Values can be further partitioned

- Simply add subclasses to the value classes

Can have alternative partitions of the same feature

OWL 2 contains specific support for defining disjoint unions

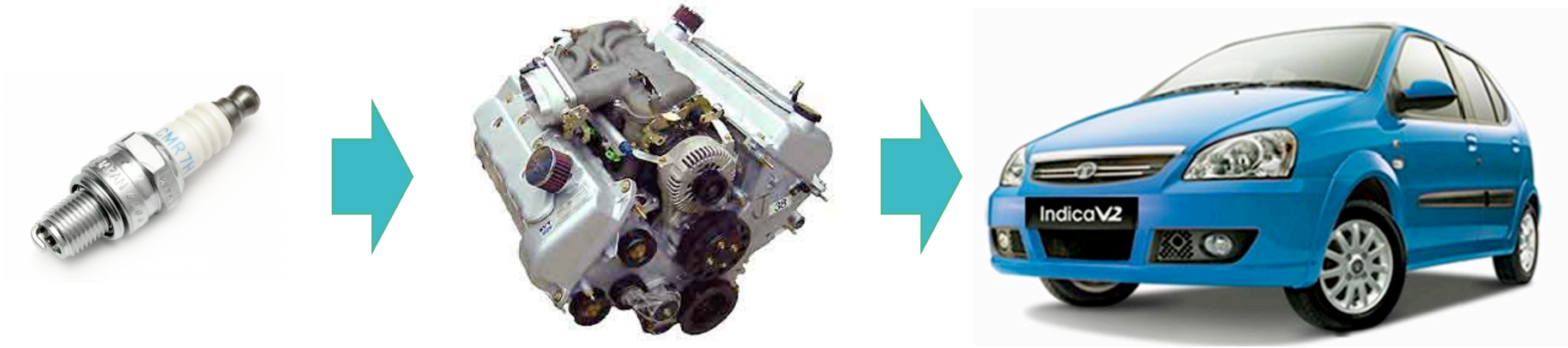
$$C \equiv C_1 \sqcup C_2 \sqcup \cdots \sqcup C_n$$
$$C_1 \sqcap C_2 \equiv \bot$$
$$C_1 \sqcap C_3 \equiv \bot$$
$$\ldots$$
$$C_{n-1} \sqcap C_n \equiv \bot$$

UNIVERSITY OF Southampton

Part-Whole Hierarchies

Meronymies (part-whole relations)

Taxonomies are not the only hierarchical relation that we wish to model



- A spark plug isn't a kind of engine (class-instance)
- A spark plug is a **part of** an engine

Simple Part-Whole Representation

We need two properties:

- partOf (a transitive property)
- directPartOf (a subproperty of partOf)

$$\text{part of} \circ \text{partOf} \sqsubseteq \text{partOf}$$
$$\text{directPartOf} \sqsubseteq \text{partOf}$$

Part-Whole Hierarchies

Represent part-whole relationships between classes using existential restrictions:

Every spark plug is a direct part of some engine:	$\mathrm{SparkPlug} \sqsubseteq \exists \mathrm{directPartOf}.\,\mathrm{Engine}$
Every engine is a direct part of some car:	$\mathrm{Engine} \sqsubseteq \exists \mathrm{directPartOf}.\,\mathrm{Car}$
Every wheel is a direct part of some car:	$\mathrm{Wheel} \sqsubseteq \exists \mathrm{directPartOf}.\,\mathrm{Car}$

Defining Classes of Parts

Extend the ontology with classes of parts for each level, so that the reasoner can automatically derive a class hierarchy:

A car part is a part of some car: $\mathrm{CarPart} \equiv \exists \mathrm{partOf}.\,\mathrm{Car}$

A direct car part is a direct part of some car: $\mathrm{DirectCarPart} \equiv \exists \mathrm{directPartOf}.\,\mathrm{Car}$

An engine part is a part of some engine: $\mathrm{EnginePart} \equiv \exists \mathrm{partOf}.\,\mathrm{Engine}$

A reasoner will infer that $\mathrm{EnginePart} \sqsubseteq \mathrm{CarPart}$ (but not $\mathrm{EnginePart} \sqsubseteq \mathrm{DirectCarPart}$)

Fault Location

Once we have a meronymy, we can use it to inherit features within that hierarchy

For example, a reasoner could infer that a fault in a part is a fault in a whole

- Need a new property for the location of a fault: $\mathrm{hasLocus}$
- Need a new class for faults: Fault

We can then define general types of located faults:

$$\mathrm{FaultInCar} \equiv \mathrm{Fault} \sqcap \exists \mathrm{hasLocus}.\, \mathrm{CarPart}$$
$$\mathrm{FaultInEngine} \equiv \mathrm{Fault} \sqcap \exists \mathrm{hasLocus}.\, \mathrm{EnginePart}$$

Fault Location

Now we can define specific types of located fault:

$$\text{DirtySparkPlug} \sqsubseteq \text{Fault} \sqcap \exists \text{hasLocus}.\, \text{SparkPlug}$$
$$\text{FlatTyre} \sqsubseteq \text{Fault} \sqcap \exists \text{hasLocus}.\, \text{Wheel}$$

The definition of the hierarchy allows a reasoner to infer that:

$$\text{DirtySparkPlug} \sqsubseteq \text{FaultInCar}$$
$$\text{DirtySparkPlug} \sqsubseteq \text{FaultInEngine}$$
$$\text{FlatTyre} \sqsubseteq \text{FaultInCar}$$

But not:

$$\text{FlatTyre} \sqsubseteq \text{FaultInEngine}$$

UNIVERSITY OF Southampton

Further Reading

SWBP Notes

Defining N-ary Relations on the Semantic Web
http://www.w3.org/TR/swbp-n-aryRelations

Representing Specified Values in OWL
http://www.w3.org/TR/swbp-specified-values

Simple part-whole relations in OWL Ontologies
http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/