Knowledge Modelling and Management

Part B (4)

Yun-Heh Chen-Burger

http://www.aiai.ed.ac.uk/~jessicac/project/KMM

Data/Domain Modelling and Knowledge Representation

Knowledge Modelling in CommonKADS

- Contains data and knowledge analysis and structures;
- Support Reasoning task;
- Has three type of knowledge (knowledge category):
  - Domain knowledge;
  - Inference knowledge;
  - Task knowledge.
- Does not consider implementation-related issues (left for design phase):
  - Even when one talks about "rules", we mean natural rules and not the actual coding/shape/forms of rules that may be used in a knowledge system.
- Domain knowledge includes concepts, but it does not includes functions (methods);
- Knowledge may be described using both graphical as well as textual notations.

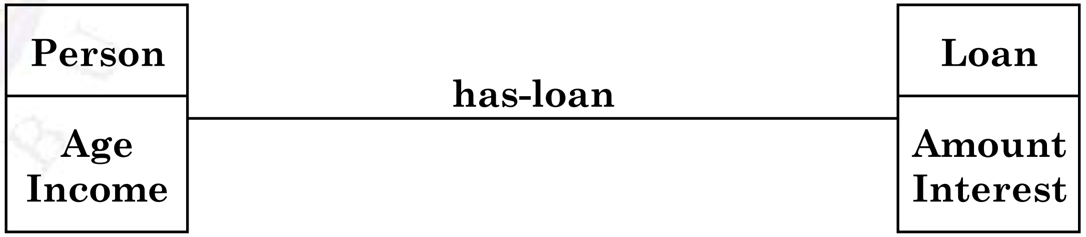
Domain Knowledge

- **Domain schema:**
  - A schematic description of the domain knowledge through a number of type definitions.
  - Its function is similar to a data model (ER), object model (OO), class diagram (UML), or an (un-populated) ontology.
- **Knowledge base:**
  - Contains instances of types specified in a domain schema,
  - Similar to a database,
  - May use different knowledge bases to store different types of knowledge.
- Some characteristics of a knowledge system:
  - Has rules;
  - Support logic based inference:
  - May propose hypothesises and verify them;
  - May hold worlds of different believes at the same time;
  - May derive and store “uncertain” information that may be over-written at a later time;
  - May backtrack to produces different plausible believes of the systems or to seek alternative solutions for a problem;
  - May carry out proofs for a statement;
  - Applications:
    - » Derivative: may derive what “may/must have happened in the past”;
    - » Predictive: may infer what may/must happen next;
    - » Provides decision-making support.

Example: Loan Application

- A piece of data
  - Name of a person
- A piece of information
  - Applicant to a loan
- A piece of knowledge
  - Credit rating and risk assessment for loans
  - Knowledge may be grouped via
    - Personal financial history in terms of repayment
    - Personal income
    - Any other existing loans?
    - Personal spending behaviours: pay by card, cash, overdraft, any credit card debt?
    - Relation to the bank, existing customers? For how long?
    - Different types of loans on offer and their own risk assessment mechanisms

Domain Modelling and Knowledge Representation



Information

John has a loan of $1,750.
Harry has a loan of $2,500.

Facts

Has-loan(john, 1750).
Has-loan(harry, 2500).

Knowledge

A person with a loan should be at least 18 years old.
A person with an income up to £10,000 can get a maximum loan of £2,000.
A person with an income between $10,000 and £20,000 can get a maximum loan of £3000.

Axioms

Rule 1: If.. Then…
Rule 2: If.. Then…
Rule 3: If.. Then…
…
Rule n: If.. Then…

Single flat knowledge base

Rule set of type A
Rule set of type C
…
Rule set of type B
Rule set of type N

Organised multiple rule sets

Example Rules
(Forward Chaining Rules)

- A person with a loan should be at least 18 years old.
- A person with an (annual) income up to £10,000 can get a maximum loan of £2,000.
- A person with an (annual) income between £10,000 and £20,000 can get a maximum loan of £3000.

Qualification process:

qualifyForLoan(Person) $\leftarrow$ age(Person, X) $\wedge$ greaterEqual(X, 18)

Deciding on loan size:

maxLoanSize(Person, 2000) $\leftarrow$ hasIncome(Person, X) $\wedge$ lessEqual(X, 10000) $\wedge$ qualifyForLoan(Person)

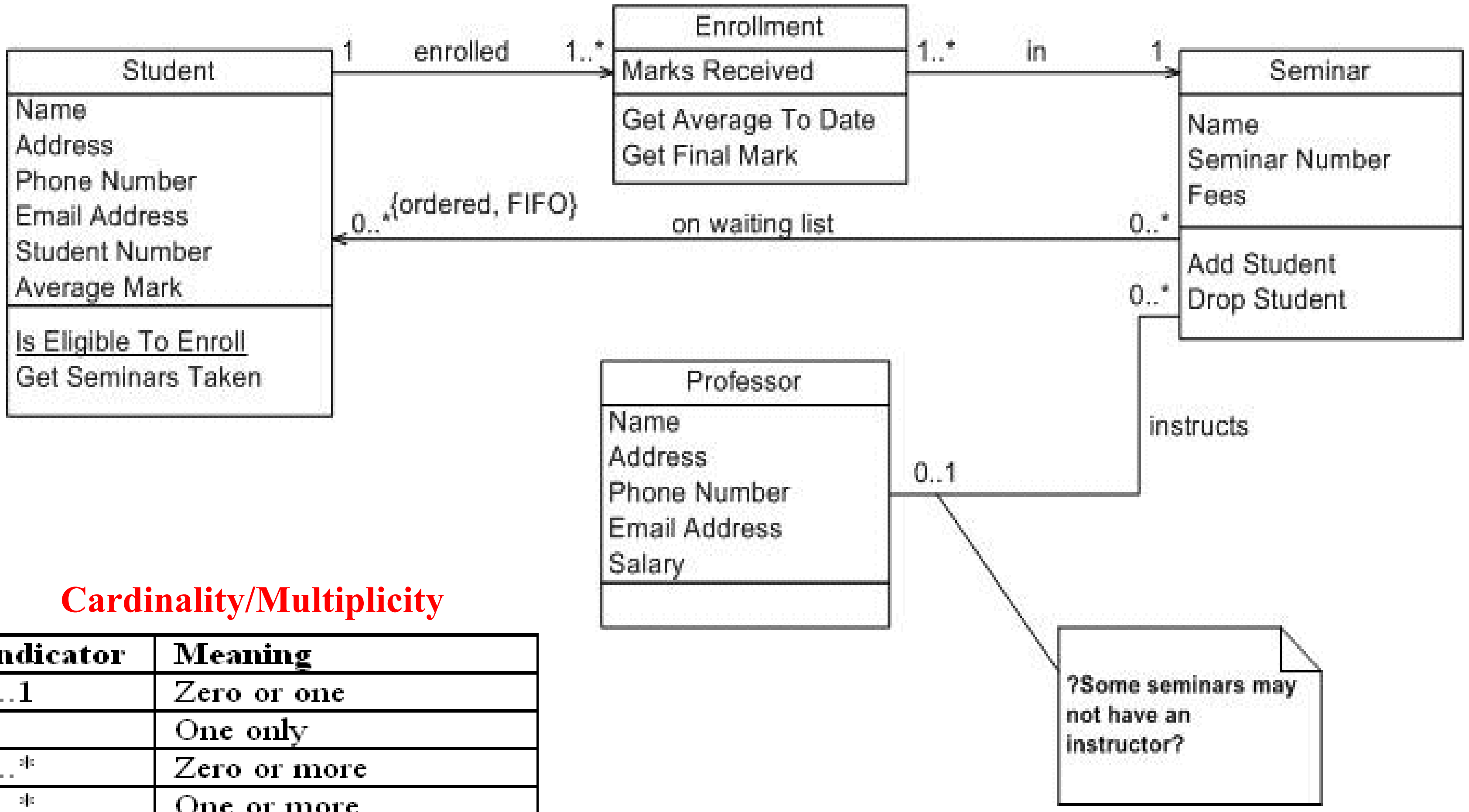
maxLoanSize(Person, 3000) $\leftarrow$ hasIncome(Person, X) $\wedge$ greaterEqual(X, 10000) $\wedge$ lessEqual(X, 20000) $\wedge$ qualifyForLoan(Person)

Specify domain knowledge using UML Class Diagram

- Main modelling primitives:
  - Class;
  - Attribute and value-types (e.g. boolean, real number, integer, natural number, numerical range, text, string, set of symbols, date, universal.);
  - Relationships: association, generalisation, aggregation, composition;
  - Cardinality/multiplicity: cardinality, mini-cardinality, max-cardinality; default 1-1;
  - Axioms.

- Recommended naming style of classes: noun, noun phrase.
- Recall the naming style of processes which may incorporate a class' name.

An UML Class Diagram

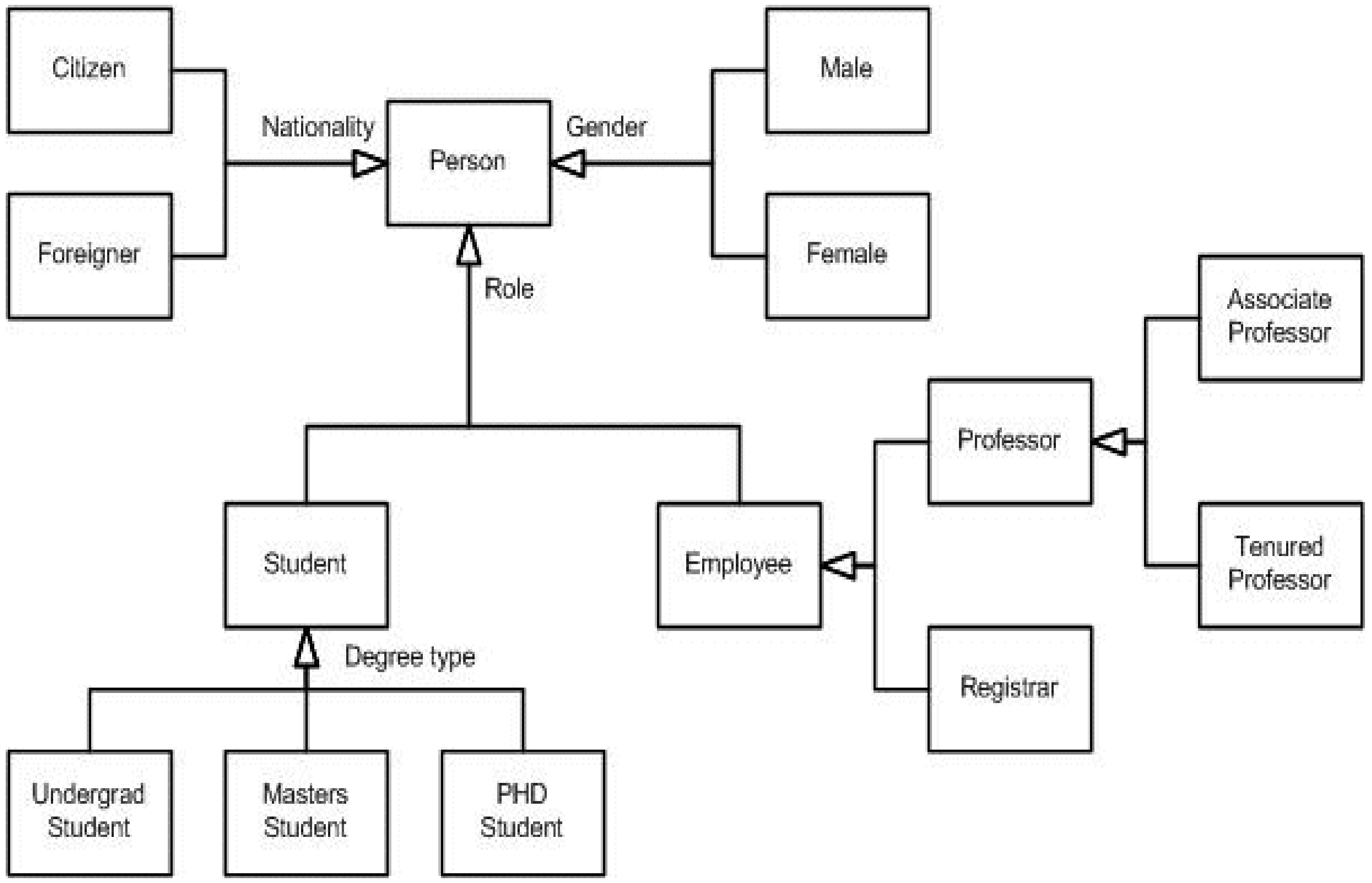


Cardinality/Multiplicity

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only n (where $n > 1$)
0..n	Zero to n (where $n > 0$)
1..n	One to n (where $n > 1$)

Note:
1. Methods as well as properties are included.
2. One relation is represented in a class here.

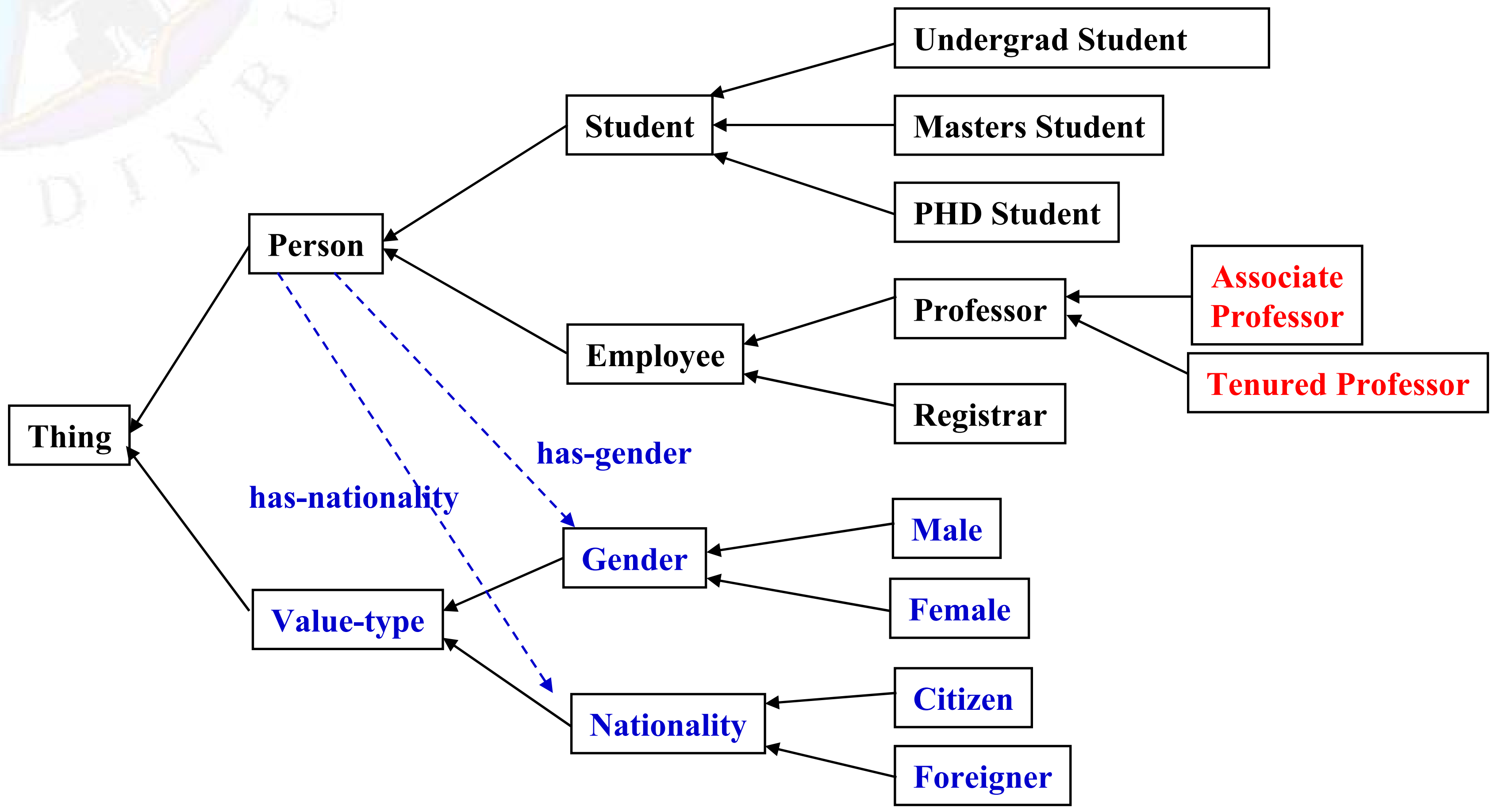
UML Class Diagram used to Express Inheritance



Sub-type and Differentiae

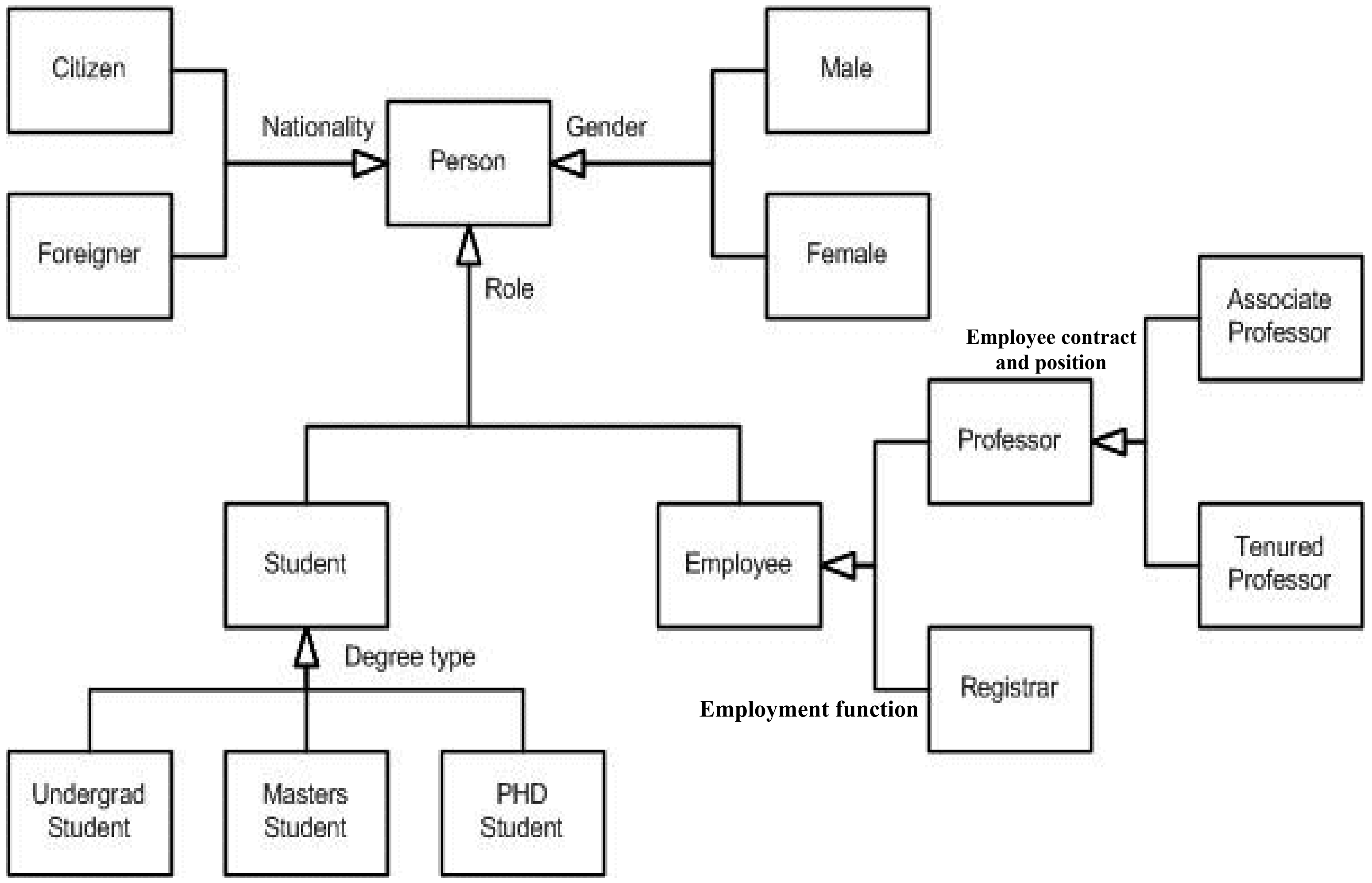
- The properties, features or attributes that are used to distinguish/group a type of instances from other types of instances thus to have a common super-type.
- This term comes from Aristotle's method of defining new types by stating the *genus* or super-type and stating the differentiae that distinguish the new type from its super-type.
- Aristotle's method of definition has become the de facto standard for natural language dictionaries, and it is also widely used for AI knowledge bases and object-oriented programming languages.

Ontology
Demote Classes into Hierarchy

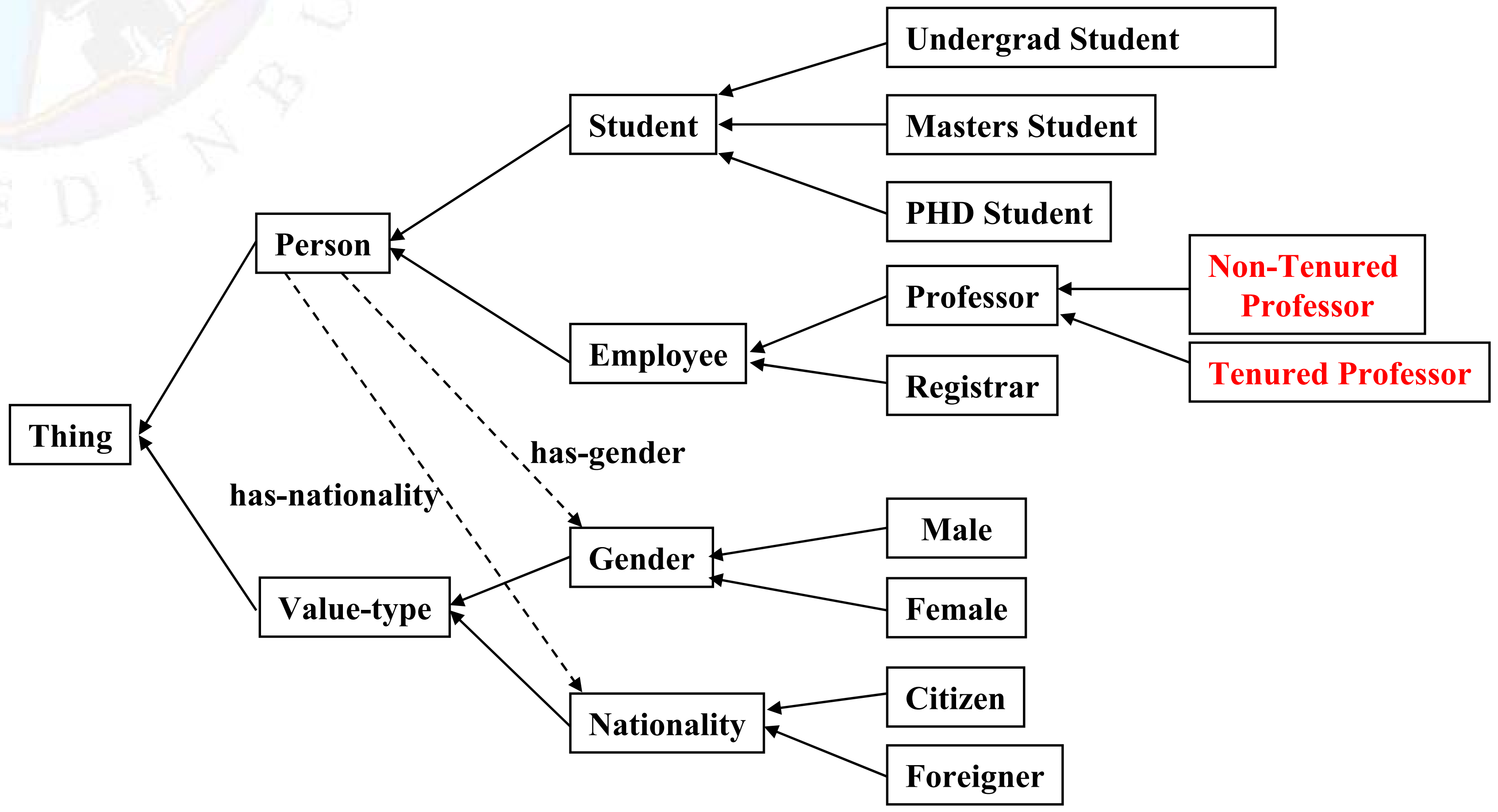


1. Check Ontology hierarchy with application domain, 2. Prioritise Differentiae in the domain.

UML Class Diagram used to Express Inheritance

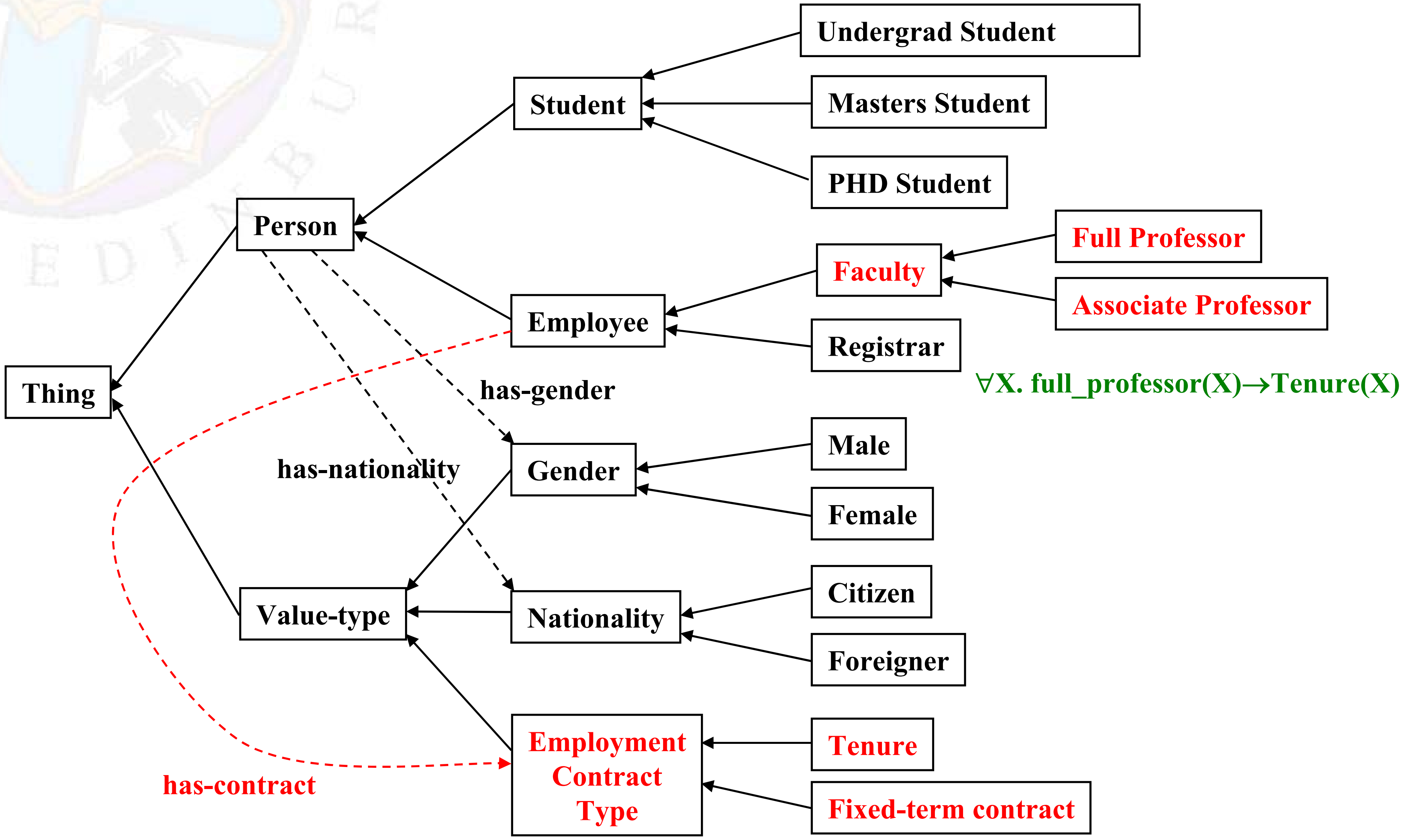


Clarify Concepts in non-disjointed classes



Two differentia have been used and one can be extracted and used alone to provide a clearer division of types. In this case, to represent the contractual status of a professor. The information of the types of Professor is lost.

Specify Contract Properties



Extract the contractual type in a different category while preserving the type information of professors.

Compare UML with Ontology

UML Class Diagram	Ontology
class (without method)	concept; class
generalisation	subclass, subtype
association: (aggregation, composition)	relationship: (part-of, has-part)
cardinality of association	cardinality of relationship
object	instance
attribute	Attribute
method, operation, function	(no exact counter part)
(informal) Rule, (UML object constraint language)	Inference rule/axiom
(informal) Rule, (UML object constraint language)	constraints on domain
(informal) Rule, (UML object constraint language)	constraints on range

Compare UML with OWL

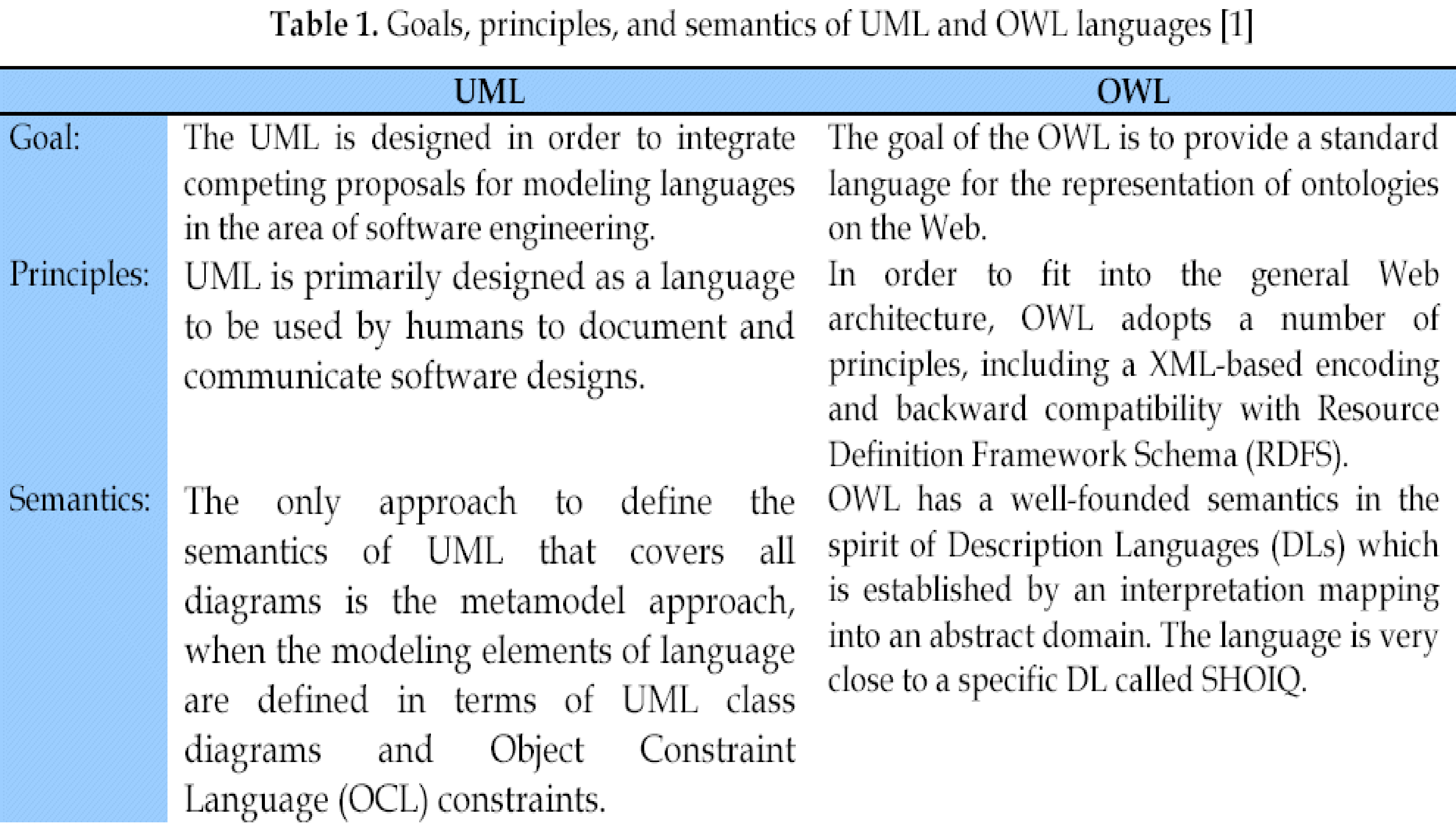


Table 1. Goals, principles, and semantics of UML and OWL languages [1]

	UML	OWL
Goal:	The UML is designed in order to integrate competing proposals for modeling languages in the area of software engineering.	The goal of the OWL is to provide a standard language for the representation of ontologies on the Web.
Principles:	UML is primarily designed as a language to be used by humans to document and communicate software designs.	In order to fit into the general Web architecture, OWL adopts a number of principles, including a XML-based encoding and backward compatibility with Resource Definition Framework Schema (RDFS).
Semantics:	The only approach to define the semantics of UML that covers all diagrams is the metamodel approach, when the modeling elements of language are defined in terms of UML class diagrams and Object Constraint Language (OCL) constraints.	OWL has a well-founded semantics in the spirit of Description Languages (DLs) which is established by an interpretation mapping into an abstract domain. The language is very close to a specific DL called SHOIQ.

Ontology – 1

- The subject of ontology is the study of the categories of things that exist or may exist in some domain.
- The product of such a study, called an ontology, is a catalogue of the types of things that are assumed to exist in a domain of interest D from the perspective of a person who uses a language L for the purpose of talking about D.
- The types in the ontology represent the predicates, word senses, or concept and relation types of the language L when used to discuss topics in the domain D.
- An un-interpreted logic, such as predicate calculus, conceptual graphs, or KIF, is ontologically neutral. It imposes no constraints on the subject matter or the way the subject may be characterized.
- By itself, logic says nothing about anything, but the combination of logic with an ontology provides a language that can express relationships about the entities in the domain of interest.

Ontology - 2

- An informal ontology may be specified by a catalogue of types that are either undefined or defined only by statements in a natural language.
- A formal ontology is specified by a collection of names for concept and relation types organized in a partial ordering by the type-subtype relation. Formal ontologies are further distinguished by the way the subtypes are distinguished from their super-types.
- An axiomatized ontology distinguishes subtypes by axioms and definitions stated in a formal language, such as logic or some computer-oriented notation that can be translated to logic.
- A prototype-based ontology distinguishes subtypes by a comparison with a typical member or prototype for each subtype.
- Large ontologies often use a mixture of definitional methods:
  - formal axioms and definitions are used for the terms in mathematics, physics, and engineering; and
  - prototypes are used for plants, animals, and common household items.

Source: John Sowa [12]

Ontology - 3

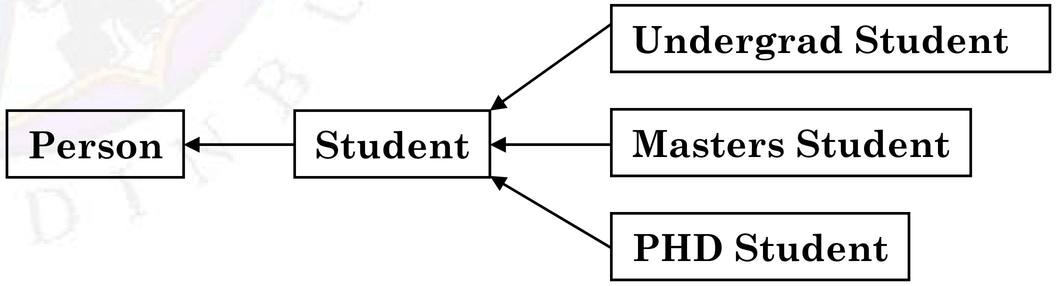
- In the context of knowledge sharing, I use the term ontology to mean a *specification of a conceptualization*. That is, an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents. This definition is consistent with the usage of ontology as set-of-concept-definitions, but more general. And it is certainly a different sense of the word than its use in philosophy.
- What is important is what an ontology is *for*. My colleagues and I have been designing ontologies for the purpose of enabling knowledge sharing and reuse. In that context, an ontology is a specification used for making ontological commitments. The formal definition of ontological commitment is given below. For pragmatic reasons, we choose to write an ontology as a set of definitions of formal vocabulary. Although this isn't the only way to specify a conceptualization, it has some nice properties for knowledge sharing among AI software (e.g., semantics independent of reader and context). Practically, an ontological commitment is an agreement to use a vocabulary (i.e., ask queries and make assertions) in a way that is consistent (but not complete) with respect to the theory specified by an ontology. We build agents that commit to ontologies. We design ontologies so we can share knowledge with and among these agents.

UML Class Diagram

- Object-Oriented approach;
- Used as a conceptual modelling method – not implementation dependent;
- Typically used at a software system design phrase that extends/adapts the conceptual model;
- Typically not (mainly) used to describe the categories of things;
- Typically not used to provide definitions for terminologies.

Representing a Domain Model

Representing a Domain Model



Person
Full Name: String Address: String DoB: Date

	Address
Domain	Person
Range	String

Representing classes:
class(person).
class(student).
class(undergrad_student).

Representing (class) relationships:
subClassOf(student, person).
subClassOf(undergrad_student, student).
subClassOf(masters_student, student).

Defining instance attributes:
domainOf(address, person).
rangeOf(address, string).
rangeOf(DoB, date).
rangeOf(color, {blue, red}).

Representing instances:
instanceOf(john, masters_student).
instanceOf(mary, phd_student).

Representing instance attribute values:
address(john, 'Edinburgh').
address(mary, 'Glasgow').

Knowledge Representation Convention

- **Representing a class:**
  - **class(Class).** **(alternatively, use type(Class).)**
- **Relationships between classes:**
  - **Relation_name(Class1, Class2).**
- **Instances:**
  - **instanceOf(Instance, Class).**
- **Defining the property for instances:**
  - **domainOf(Property_name, Domain_type).**
  - **rangeOf(Property_name, Range_type).**
- **Property values of an instance:**
  - **Property_name(Instance, Value).**
- **Axiom: P → Q**
  - **Q :- P.**

A Well-Formed Formulae in FOPL - 1

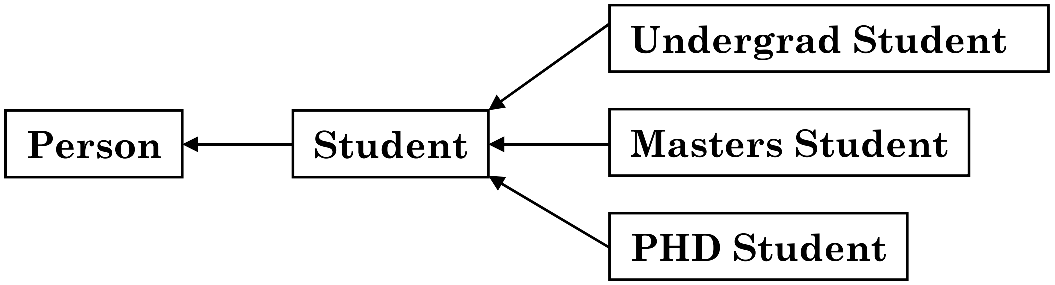
1. A constant can be any number or any unbroken sequence of lower-cased symbols.
2. A variable is any unbroken sequence of symbols beginning with an upper case letter.
3. A predicate is a term consisting of a functor (predicate name), and an ordered set of 0 or more arguments. $F(A_1,\ldots A_n)$.
4. Predicate name must be a constant.
5. An argument (of a predicate) may either be a constant or variable.

A Well-Formed Formulae in FOPL - 2

6. If P and Q are formulae, then the following are also formulae:
   - $\neg P$
   - $\neg Q$
   - $P \wedge Q$
   - $P \vee Q$
   - $P \rightarrow Q$
   - $P \leftrightarrow Q$
7. Only expressions using rules 1 to 6 are formulae.
8. If P is a well-formed term, then $\forall X.P$ and $\exists X.P$ are terms quantified over X. Any variables not quantified using either $\forall$ or $\exists$ are free variables in P.
9. A sentence does not contain free variables.

Representing Logic Sentences - 1

- ¬ P (negation)
  - \+ P.
- P ∧ Q (conjunction)
  - P, Q.
- P ∨ Q (disjunction)
  - P ; Q.
- P → Q (implication)
  - Q :- P.
- P ↔ Q (double implication)
  - R :- P, Q.
  - P :- Q.
  - Q :- P.



Class Relation: Relation(Class1, Class2)

subClassOf(student, person).

If an individual is a member of a particular class, then it is automatically an instance of its superclass:

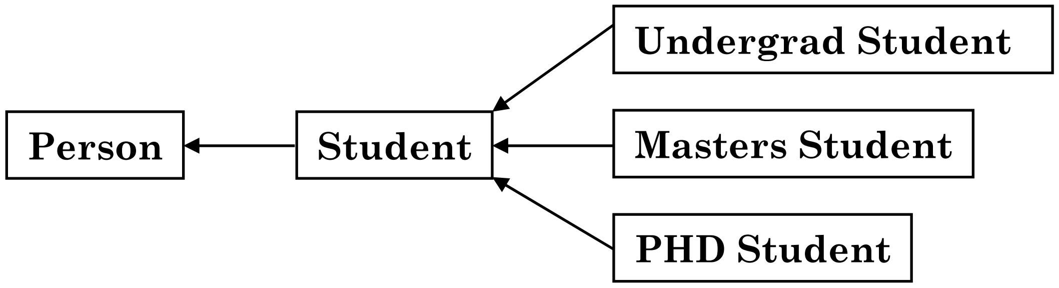
$\forall$X. instanceOf(X, student)
→ instanceOf(X, person)

Prolog sentence:

```
instanceOf(X, person) :-
        instanceOf(X, student).
```

Representing Logic Sentences - 2

- ¬ P
  - \+ P.
- P ∧ Q
  - P, Q.
- P ∨ Q
  - P ; Q.
- P → Q
  - Q :- P.
- P ↔ Q
  - R :- P, Q.
  - Q :- P.
  - P :- Q.



Class Properties: Property(Class1, Class2)

disjointness assertions:

```
disjointWith(undergrad_student, masters_student).
disjointWith(masters_student, phd_student).
disjointWith(undergrad_student, phd_student).
```

It states that an individual that is a member of one class cannot simultaneously be an instance of a specified other class. Therefore a violation rule in Prolog:

```
violation(disjoint, Instance, Class1, Class2) :-
    disjointWith(Class1, Class2),
    instanceOf(Instance, Class1),
    instanceOf(Instance, Class2).
```

Deriving interesting properties using subClassOf Relation- 1

Knowledge Base:

```
subClassOf(undergrad_student, student).
subClassOf(masters_student, student).
subClassOf(phd_student, student).
subClassOf(registrar, employee).
subClassOf(professor, employee).
```

Query (1): what are the different types of students?

```
all_student_types(X) :- subClassOf(X, student).
```

```
| ?- [student].                  ← Load the program
| ?- all_student_types(X).       ← send the query
X = undergrad_student ? ;        ← answers returned
X = masters_student ? ;
X = phd_student ? ;
No
```

Deriving interesting properties - 2

- As the subClassOf relation often can be interpreted as a “is-a” relation, we can use this to answer the below query:

Query (2): what (type) is X ?

```
what_is(X, Type) :- subClassOf(X, Type).

| ?- what_is(registrar, Type).
Type = employee ? ;
No.
```

Deriving interesting properties - 3

Query (3): What are similar to X, i.e. in the same category of X ?

```
same_category(X, Y)  : -  subClassOf(X,  Category),
                            subClassOf(Y, Category),
                          \+ X = Y.
```

```
 | ?- same_category(registrar, Y).
Y = professor ? ;
no.
```

Combined queries of (2) and (3)… What is a registrar and who are similar to registrar?

```
| ?- what_is(registrar, Type), same_category(registrar, Others).
Type = employee,
Others = professor ? ;
No.
```

Summary: Representing a Conceptual Model in FOPL

- **Identify what are the information that you are trying to represent. Do you need to differentiate characteristics and use them to divide the domain in the different models and thus your representations?**
- **Identify what are the classes, instances, properties, relationships, axioms, and typing information in your model. Decide how you want to represent them.**
- **Identify the desirable properties that you wish to infer from your model. You may need to go back to the representational issues on this.**
- **Decide how you are going to derive those properties.**
- **Decide how you may want to verify certain properties, e.g. in the model or when receives an enquiry about information stored in the model.**

Summary: Concepts and Terminologies

- Concept/Class:
  - A concept describes a set of objects or instances which occur in the application domain and which share similar characteristics. [p. 92, 1]
  - Also sometimes refers to as:
    - » Type, Sort, Entity, Unary relation;
    - » Related to/originate from “set”.
- Relationship/association:
  - Examples: sub-class, sub-type, is-a, instance-of, inheritance.
- Object: also called instance, occurrence, individual;
- Process: also called activity, task, procedure.
- Other concepts: disjoint decomposition, completeness (exhaustive decomposition), partition (a non-overlapping division of a set).

Main Reference

- [1] (Chapter 5, 14, 13-13.2.3) in Knowledge Engineering and Management: The CommonKADS Methodology. Guus Schreiber, Robert de Hoog, Hans Akkermans, Anjo Anjewierden, Nigel Shadbolt, Walter Van de Velde.

Additional Reference

- [14a] Prolog program Library: student.pl: http://www.aiai.ed.ac.uk/~jessicac/project/prolog/.
- [1b] For understanding Prolog and Clausal Logic: Peter Flach. Simply Logical: Intelligent Reasoning by Example. Wiley Professional Computing. 1994.
- [2b] For understanding Prolog: Ian Bratko, Prolog Programming for Artificial Intelligence (2nd edition), Addison Wesley, 1986.

Other Reference (not examable)

- [11a] OWL Overview: http://www.w3.org/TR/owl-features/.
- [12a] RDF: http://www.w3.org/RDF/.
- [13a] RDFS: http://www.w3.org/TR/rdf-schema/.
- [15a] OWL Reference: http://www.w3.org/TR/2004/REC-owl-guide-20040210/#DisjointClasses.
- [16a] Yun-Heh Chen-Burger and Dave Robertson. Automating Business Modelling. Book Series of Advanced Information and Knowledge Processing, Springer Ver-Lag, December 2004. http://www.springeronline.com/sgw/cda/frontpage/0,11855,5-40356-72-34527494-0,00.html.
- [17a] Tom Gruber: What is an ontology?. http://www-ksl.stanford.edu/kst/what-is-an-ontology.html.

Additional Information

What is UML OCL (Object Constraint Language)

- OCL is a small, text-based formal language for object modelling, using only ASCII characters.
- It can be used to describe pre- and post-conditions.
- However, it is difficult to generate code automatically from OCL because
  - It is declarative and not imperative – e.g. it tells you that balance equals the total balance of an account, but it does not tell you to add up or subtract transactions;
  - It is a constraint language, and not an action language – it is much better to generate codes form an action language, and not from a constraint language.

Why OCL ?

- OCL is suitable to be used to generate (automatic) verification and validation rules – which can be used for constraint checking at the model layer (i.e. checking class diagrams)
- Roundtrip support for assertions.
- Model Animation ("running" scenarios in the tool).
- Tool: Poseidon from Gentleware, or the Boldsoft tools (now part of Borland) They both support OCl, including code generation; Rational Rose with EmPowerTecs add-in.

Partial Order and Total Order

A *partial order* (often simply referred to as an *order* or *ordering*) is a relation $\leq \subset A \times A$ that satisfies the following three properties:

1. Reflexivity: $a \leq a$ for all $a \in A$
2. Antisymmetry: If $a \leq b$ and $b \leq a$ for any $a, b \in A$, then $a = b$
3. Transitivity: If $a \leq b$ and $b \leq c$ for any $a, b, c \in A$, then $a \leq c$

A *total order* is a partial order that satisfies a fourth property known as *comparability*:

- Comparability: For any $a, b \in A$, either $a \leq b$ or $b \leq a$.

A set and a partial order on that set define a Poset.

Poset

A *poset* is a partially ordered set, that is, a poset is a pair $(P, \leq)$ where $\leq$ is a partial order relation on P.

A few examples:

- $(\mathbb{Z}, \leq)$ where $\leq$ is the common ``less than or equal'' relation on the integers.
- $(\mathcal{P}(X), \subseteq)$. $\mathcal{P}(X)$ is the power set of X and the relation $\subseteq$ given by the common inclusion of sets.

In a partial order, not any two elements need to be comparable. As example, consider $X = \{a, b, c\}$ and the poset on its power set given by the inclusion. Here, $\{a\} \subseteq \{a, c\}$ but the two subsets $\{a, b\}$ and $\{a, c\}$ are not comparable. (Neither $\{a, b\} \subseteq \{a, c\}$ nor $\{a, c\} \subseteq \{a, b\}$).