Semantic Web

Ontology Pattern

Gerd Gröner, Matthias Thimm

{groener,thimm}@uni-koblenz.de

Institute for Web Science and Technologies (WeST)
University of Koblenz-Landau

July 18, 2013

Outline

Outline

Learning Goals

- Qualify in the engineering and design of an ontology.
- Reflect on different aspects when creating an ontology.

Modeling in OWL

- OWL gives us logical language constructs, but does not give us any guidelines on how to use them in order to solve our tasks.
  - E.g., modeling something as an individual, a class, or an object property can be quite arbitrary
- ... OWL is not enough for building a good ontology and we cannot ask all web users either to learn logic, or to study ontology design
- Reusable solutions are described as *Ontology Design Patterns*, which help reducing arbitrariness without asking for sophisticated skills ...
- ... provided that tools are built for any user.

Modeling in OWL

- OWL gives us logical language constructs, but does not give us any guidelines on how to use them in order to solve our tasks.
  - E.g., modeling something as an individual, a class, or an object property can be quite arbitrary
- ... OWL is not enough for building a good ontology and we cannot ask all web users either to learn logic, or to study ontology design
- Reusable solutions are described as *Ontology Design Patterns*, which help reducing arbitrariness without asking for sophisticated skills ...
- ... provided that tools are built for any user.

Modeling in OWL

- OWL gives us logical language constructs, but does not give us any guidelines on how to use them in order to solve our tasks.
  - E.g., modeling something as an individual, a class, or an object property can be quite arbitrary
- ... OWL is not enough for building a good ontology and we cannot ask all web users either to learn logic, or to study ontology design
- Reusable solutions are described as *Ontology Design Patterns*, which help reducing arbitrariness without asking for sophisticated skills ...
- ... provided that tools are built for any user.

Ontology Design Pattern (OP)

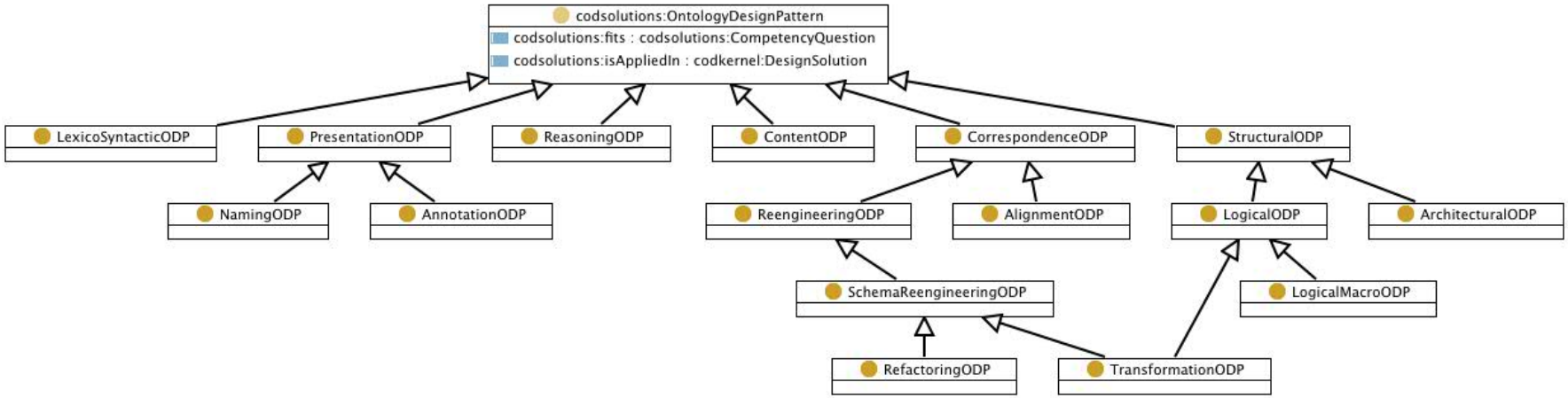
Definition

An ontology design pattern is a reusable solution to a recurrent modeling problem.

Pattern-based Design

- Pattern-based ontology design is the activity of searching, selecting, and composing different patterns
  - Logical,
  - Reasoning,
  - Architectural,
  - Naming,
  - Correspondence,
  - Reengineering,
  - Content
- Common framework to understand modeling choices (the "solution space") with respect to task- and domain-oriented requirements (the "problem space")
- `http://www.ontologydesignpatterns.org`

Types of OPs



There are

- Ontology Design Patterns
- not really Ontology Patterns (indifferent)
- Ontology Design Anti-Patterns (AntiOPs)

Namping OPs

- Naming OPs are conventions on how to create names for namespaces, files, and ontology elements in general (classes, properties, etc.).
- Naming OPs are good practices that boost ontology readability and understanding by humans, by supporting homogeneity in naming procedures.
- Example:
  - Class names – singular, capital letter
  - Human readable names – not only numeric values
  - Use of suffixes or prefixes

Annotation OPs

- Annotation OPs provide annotation properties or annotation property schemas that are meant to improve the understandability of ontologies and their elements

- Example:
  - Use of RDF Schema labels and comments (crucial for manual selection and evaluation)
  - Each class and property should be annotated with meaningful labels (also: different language)
  - Each ontology and ontology element should be annotated with the rationale they are based on using *rdfs:comment*

Correspondence OPs

- Correspondence OPs include
  - Reengineering OP,
  - Mapping OP.

- OPs provide designers Reengineering with solutions to the problem of transforming a conceptual model, which can even be a non-ontological resource, into a new ontology.

- Mapping OPs are patterns for creating semantic associations between two existing ontologies.

Reengineering OPs

- Reengineering OPs are transformation rules applied in order to create a new ontology (target model) starting from elements of a source model
- The target model is an ontology, while the source model can be either an ontology, or a non-ontological resource e.g., a thesaurus concept, a data model pattern, a UML model, a linguistic structure, etc.
- Two types:
  - **Schema reengineering OPs** are rules for transforming a non-OWL DL metamodel into an OWL DL ontology
  - **Refactoring OPs** provide designers with rules for transforming, i.e., "refactoring", an existing OWL DL "source" ontology into a new OWL DL "target" ontology
    - e.g., a guideline to reengineer a piece of an OWL ontology in presence of a requirement change, as when moving from individuals to classes, or from object properties to classes.

Reengineering OPs

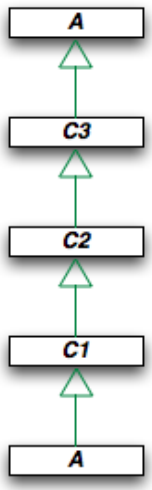
- Reengineering OPs are transformation rules applied in order to create a new ontology (target model) starting from elements of a source model
- The target model is an ontology, while the source model can be either an ontology, or a non-ontological resource e.g., a thesaurus concept, a data model pattern, a UML model, a linguistic structure, etc.
- Two types:
  - **Schema reengineering OPs** are rules for transforming a non-OWL DL metamodel into an OWL DL ontology
  - **Refactoring OPs** provide designers with rules for transforming, i.e., "refactoring", an existing OWL DL "source" ontology into a new OWL DL "target" ontology
    - e.g., a guideline to reengineer a piece of an OWL ontology in presence of a requirement change, as when moving from individuals to classes, or from object properties to classes.

Refactoring Pattern – Example
Cyclic SubClassOf

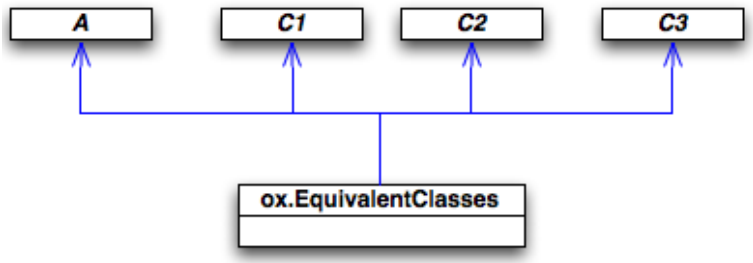
- *Problem:* An explicitly defined cyclic chain of subClassOf axioms. $\rightarrow$ This is sometimes difficult to understand
- *Ontology description:* There are the following axioms:

$$\begin{aligned} A &\sqsubseteq C_1 \\ C_1 &\sqsubseteq C_2 \\ &\cdots \\ C_n &\sqsubseteq A \end{aligned}$$

- *Solution (Refactoring):* Replace the cyclic subClassOf chain by an equivalent class axiom



equivalent classes:



Structural OPs

- Structural OPs affect the shape / structure of an ontology.

- Two kinds of structural OPs:
  - Architectural OPs
  - Logical OPs

Architectural OPs

- Architectural OPs affect the overall shape of the ontology: their aim is to constrain 'how the ontology should look like'
- Architectural OPs emerged as design choices motivated by specific needs e.g., computational complexity constraints.
- They are useful as reference documentation for those initially approaching the design of an ontology

Architectural OPs (2)

- Architectural OPs can be of two types: internal APs and external APs
  - **Internal APs** are defined in terms of collections of Logical OPs that have to be exclusively employed when designing an ontology
    - e.g., an OWL species, or the varieties of description logics: `http://www.cs.man.ac.uk/~ezolin/dl/`
  - **External APs** are defined in terms of meta-level constructs
    - e.g., the modular architecture consists of an ontology network, where the involved ontologies play the role of modules. The modules are connected by the import operation.

Logical OPs

- A Logical OP is a *formal expression*, whose only parts are expressions from a logical vocabulary e.g., OWL DL, that solves a problem of expressivity
- Logical OPs are independent from a specific domain of interest – they are content-independent
- Logical OPs depend on the expressivity of the logical formalism that is used for representation
  - They help to solve design problems where the primitives of the representation language do not directly support certain logical constructs
- They can be of two types: **logical macros**, and **transformation patterns**

Logical Macros

- Logical macros provide a shortcut to model a recurrent intuitive logical expression
- Example:
  - the macro: $\forall R.C$
  - colloquially means "every R must be a C"
  - formally: $\exists R.T \sqcap \forall R.C$

  $\rightarrow$ this would be expressed in OWL as the combination of an owl:allValuesFrom restriction with an owl:someValuesFrom restriction.

Transformation Patterns

- Transformation patterns translate a logical expression from one logical language into another. The semantics of the first is approximated, in order to find a trade-off between requirements and expressivity.
- Example: N-ary relations that cannot be directly expressed in OWL.
  $\rightarrow$ An approximation of an N-ary relation in OWL is to create a new class representing the relation and indicate the arguments through properties.

Content Ontology Design Pattern (CP)

- CPs encode conceptual, rather than logical design patterns.
  - Logical OPs solve design problems independently of a particular conceptualization
  - CPs are patterns for solving design problems for the domain classes and properties that populate an ontology, therefore they address content problems

Content Ontology Design Pattern (CP) (2)

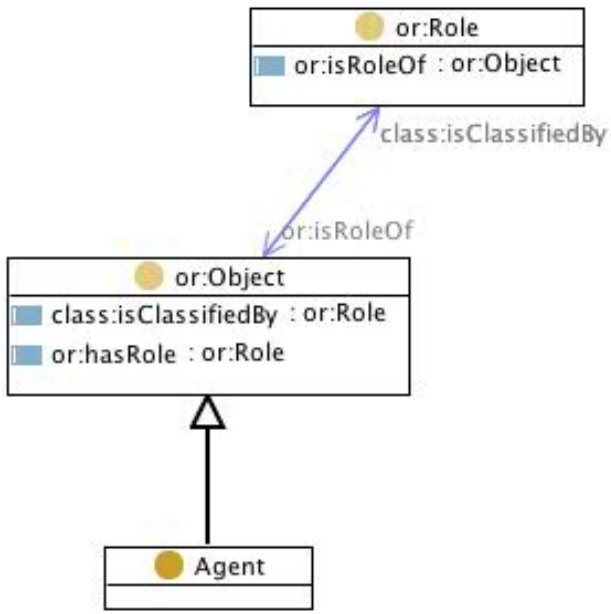
- CPs are instantiations of Logical OPs (or of compositions of Logical OPs)
  - they have an explicit non-logical vocabulary for a specific domain of interest – **they are content-dependent**

- Modeling problems solved by CPs have two components: *domain* and *requirements*
  - one domain – many requirements
  - same requirement – different domains
  - typical way of capturing requirements is by means of competency questions

Characteristics of CPs

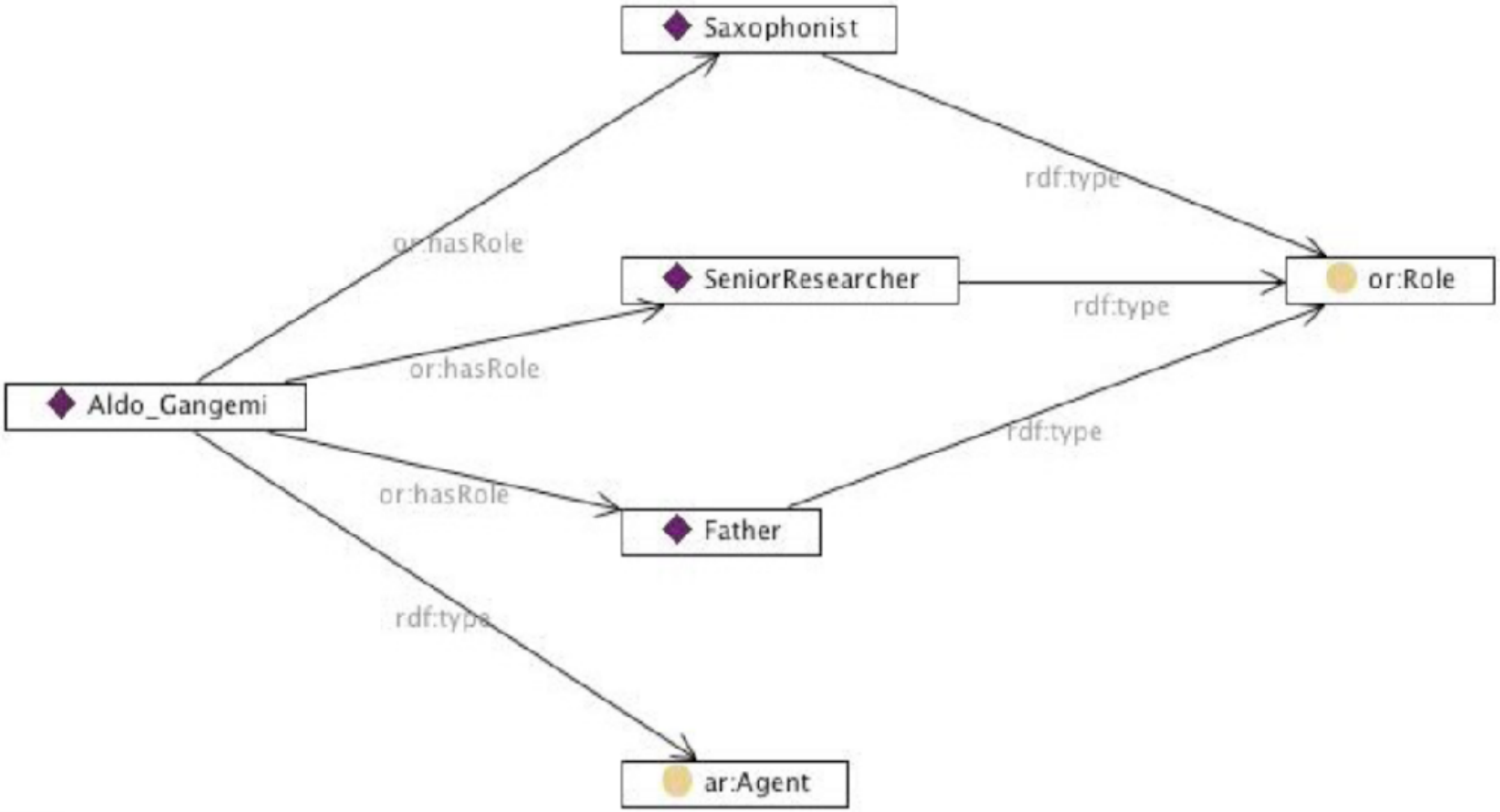
- (Small) ontology morphing
  - "being a part of something at some time"
- Downward subsumption of at least one element
  - "being *a component* of a system at some time"
- Mostly graphs of classes and properties that are self-connected through axioms (subClassOf, equivalentClass, domain, range, disjointFrom)
- Usually there is an underlying n-ary relation (sometimes polymorphic)
- For instance: $\mathit{Consumer} \sqcap \exists \mathit{connect}.\mathit{Producer} \sqsubseteq \mathit{Supplied}$

Example: Agent role



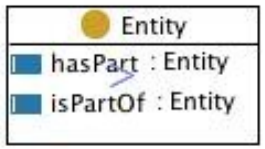
- **Intent / modeling problem:** To represent agents and the roles they play.
- Competency question: "Which agent does play this role?"
- **Consequence / solution:** This CP allows designers to make assertions on roles played by agents without involving the agents that play that roles, and vice versa. It does not allow to express temporariness of roles.

Example: Agent role – instantiation



Example: PartOf pattern

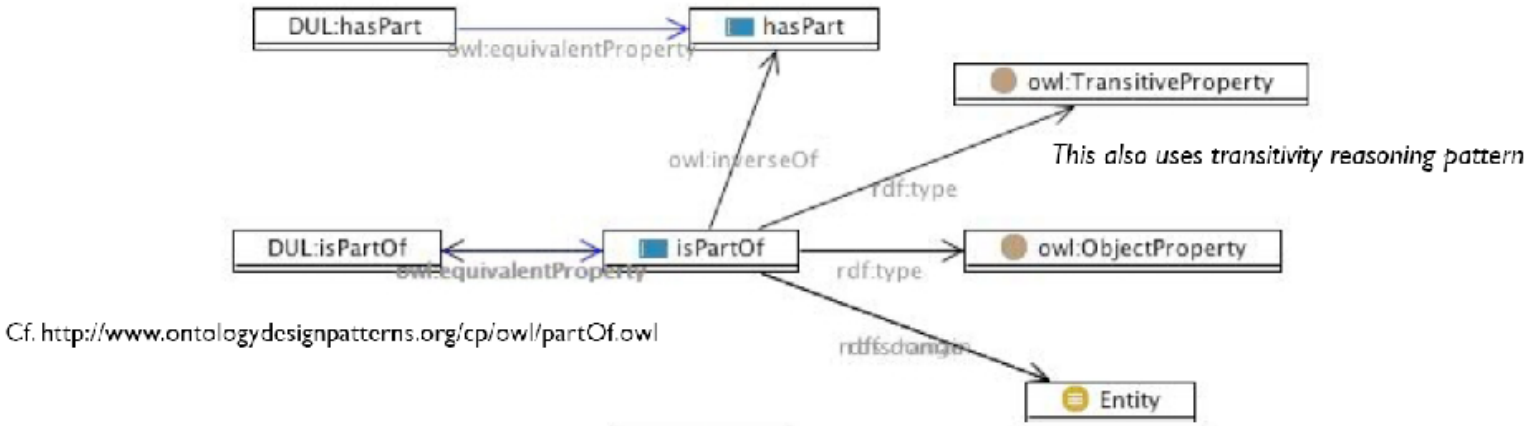
- Pattern type: Content pattern
- **Intent / modeling problem:** To represents entities and their parts.
- Competency question: "What is this entity part of?" and "What are the parts of this entity?"
- **Consequence / solution:** this content OP allows designers to represent entities and their parts i.e., part-whole relations.



→ this pattern is extracted from DUL (DOLCE+DnS Ultralite)

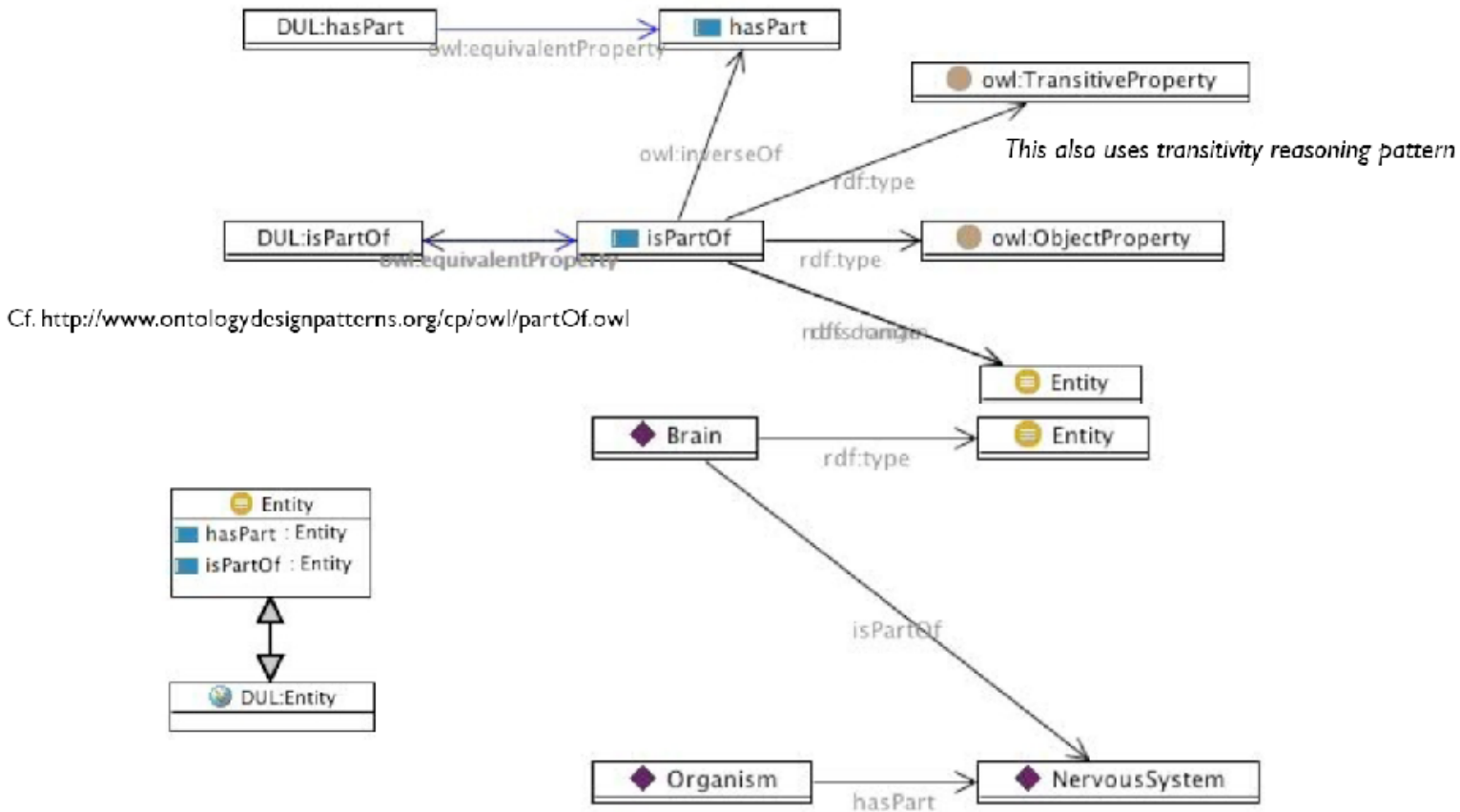
→ core ontology, upper ontology (for purpose of re-use)

Example: PartOf pattern – extracted from DUL



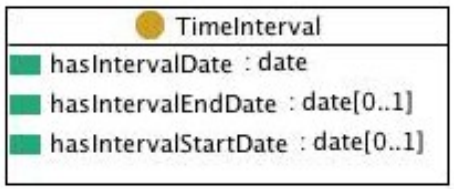
Example: PartOf pattern – usage

use this pattern, e.g., in a medical domain:



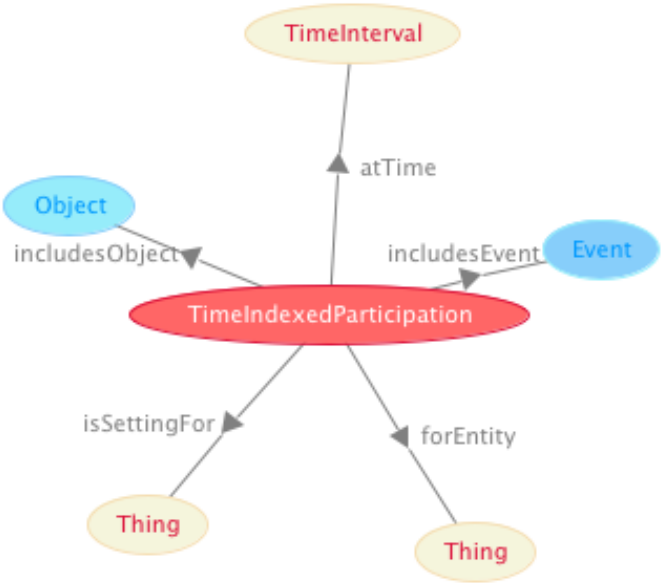
Example: Time interval

- Pattern type: Content pattern
- **Intent / modeling problem:** To represent time intervals.
- Competency question: "What is the end time of this interval?", "What is the starting time of this interval?" and "What is the date of this time interval?"
- **Consequence / solution:** The dates of the time interval are not part of the domain of discourse, they are datatype values.
  - Note: If there is the need of reasoning about dates this Content OP should be used in composition with the *region Content OP*. (This pattern is a basic one, which allows to talk about attributes/parameters/dimensions, while still referring to their values-)

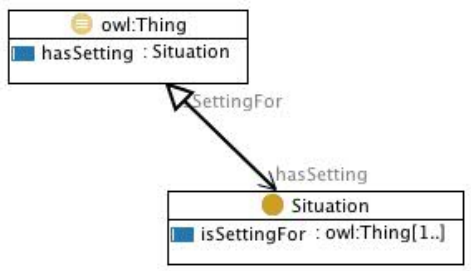


Example: Time-indexed participation

- Pattern type: Content pattern
- **Intent / modeling problem:** To represent participants in events at some time. To represent participants in parts of events.
- Competency question: "When something participated in some event?" and " At what time an event had some participant?"
- **Consequence / solution:** This pattern uses the situation pattern to add temporal information to participation of objects into events.



- The situation pattern is a more general content OP:

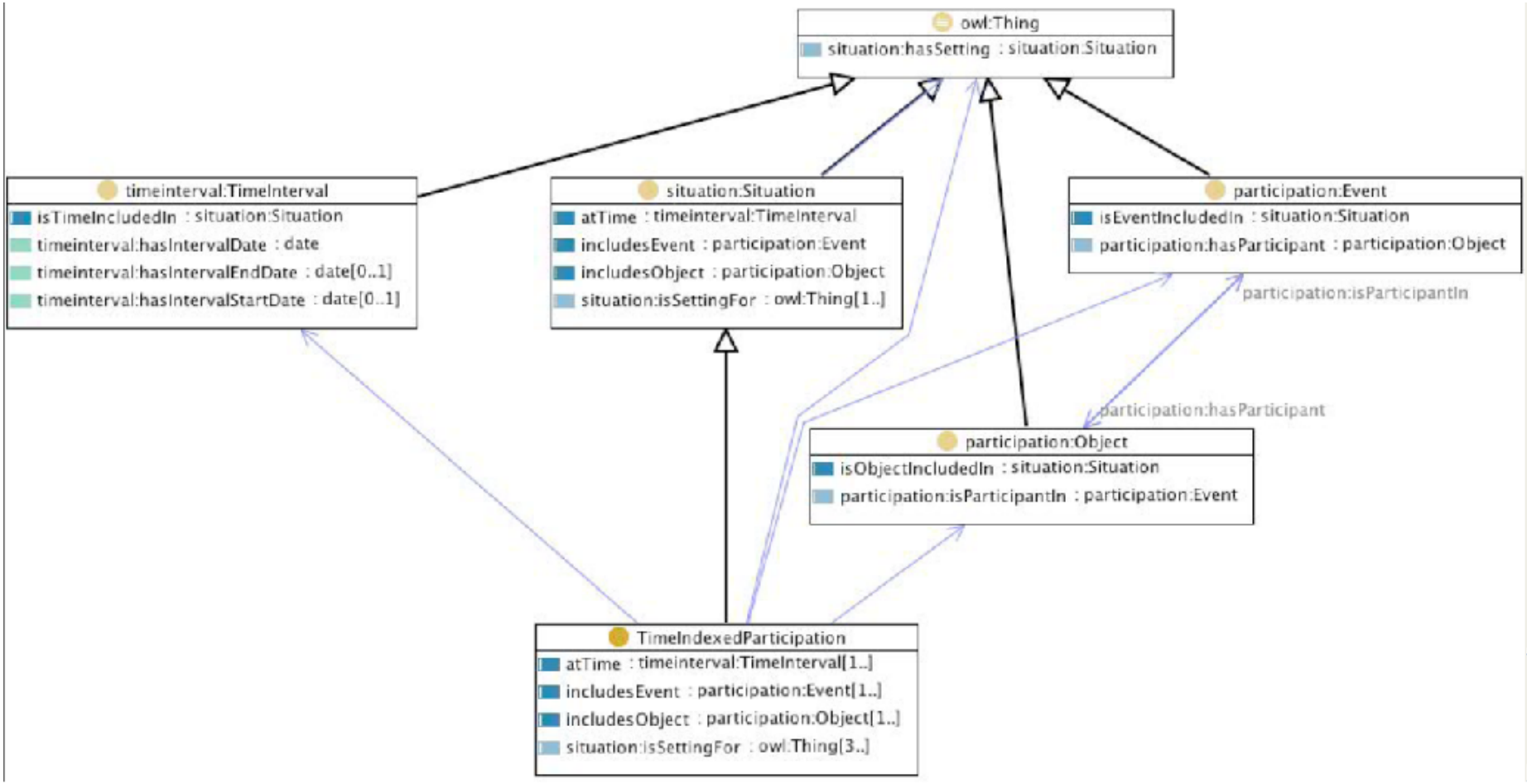


Reuse of pattern

- Matching pattern to your situation / ontology
  - Precise or redundant matching
  - Broader or narrower matching
  - Partial matching

- Content Pattern creation and reuse for specific ontology relies on a set of operations:
  - import
  - specialization
  - generalization (converse of specialization)
  - composition
  - expansion
  - clone

Example reuse of patterns: composition



Summary Content Pattern

- Content ontology design patterns (CPs) come from the experience of ontology engineers in modeling foundational, core, or domain ontologies
- Engineering principle (engineering way):
  - Specialization, generalization, expansion and composition of other CPs
  - Extraction from other (reference) ontologies

  $\rightarrow$ sure, the same holds for (re-)using of CPs

Summary

- Patterns are used in many areas as "templates" or abstract descriptions encoding best practices of some field.
  - inspiration taken from the architecture field and Christopher Alexander
  - Software patterns are probably most well known as Design Patterns, as in the GoF book from 1995.

  $\rightarrow$ carrying this principle to ontology / knowledge engineering
- Remember: "An ontology design pattern is a reusable successful solution to a recurrent modeling problem."
- Most important pattern:
  - Structural OPs: Logical OPs and Architectural OPs
  - Correspondence OPs: Reengineering OPs and Alignment OPs
  - Content OPs (CPs)
  - Reasoning OPs
  - Presentation OPs: Naming OPs and Annotation OPs