Pattern-based design, part I

Aldo Gangemi
Valentina Presutti

Semantic Technology Lab
ISTC-CNR, Rome
From “raw” data to patterns

- Moving from “raw” knowledge resources to networked ontologies require: [cf. C-ODO]
  - Ontology requirement analysis (domain(s), task(s), and sustainability constraints for ontologies to be built/managed)
  - Tool/resource requirement analysis (functionalities to be covered by tools, and competences needed)
  - Project planning (deciding on knowledge resources, economic resources, team composition and responsibilities, data copyright management, tools)
  - Workflow decision making (specially for reengineering and argumentation)
  - Rationale elicitation (“critiquing” the reengineered data)
  - Providing solutions (e.g. based on design patterns, or conveying new ones)
- Not one, “best” methodology
  - A project can start spontaneously to solve a rationale elicitation problem, can be planned in order to reengineer knowledge resources, or to reuse existing ontologies or patterns, etc.
  - A project can be started either with or without requirement analyses
  - Even the solutions can consist only of a “bulk” reengineering process, without explicit patterns
  - eXtreme Design?
- In this tutorial, I concentrate on solutions based on ontology design patterns
OPs and patterns in other disciplines 1/3

• One might expect OPs to be easily comparable to software engineering design patterns.

• The same analogy has been done with architecture, linguistics, and other disciplines.

• Ontology engineering and software engineering show many similarities from the pragmatic viewpoint, but they are quite different from the theoretical viewpoint.
We use comparisons between ontology engineering and software engineering for clarifying concepts and intuitions behind the definitions.

We do not take theoretical aspects of OP as dependent on those of software engineering (or other fields).
OPs and patterns in other disciplines 3/3

- Our concept of “pattern” is associable with the wider “good/best practice” of software engineering.

- It includes a wider range of solution types. For example:
  - naming conventions in software engineering are considered good practices, they are not design patterns.
  - In ontology engineering “naming” is an important design activity (it can have a strong impact on the usage of the ontology e.g., for selection, mapping, etc.).
  - We classify ontology naming conventions as OPs.

- We distinguish the different types of OPs by grouping them into six families.

- Each family addresses different kinds of problems, and can be represented with different levels of formality.
Types of Ontology Design Patterns (OPs)

- We also distinguish between ontological resources that are not OPs and Ontology Design Anti-Patterns (AntiOPs)
Presentation OPs

Definition

• Presentation OPs deal with usability and readability of ontologies from a user perspective.

• They are meant as good practices that support the reuse of patterns by facilitating their evaluation and selection.

• Two types:
  • Naming OPs
  • Annotation OPs
Naming OPs

Definition

• 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.
Examples of Naming OPs 1/2

• Namespace declared for ontologies.

• It is recommended to use the base URI of the organization that publishes the ontology

• followed by a reference directory for the ontologies
  • e.g. http://www.loa-cnr.it/ontologies/

• It is also important to choose an approach for encoding versioning, either on the name, or on the reference directory
Examples of Naming OPs  2/2

- Class names

- They should not contain plurals, unless explicitly required by the context
  - Names like Areas is considered bad practice, if e.g. an instance of the class Areas is a single area, not a collection of areas

- It is also recommended to use readable names instead of e.g. alphanumerical codes
  - Non-readable name can be used (even if not recommended) if associated to proper annotations (see Annotation OPs)

- It is useful to include the name of the parent class as a suffix of the class name
  - e.g. MarineArea rdfs:subClassOf Area

- Class names conventionally start with a capital letter
  - e.g. Area instead of area
Annotation OPs

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

- RDF Schema labels and comments (crucial for manual selection and evaluation)

- Each class and property should be annotated with meaningful labels
  - i.e., by means of the annotation property rdfs:label, with also translations in different languages.

- Each ontology and ontology element should be annotated with the rationale they are based on
  - i.e., by means of the annotation property rdfs:comment
Reasoning OPs

Definition

- Reasoning OPs are applications of Logical OPs oriented to obtain certain reasoning results, based on the behavior implemented in a reasoning engine.
Examples of Reasoning OPs

• Precise
  • Classification
  • Subsumption
  • Inheritance
  • Materialization
  • De-anonymizing
  • Normalization [6]

• Approximate
  • Approximate classification
  • Similarity induction
  • Taxonomy induction
  • Relevance detection
  • Latent semantic indexing
  • Automatic alignment
Classification and Subsumption RPs

- **Automatic classification**
  - Yes-Man(x) =_{df} Man(x) \land \exists y(\text{hasFiancee}(x,y))
  - Man(John)
  - hasFiancee(John,Mary)
  - \therefore Yes-Man(John)

- **Automatic subsumption**
  - Yes-Man(x) =_{df} Man(x) \land \exists y(\text{hasFiancee}(x,y))
  - ItalianMan(x) \Rightarrow Man(x)
  - hasFrenchFiancee(x,y) \Rightarrow hasFiancee(x,y)
  - \therefore ((ItalianMan(x) \land \exists y(\text{hasFrenchFiancee}(x,y)) \Rightarrow Yes-Man(x))
Inheritance and Materialization RPs

- **Inheritance**
  - Man\( (x) \Rightarrow \) Human\( (x) \)
  - Yes-Man\( (x) \Rightarrow \) Man\( (x) \)
  - \( \therefore \) (Yes-Man\( (x) \Rightarrow \) Human\( (x) \))

- **Materialization**
  - hasFiancee\( (x,y) \Leftrightarrow \) hasFiance\( (y,x) \)
  - hasFiancee\( (John,Mary) \)
  - \( \therefore \) hasFiance\( (Mary,John) \)
Construction RP

- **Query result construction**
  - `CONSTRUCT { ?x insanelyDesires ?z }
    WHERE { ?x hasFiancee ?y .
    ?y hasFemaleFriend ?z . }
  - `hasFiancee(John,Mary)
  - `hasFemaleFriend(Mary,Pamela)
  - `∴ insanelyDesires(John,Pamela)
Rule firing RP

- **SWRL rule firing**
  - \((\text{hasFiancee}(x,y) \land \text{hasFemaleFriend}(y,z)) \Rightarrow \text{insanelyDesires}(x,z)\)
  - \(\text{hasFiancee}(\text{John},\text{Mary})\)
  - \(\text{hasFemaleFriend}(\text{Mary},\text{Pamela})\)
  - \(\therefore \text{insanelyDesires}(\text{John},\text{Pamela})\)
Normalization

- Normalizations [5,6]:
  - Name all relevant classes, so that no anonymous complex class descriptions are left (restriction de-anonymizing)
  - Name anonymous individuals (skolem de-anonymizing)
  - Materialize the subsumption hierarchy (automatic subsumption) and normalize names
  - Instantiate the deepest possible class or property ("leaf")
  - Normalize property instances (property value materialization)
Common misconceptions

- Disjointness of primitives
- Interpreting domain and range
- And and Or
- Quantification
- Closed and Open Worlds
Disjointness

• By default, primitive classes are not disjoint.
• Unless we explicitly say so, the description (Animal and Vegetable) is not inconsistent.
• Similarly with individuals -- the so-called Unique Name Assumption (often present in DL languages) does not hold, and individuals are not considered to be distinct unless explicitly asserted to be so.
Domain and Range

- OWL allows us to specify the domain and range of properties.
- Note that this is not interpreted as a constraint as you might expect.
- Rather, the domain and range assertions allow us to make inferences about individuals.
- Consider the following:
  - `ObjectProperty(employs domain(Company) range(Person))`
  - `Individual(IBM value(employs Jim))`
- If we haven’t said anything else about IBM or Jim, this is not an error. However, we can now infer that IBM is a Company and Jim is a Person.
And/Or and quantification

• The logical connectives And and Or often cause confusion
  • Tea or Coffee?
  • Milk and Sugar?
• Quantification can also be contrary to our intuition.
  • Universal quantification over an empty set is true.
  • Aldo is a member of restriction(insanelyDesires allValuesFrom beetle)
• Existential quantification may imply the existence of an individual that we don’t know the name of.
  • Aldo is a member of restriction(insanelyDesires someValuesFrom FemaleFriend)
Close and Open World assumptions

• The standard semantics of OWL makes an Open World Assumption (OWA).
  • We cannot assume that all information is known about all the individuals in a domain.
  • Negation as contradiction
    • Anything might be true unless it can be proven false

• Closed World Assumption (CWA)
  • Named individuals are the only individuals in the domain
  • Negation as failure.
    • If we can’t deduce that x is an A, then we know it must be a (¬ A).
    • Facilitate reasoning about a particular state of affairs.
Correspondence OPs

**Definition**

- Correspondence OPs include Reengineering OPs and Mapping OPs.

- *Reengineering* OPs provide designers 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

Definition

• 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. See also N-ary relation transformation pattern
Schema Reengineering OP example: kos2skosABox

KOS $\mapsto$ skos:ConceptSchema  \hspace{1cm} (2.1)
Descriptor $\mapsto$ skos:Concept  \hspace{1cm} (2.2)
Broader Term $\mapsto$ skos:broader  \hspace{1cm} (2.3)
Related Term $\mapsto$ skos:related  \hspace{1cm} (2.4)

- The rule (2.1) states that, given a KOS, it maps to an instance of the class skos:ConceptSchema.
- The rule (2.2) maps each “Descriptor” from a KOS to a specific instance of the class skos:Concept.
- The rule (2.3) relates to the case of having two “Descriptors” d1 and d2 in a KOS, where d1 has “Broader Term” d2. Given the corresponding instances of skos:Concept skos:c1 and skos:c2, the broader term relationship between d1 and d2 maps to an object property value having the subject skos:c1, the object property skos:broader, and the object skos:c2.
- The rule (2.4) relates to the case of having two “Descriptors” d1 and d2 in the KOS that are “Related Terms”. Given the corresponding instances of skos:Concept skos:c1 and skos:c2, the related term relationship between d1 and d2 maps to a (symmetric) object property value having the subject skos:c1, the object property skos:related, and the object skos:c2.
Mapping OPs

Definition

- Mapping OPs refer to the semantic relations between mappable elements:
  - equivalent to, (not equivalent to)
    - foaf:Agent ≡ wn16:Agent-3
  - contained in, (not contained in)
    - foaf:Person ⊑ geo:SpatialThing
  - overlap with
    - foaf:Person ∩ dul:Person
  - disjoint with
    - (dul:PhysicalPerson ∩ dul:SocialPerson) = Φ

- Also called “correspondence patterns” in [16]
- We also consider an additional semantic relation that we call cloned from
  - ontology element oe₁ in one ontology is the clone of an ontology element oe₂ in another ontology
  - this relation is put in place when extracting a Content Ontology Design Pattern (see later)
Structural OPs

- Structural OPs includes Logical OPs and Architectural OPs.

- Architectural OPs affect the overall shape of the ontology either internally or externally.
  - i.e., an internal Architectural OP identifies a composition of Logical OPs that are to be exclusively used in the design of an ontology.

- Logical OPs are compositions of logical constructs that solve a problem of expressivity.
**Architectural OPs**

*Definition*

- 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

- 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.
Examples of Internal APs

- **Taxonomy**
  - A hierarchical structure of classes only related by subsumption relations.

- **Lightweight ontology. Taxonomy + other features, e.g.**:
  - A class can be related to other classes through the `disjointWith` relation.
  - Object and datatype properties can be defined and used to relate classes.
  - A specific domain and range can be associated with defined object and datatype properties.
Taxonomy AP

- **Intent**
  - To create an ontology consisting only of a subsumption graph
Primitives-Modifiers-Definables AP 1/2

- Intent: to create a compositional content architecture within an ontology
  - Choose some main axes
  - Add abstractions where needed; identify relations
  - Identify definable things, make names explicit
Modular AP

- Intent
  - To represent an ontology into self-consistent pieces, according to some criterion, and with an explicit ordering
Stratified AP (*external* AP)

- **Intent**
  - To create a layering of modules, according to some criterion

![Ontology Diagram]

- **Foundational ontology** (domain-independent)
  - {Object, Process, Part, Time, Location, Representation, Plan, …}

- **Core ontology** (specific domain-independent)
  - {Work of art, Painting technique, Author, Artistic period, Plastic art, Interpretation, …}

- **Domain ontology**
  - {Sculpture, Restoration, Mythical being, Caryatid, Doric order, Armilla, Fresco, …}
Logical OPs

Definition

• 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
  • i.e. 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: \( \forall R.C [7] \)

colloquially means “every R must be a C”

formally: \( \exists R.\top \sqcap \forall R.C \)

in OWL:

the combination of an owl:allValuesFrom restriction with an owl:someValuesFrom restriction.
Transformation patterns

Definition

• Transformation patterns translate a logical expression from a logical language into another, which approximates the semantics of the first, in order to find a trade-off between requirements and expressivity

• We describe transformation patterns by two diagrams at different levels:
  • The first diagram shows the meta model elements needed for representing the pattern in OWL DL. Such elements are defined in http://www.loa-cnr.it/codeps/owl/owl10a.owl, an OWL ontology that encodes OWL DL constructs in a metamodel. The ontology is referred to by the prefix “a:”

  • The second diagram shows an example of usage for the Logical OP
Examples of Transformation patterns: N-ary relation (1/2)
Examples of Transformation pattern: N-ary relation (2/2)

But beware of identification constraints! [15]