Ontology Design Patterns

Aldo Gangemi[1] and Valentina Presutti[2]

Institute for Cognitive Sciences and Technology (ISTC-CNR), Rome
aldo.gangemi@cnr.it valentina.presutti@istc.cnr.it

1 Introduction

Computational ontologies in the context of information systems are artifacts that encode a description of some world (actual, possible, counterfactual, impossible, desired, etc.), for some purpose. They have a (primarily logical) structure, and must match both domain and task: they allow the description of entities whose attributes and relations are of concern because of their relevance in a domain for some purpose, e.g. query, search, integration, matching, explanation, etc.
Like any artifact, ontologies have a lifecycle: they are designed, implemented, evaluated, fixed, exploited, reused, etc. (cf. chapter 6 for an in-depth examination of ontology engineering methodologies).
In this chapter, we focus on patterns for ontology design [14, 18].
Despite the original ontology engineering approach, when ontologies were seen as "portable" components [22], and its enormous impact on Semantic Web and interoperability, today one of the most challenging and neglected areas of ontology design is *reusability.* The possible reasons include at least: *size* and *complexity* of the major reusable ontologies, *opacity* of design rationales in most ontologies, *lack of criteria* in the way existing knowledge resources (e.g. thesauri, database schemata, lexica) can be reengineered, and *brittleness* of tools that should assist ontology designers.
Nowadays, an average user that is trying to build or reuse an ontology, or an existing knowledge resource, is typically left with just some limited assistance in using unfriendly logical structures, some large, hardly comprehensible ontologies, and a bunch of good practices that must be discovered from the literature. A typical usage scenario includes e.g. a large set of web ontologies that are evaluated (usually in an implicit way) against the intended domain and tasks. The selected ontology (if any) is reused, and then an adaptation process is started in order to cope with the implicit requirements from an ontology project. This scenario is costly in many cases, and automatic selection mechanisms do not help with the adaptation process. Another typical sce-

nario includes so-called “reference” or “core” ontologies that are supposed to be directly reused and specialized. Unfortunately, even if well designed, they are usually large and cover more knowledge than what a designer might need. In this case, it is hard to reuse only the “useful pieces” of the ontology, and consequently the cost of reuse is higher than developing a new ontology from scratch.
On the other hand, the success of very simple and small ontologies like FOAF [6] and SKOS [31] shows the potential of really portable, or “sustainable” ontologies. The lesson learnt supports the new approach to ontology design, which is sketched here.
Under the assumption that there exist classes of problems that can be solved by applying common solutions (as it has been experienced in software engineering), we propose to support reusability on the design side specifically. We envision small (or cleverly modularized) ontologies with explicit documentation of design rationales, and best reengineering practices. These components need specific functionalities in order to be implemented in repositories, registries, catalogues, open discussion and evaluation forums, and ultimately in new-generation ontology design tools. In this chapter, we describe small, motivated ontologies that can be used as *building blocks* in ontology design. A formal framework for (collaborative) ontology design that justifies the use of building blocks with explicit rationales is presented in [18].
We call the basic *building blocks* to be used in ontology design *Content Ontology Design Patterns* (CP) [14]. CPs are small ontologies that mediate between use cases (problem types) and design solutions. They are used as modeling components: ideally, an ontology results from a composition of CPs, with appropriate dependencies between them, plus the necessary design expansion based on specific needs.
Throughout experiences in ontology engineering projects[1] as well as in other ongoing international projects that have experimented with these ideas, typical conceptual patterns have emerged out of different domains, for different tasks, and while working with experts having heterogeneous backgrounds. For example, a simple CP called **participation** (including objects taking part in events) emerges in domain ontologies as different as enterprise models [23], legal norms [19], sofware management [34], biochemical pathways [16], and fishery techniques [17]. Other, more complex CPs have also emerged in the same disparate domains. Moreover, since CPs are strictly related to small use cases, they are transparent with respect to the rationales applied to the design of a certain ontology. CPs are therefore an additional tool to achieve tasks such as ontology evaluation, matching, modularization, etc. For example, an ontology can be evaluated against the presence of certain patterns (which act as *unit tests* for ontologies, cf. [50] and chapter 14) that are typical of the tasks

[1] For example, in the projects *FOS*: http://www.fao.org/agris/aos/, *WonderWeb*: http://wonderweb.semanticweb.org, *Metokis*: http://metokis.salzburgresearch.at, and *NeOn*: http://www.neon-project.org

addressed by a designer. Furthermore, mapping and composition of CPs can facilitate ontology mapping and alignment/merging. Two ontologies drafted according to CPs can be mapped in an easier way: CP hierarchies will be more stable and well-maintained than local, partial, scattered ontologies. Finally, CPs can be also used in training and educational contexts for ontology engineers.
CPs are a very beneficial kind of patterns for ontology design, because they provide solutions to domain-oriented problems, and are directly reusable. On one hand, CPs are comparable to software engineering (SE) design patterns for what concerns the way they are documented and communicated. On the other hand, the intuition behind their usage is analogous to that of software engineering (object oriented) reusable libraries, e.g. Java libraries. A similar intuition is at the base of approaches to modularization of ontologies e.g., [8], where the typical distinction between interface and implementation is used in order to distinguish between the module interface and the ontologies that a module encapsulates. CPs are compliant with this approach, and can be encapsulated in modules. However, this aspect is not key to the purpose of this chapter, and does not impact on their expected usage.
There are other types of Ontology Design Patterns (OPs) that are beneficial for different purposes and targeted at different types of users. A typology of OPs will be also introduced in this chapter.
In principle, OPs do not depend on any specific representation language[2]. In this context, we focus mainly on CPs; in order to provide the readers with concrete examples and a closer view on their exploitation on the Semantic Web, we have decided to refer to OWL CPs (cf. chapter 4 for details on OWL). In fact, CPs fit well with Semantic Web requirements for reuse and interoperability of ontologies and data, and as part of our work we have set up the ontologydesignpatterns.org web portal, which collects and makes them available on the Web [36].
Chapter's content is organized as follows: section 1.1 gives some background notions; section 2 introduces the types of OPs, defines them, and provides the reader with some examples; section 3 presents a sample catalogue of CPs; section 4 describes ways to create and work with CPs, and section 5 presents an example of their application. Finally, section 6 provides some conclusions and remarks.

1.1 Background

In the seventies, the architect and mathematician Christopher Alexander introduced the term "design pattern" for shared guidelines that help solve design problems. In [1] he argues that a good (architectural) design can be achieved by means of a set of rules that are "packaged" in the form of patterns, such as "courtyards which live", "windows place", or "entrance room". Design patterns are then assumed as archetypal solutions to design problems in a certain

[2] With the exception of Logical OPs.

context.
Taking seriously the architectural metaphor, the notion has been eagerly endorsed by software engineering [21, 12, 29], and DBMS applications with so-called data model patterns [27]. In these areas, *pattern* is used as a general term for formatted guidelines in software reuse, and, more recently, has also appeared in requirements analysis, conceptual modelling, and ontology engineering [7, 24, 39, 11, 48, 44][3]. Traditionally, design patterns appear more like a collection of shortcuts and suggestions related to a class of context-bound problems and success stories. Software engineering patterns are largely used for documenting software [26], and there is software support for automatic code generation based on them (see e.g the Eclipse functionality for generating factory methods[4], and the *Whole platform*[5]). Furthermore, there is recent work going towards a more formal encoding of design patterns (notably [3, 24, 30]), and even towards using ontology patterns to encode software engineering patterns [34].
Ontology engineering literature has tackled the notion of design pattern at least since [7, 39], while in the context of Semantic Web research and application, where OPs are now a hot topic, the notion has been introduced by [45, 16, 38, 48] and has been approached also by the W3C Semantic Web Best Practices and Deployment Group[6]. In particular, [16, 48] take a foundational approach that anticipates that presented in [14, 37] (which are closely related to this chapter). Some work [4] has also attempted a learning approach (by using case-based reasoning) to derive and rank patterns with respect to user requirements. The research has also addressed domain-oriented best practices and patterns, e.g. to express sequences in OWL [10], for content objects and multimedia [2] (cf. chapter 21), software components (cf. chapter 18), business modelling and interaction [20], medical [46, 43] (cf. chapter 17).

2 Types of Ontology Design Patterns

An Ontology Design Pattern (OP) is a modeling solution to solve a recurrent ontology design problem. We have identified several types of OPs, and have grouped them into six families (cf. Figure 1): *Structural* OPs, *Correspondence* OPs, *Content* OPs (CPs), *Reasoning* OPs, *Presentation* OPs, and *Lexico-Syntactic* OPs.

Although this chapter mainly focuses on CPs, in this section we give an overview of the OP families, with some examples. For more details, the reader can refer to [37].

[3] In software engineering, formal approaches to design patterns, based on dedicated ontologies, are being investigated, e.g. in so-called *semantic middleware* [34]
[4] Eclipse (http://www.eclipse.org/) is a programming environment used for developing Java projects
[5] http://whole.sourceforge.net/
[6] See http://www.w3.org/2001/sw/BestPractices/

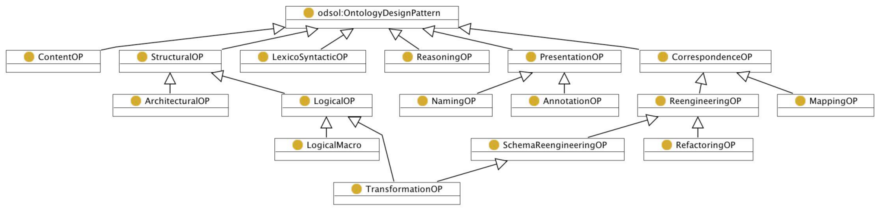


Fig. 1. Ontology Design Pattern types

Structural OPs

Structural OPs include Logical OPs and Architectural OPs. Logical OPs are compositions of logical constructs that solve a problem of expressivity, while Architectural OPs affect the overall shape of the ontology either internally or externally.
Logical OPs are only expressed in terms of a logical vocabulary, because their signature (the set of predicate names, e.g. the set of classes and properties in an OWL ontology) is empty (with minor exceptions, e.g. the default inclusion of `owl:Thing` in OWL). On one hand, Logical OPs are independent from a specific domain of interest (i.e. they are content-independent), on the other hand, they depend on the expressivity of the logical formalism that is used for representation. In other words, Logical OPs help to solve design problems where the primitives of the representation language do not directly support certain logical constructs. For example, if the representation language is OWL, and a designer needs to represent a relation between more than two elements, a Logical OP is needed in order to express an n-ary relation semantics by only using class and binary relation primitives. The root of *Logical OPs* can be found in [5], where so-called description logics were conceived as a way of representing knowledge in a structural manner by singling out the most relevant and tractable patterns from first-order logic (and beyond). The first proposal for a library of Semantic Web logical patterns is [45]. We can informally divide Logical OPs into two types:
Logical macros provide a shortcut to model a recurrent intuitive logical expression e.g., the combination of *owl:allValuesFrom* restriction with *owl:someValuesFrom* restriction.
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. For example, the so called **n-ary relation** pattern, documented in [33] with respect to OWL, is a transformation pattern from first-order logic to OWL DL. Other Logical OPs are documented in [37, 47, 33].
The application of Logical OPs has consequences on the results and efficiency of reasoning procedures. They can be used in order to document design choices and are particularly suitable for teaching good practices of ontology design

as they provide designers with solutions to represent complex logical expressions.
Architectural OPs affect the overall shape of the ontology: their aim is to constrain ‘how the ontology should look like’. They can be of two types: (i) *internal*, defined in terms of collections of Logical OPs that have to be exclusively employed when designing an ontology e.g., an OWL species (cf. chapter 4), or the varieties of description logics (cf. chapter 1); (ii) *external*, 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 (according to definitions given in [25]). The modules are connected by the *import* operation.
Architectural OPs emerged as design choices motivated by specific needs e.g., computational complexity constraints. Such OPs are also useful as reference documentation for those initially approaching the design of an ontology.

Reasoning OPs

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 include: **classification**, **subsumption**, **inheritance**, **materialization**, **de-anonymizing**, etc.
Reasoning OPs, when declared on top of an ontology, inform about the state of that ontology, and let a system decide what reasoning has to be performed on the ontology in order to carry out queries, evaluation, etc. Examples of Reasoning OPs are so called *normalizations*. In [51, 52] five normalizations have been identified (cf. chapter 14).

Correspondence OPs

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 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.
Reengineering OPs are described in terms of metamodel transformation rules. We distinguish two types of Reengineering OPs.
Schema reengineering patterns are rules for transforming e.g., a non-OWL DL metamodel into an OWL DL ontology. For example, consider the use of SKOS [31] for Knowledge Organization Systems (KOS) reengineering to a knowledge base (an OWL ABox), based-on the SKOS TBox. Transformation Logical OPs

are a kind of schema reengineering patterns. In principle, all modeling problems can be represented as higher-order logical expressions, and if we have to represent them e.g. in OWL DL, we implicitly apply a schema reengineering pattern in order to stay within the expressivity of OWL DL. However, we also (pragmatically) distinguish between transformation and schema reengineering patterns because of the different intention of the designer. In the first case, the designer wants to directly represent a modeling solution in a certain representation formalism e.g., OWL DL[7], while in the second case the designer wants to reengineer e.g., an existing non-OWL DL model into an OWL DL ontology.
Refactoring patterns provide designers with rules for transforming, i.e. *refactoring*, e.g., an existing OWL DL source ontology into a new OWL DL target ontology. In this case, the transformation rule has the effect of changing the type of the ontology elements that are involved in the refactoring. For example, let's consider the case in which an ontology defines an object property for representing the relation of *preparing a coffee*, which holds between *agents* and *coffees*. Now, let's consider a change of requirements, so that a designer has to represent that the coffee is prepared by an agent at a certain time by using a certain tool. In order to address such a change in OWL DL, a designer has to apply an **n-ary relation** Logical OP, because *preparing a coffee* has now four arguments: *agent*, *coffee*, *time interval*, and *tool*. The **n-ary relation** Logical OP plus the description of how to apply it in order to replace an object property from an existing ontology is a Refactoring OP.
Mapping Ontology Design Patterns Mapping OPs refer to the possible semantic relations between mappable elements, as defined in [25]. There are three basic semantic relations that are used for mapping assertions: *equivalence*, *containment*, and *overlap*. They can be supplemented by their negative counterparts i.e., *not equivalent*, *not contained*, and *not overlap* or *disjoint*, respectively. Mapping OPs provide designers with solutions to relate two ontologies without changing the logical types (e.g. owl:Class) of the ontology elements involved.

Presentation OPs

Presentation OPs deal with usability and readability of ontologies from a user perspective. They are meant as good practices that support the reuse of ontologies by facilitating their evaluation and selection. Examples are Naming OPs and Annotation OPs . The former are conventions about how to create names for namespace, files, and ontology elements in general (classes, properties, etc.). They are good practices that boost ontology readability and understanding by humans, by supporting homogeneity in naming procedures. Annotation OPs provide annotation properties or annotation property schemas

[7] In the pragmatics of an ontology designer, the fact that all modeling solutions are representable as higher-order logic expressions is hardly relevant, and such implicit reengineering has been never documented as actually happening.

that can be used in order to improve the understandability of ontologies and their elements.
An example of Naming OP relates to **namespace declared for ontologies**. It is recommended to use the base URI of the organization that publishes the ontology (e.g. `http://www.w3.org` for the W3C, `http://www.fao.org` for the FAO, `http://www.loa-cnr.it` for the Laboratory for Applied Ontologies (LOA) etc.) followed by a reference directory for the ontologies (e.g. `http://www.loa-cnr.it/ontologies/`). Additionally, it is also important to choose an approach for encoding versioning, either on the name, or on the reference directory.

Lexico-Syntactic OPs

Lexico-Syntactic OPs are linguistic structures or schemas that consist of certain types of words following a specific order, and that permit to generalize and extract some conclusions about the meaning they express. They are useful for associating simple Logical and Content OPs with natural language sentences e.g., for didactic purposes.

Content Ontology Design Patterns (CPs)

CPs encode *conceptual*, rather than *logical* design patterns. In other words, while Logical OPs solve design problems independently of a particular conceptualization, CPs propose patterns for solving design problems for the domain classes and properties that populate an ontology, therefore addressing *content* problems [14]. CPs are instantiations of Logical OPs (or of compositions of Logical OPs), featuring a non-empty signature. Hence, they have an explicit non-logical vocabulary for a specific domain of interest (i.e. they are content-dependent). CPs provide solutions to domain modeling problems and affect only the specific region of the ontology dealing with such domain modeling problems. They are typically reused by applying specialization, extension, and composition to them. In principle, CPs do not depend on any specific language, however in order to reuse them as *building blocks*, they have to be implemented in some way. In the context of this chapter, we deal with CPs in a Semantic Web context. Hence, we use OWL as a reference formalism for representation.

3 Towards a catalogue and repository of CPs

In this section we focus on CPs. We define them, and explain the dependencies between CPs and use cases (section 3.1). Section 3.2 lists the characteristics that differentiate CPs as special ontologies (such characteristics cross the boundaries between ontology engineering, cognitive science, and linguistics). Finally, we describe two CPs (section 3.3).
The way to document Ontology Design Patterns (OPs) can be compared

to the typical way followed for SE patterns. The mainstream approach for describing SE patterns is to use a template, although there is no standard format. A description of the most well-known SE pattern templates can be found at Martin Fowler's web site.[8] The templates used for describing SE patterns follow quite closely that suggested by Alexander [1]: given an *artifact type*, the pattern provides *examples* of it, its *context*, the *problem* addressed by the pattern, the involved "*forces*" (requirements and constraints), and a *solution*. In order to describe CPs, we follow a similar approach: each CP is associated with a *catalogue entry* including the following set of information fields.
Name provides a name for the pattern; **Intent** describes the *Generic Use Case* addressed by the pattern; **Competency questions** contains examples of competency questions that the knowledge base associated with the CP needs to address; **Also Known as** provides other names (if any) with which the pattern is known; **Scenarios** provides examples of requirements, expressed in natural language, which can be modeled by using the pattern; **Diagram** depicts a UML class diagram representing the pattern; **Elements** describes the elements (classes and relations) included in the pattern, and their role within the pattern; **Consequences** provides a description of the benefits and/or possible trade-offs when using the pattern; **Known uses** gives examples of realistic ontologies where the pattern is used, **Extracted from/Reengineered from** provides the reference ontology/conceptual schema (if any), from which the pattern has been extracted/reused; **Related patterns** indicates other patterns (if any) that are either a *specialization*, *generalization*, *composition*, or *component* of the pattern being described. Furthermore, this field may indicate other patterns that are typically used in conjunction with the described one. Important similarities and differences with other patterns can be also described here; **Building block** provides references to implementations of the pattern, a URI. In the case of CPs for Semantic Web ontologies, this field provides the URI of an OWL file (containing an implementation of the pattern). Section 3.3 contains two examples of CPs that are described by means of a simplified version of the catalogue template. Such a catalogue can be found at the ontologydesignpatterns.org web portal [36], a dedicated wiki site through which a *lightweight* repository of CPs can be accessed. In fact, [36] allows users to download, propose, and discuss CPs. Furthermore, each CP includes a set of annotations[9] that can be exploited by Semantic Web applications. The reader can refer to chapter 25 for more details on ontology repositories.

3.1 CPs and competency questions

Content Ontology Design Patterns (CPs) are reusable solutions to recurrent modelling problems. As known from a long time in conceptual modeling (cf.

[8] http://www.martinfowler.com/articles/writingPatterns.html#CommonPatternForms
[9] http://www.ontologydesignpatterns.org/schema/cpannotationschema.owl

the difference between class and use case diagrams in UML) and knowledge engineering (cf. the distinction between domain and task ontologies in UPML [32]), these problems have two components: a domain and a use case (or task). A same domain can have many use cases (e.g. different scenarios in a clinical information context), and a same use case can be found in different domains (e.g. different domains with a same "competence finding" scenario).
Ontologies are usually considered models for a domain, but their use case is usually unknown. As reusable solutions, CPs must explicitly encode both a domain and a use case. Since use cases are extremely diversified, a catalogue of CPs requires the notion of a "Generic Use Case" (GUC), i.e. a generalization of use cases that can be provided as examples for an issue of domain modelling. A GUC is the expression of a recurrent scenario in different domain ontology projects.
Being generic at the use case level allows us to divide, or to refactor the design problems of a use case, by composing different GUCs. We can hierarchically organize GUCs from the most generic to the most specific ones, and from the "purest" (e.g.: 'which objects take part in a certain event?') to the most articulated and applied ones (e.g.: 'what protein is involved in the Jack/Stat biochemical pathway?').
The intuition underlying GUC hierarchies is based on a methodological observation: ontologies must be built out of domain tasks that can be captured by means of *competency questions* [23].
A competency question is a typical query that an expert might want to submit to a knowledge base of its target domain, for a certain task. In principle, an accurate domain ontology should specify *all and only* the conceptualizations required in order to answer all the competency questions formulated by, or acquired from, experts. A GUC cannot do much as a guideline, unless we are able to find formal patterns that encode it. CPs are the solution to this issue.
Based on the above assumptions, we define a CP as:

> *CPs are distinguished ontologies. They address a specific set of competency questions, which represent the problem they provide a solution for. Furthermore, CPs show certain characteristics, i.e. they are: computational, small, autonomous, hierarchical, cognitively relevant, linguistically relevant, and best practices.*

3.2 General characteristics of CPs

CPs are components that represent, and possibly help solving a modelling problem arising across different use cases. E.g. the *agent-role* pattern provides a solution to represent agents that play some role. We have sketched their theoretical basis in section 2, and explained their dependance on use cases (section 3.1). Before providing a sample list of CPs against an example use case (section 3.3), we now describe a more inclusive set of general, pragmatic features of CPs. These features, besides positioning CPs in a wider

scientific context, give hints on how to discover or to extract CPs from existing knowledge resources.
Computational components CPs are language-independent, and should be encoded in a higher-order representation language. Nevertheless, their (sample) representation in OWL is needed in order to (re)use them as building blocks over the Semantic Web.
Small, autonomous components Regardless of the particular way a CP has been created, it is a *small, autonomous* ontology. Smallness (typically two to ten classes with relations defined between them) and autonomy of CPs facilitate ontology designers: composing CPs enable them to govern the complexity of the whole ontology, because of the explicit rationales and the amount of know-how provided by the users of a same CP library. Smallness also allows diagrammatical visualizations that are aesthetically acceptable and easily memorizable.
Hierarchical components A CP can be an element in a partial order, where the ordering relation requires that at least one of the classes or properties in the pattern is specialized. A hierarchy of CPs can be built by specializing or generalizing some of the elements (either classes or relations).
Inference-enabling components There are combinations of ontology elements that do not allow any useful inference e.g., a taxonomy with two sibling classes, an object property alone, etc. A CP allows some form of inference e.g. a taxonomy with two sibling disjoint classes, a property with explicit domain and range set, a property and a class with a universal restriction on that property, etc.
Cognitively relevant components CP visualization must be intuitive and compact, and should catch relevant, "core" notions of a domain [14].
Linguistically relevant components Many CPs nicely match linguistic patterns called *frames*. A frame can be described as a lexically founded OP. The richest repository of frames is FrameNet [3]. Frames can be used for validating CPs with respect to lexical coverage, for lexicalizing them, and can be reengineered as CPs.
Best practice components A CP should be used to describe a "best practice" of modelling. Best practices are intended here as *local*, thus derived from experts, emerging from real applications. The quality of CPs is currently based on the personal experience and taste of the proposers, or on the provenance of the knowledge resource where the pattern comes from. However, evidence from reusability across different projects, large-scale applications, and open rating systems will provide a good base for CP evaluation.

3.3 Samples of CP catalogue entries

In this section we show two CPs taken from [36], Each CP is presented in a catalogue-like way, and with reference to the OWL language.
For space reasons, we describe each CP with a simplified catalogue entry composed of: the *Name* (including possible alternative names), the *Intent* (i.e.,

the GUC), **Competency questions**, some *Examples* of its application, the *Diagram* describing its structure, the *Elements* and the role they play in the pattern, and some *General Remarks* that indicate general guidelines about how to use it, including relations to other CPs.
The complete entry[10] also contains a field named *building block* that provides references to implementations of the pattern, i.e. repository of reusable components. In the case of CPs for Semantic Web ontologies, this field provides the URI of an OWL file (containing an implementation of the pattern).
We have used TopBraid Composer[11] in order to produce the OWL encoding. With the same tool, we automatically generated a diagrammatical visualization based on a UML profile for OWL. UML classes (boxes) are used in order to depicts OWL classes. Two kinds of OWL classes can be visualized in a diagram: named classes (`owl:Class`, in white boxes) and anonymous classes (in grey boxes), e.g. `owl:Restriction` with `owl:someValuesFrom`. UML generalization (arrow with a large end) corresponds to `rdfs:subClassOf`, while UML association (arrow with a small end) corresponds to `owl:ObjectProperty`. Finally, UML class attributes (statements inside white boxes) are used in order to indicate either `rdfs:domain` and `rdfs:range`, or `owl:Restriction` with `owl:allValuesFrom`.
When a class name is preceded by a prefix e.g.: `sit:`, it is interpreted as a class imported (e.g. by `owl:imports`) from another (typically more general) CP that is indexed by means of that prefix.
In the rest of this section we use the OWL terminology in order to describe the proposed design solutions e.g., object property, datatype property, etc.

The information realization CP

The **information realization** CP is extracted from the Dolce+DnS Ultra Lite ontology[12], and represents the relations between information objects like poems, songs, formulas, etc., and their physical realizations like printed books, registered tracks, physical files, etc..

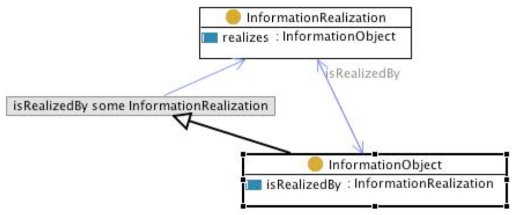

InformationRealization
realizes : InformationObject
isRealizedBy
isRealizedBy some InformationRealization
InformationObject
isRealizedBy : InformationRealization

Fig. 2. The information realization CP UML graphical representation.

[10] See [36, 37].
[11] http://www.topbraidcomposer.com/
[12] http://www.ontologydesignpatterns.org/ont/dul//DUL.owl

The **information realization CP** is associated with information according to the catalogue entry fields reported below:
Intent: to represent relations between information objects and their physical realizations.
Competency questions: which physical object realizes a certain information object? Which information object is realized by a certain physical object?
Diagram: Figure 2 shows a UML diagram of the information realization CP.
Elements:

- `InformationObject:` a piece of information, such as a musical composition, a text, a word, a picture, independently from how it is concretely realized.
- `InformationRealization:` a concrete realization of an InformationObject, e.g. the written document containing the text of a law.
- `realizes:` a relation between an information realization and an information object, e.g. the paper copy of the Italian Constitution realizes the text of the Constitution.
- `isRealizedBy:` a relation between an information object and an information realization, e.g. the text of the Constitution is realized by the paper copy of the Italian Constitution.

General remarks: this CP[13] allows to distinguish between information encoded in an object and the possible physical representations of it. The Multimedia ontology (cf. chapter 21) uses this CP.

The Time Indexed Person Role CP

The **time indexed person role** is a CP that represents time indexing for the relation between persons and roles they play e.g., George W. Bush was the president of the United States in 2007. This CP is also extracted from the Dolce+DnS Ultra Lite ontology.
According to its associated catalogue entry, the main information associated with this CP are the following:
Intent: to represent time indexing for the relation between persons and roles they play.
Competency questions: who was playing a certain roles during a given time interval? When did a certain person play a specific role?
Diagram: see Figure 3, the elements which compose the CP are described in the **Elements** field.
Elements:

- `Entity:` anything: real, possible, or imaginary, which some modeller wants to talk about for some purpose.
- `Person:` persons in commonsense intuition, i.e. either as physical agents (humans) or social persons.

[13] http://www.ontologydesignpatterns.org/cp/owl/informationrealization.owl

- `Role:` a Concept that classifies a Person
- `TimeInterval:` any region in a dimensional space that aims at representing time.
- `TimeIndexedPersonRole:` a situation that expresses time indexing for the relation between persons and roles they play.
- `hasRole:` a relation between a Role and an Entity, e.g. 'John is considered a typical rude man'; your last concert constitutes the achievement of a lifetime; '20-year-old means she's mature enough'.
- `isRoleOf:` a relation between a Role and an Entity, e.g. the Role 'student' classifies a Person 'John'.
- `isSettingFor:` a relation between time indexed role situations and related entities, e.g. 'I was the director between 2000 and 2005 ', i.e.: the situation in which I was a director is the setting for a the role of director, me, and the time interval.
- `hasSetting:` the inverse relation of `isSettingFor`.

General remarks: this CP[14] allows to assign a time interval to roles played by people.

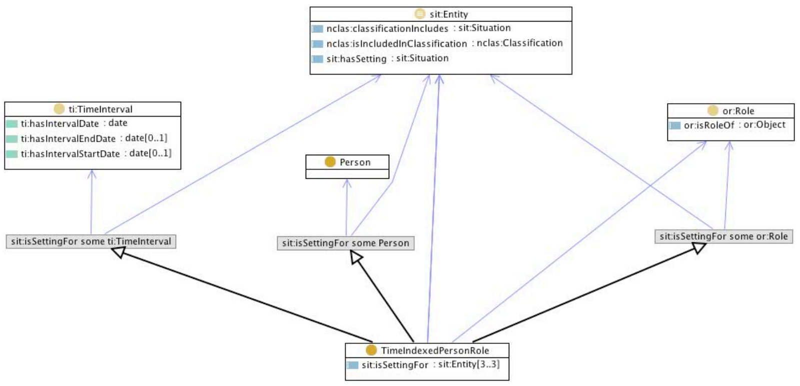

sit:Entity
nclas:classificationIncludes : sit:Situation
nclas:isIncludedInClassification : nclas:Classification
sit:hasSetting : sit:Situation
ti:TimeInterval
ti:hasIntervalDate : date
ti:hasIntervalEndDate : date[0..1]
ti:hasIntervalStartDate : date[0..1]
Person
or:Role
or:isRoleOf : or:Object
sit:isSettingFor some ti:TimeInterval
sit:isSettingFor some Person
sit:isSettingFor some or:Role
TimeIndexedPersonRole
sit:isSettingFor : sit:Entity[3..3]

Fig. 3. The time indexed person role CP UML graphical representation.

4 Creating and working with CPs

This section discusses how CPs can be created, and provides guidelines on how they can be practically (re)used. Section 4.1 describes four approaches to create CPs, while section 4.2 shows the main operations that are performed for reusing a CP, and describes the possible situations of CP selection and usage that can occur in practice.

[14] http://ontologydesignpatterns.org/cp/owl/timeindexedpersonrole.owl

4.1 Where do Content Ontology Design Patterns come from?

CP creation and usage rely on a common set of operations.

import: consists of including a CP in the ontology under development. This is the basic mechanism for reusing CPs (and ontologies in general). By importing a CP, the importing ontology ensures the set of inferences allowed by the CP in its corresponding knowledge base. Elements of an imported CP cannot be modified.

specialization: can be referred to ontology elements or to CPs. Specialization between ontology elements of a CP consists of creating sub-classes of some CP's class and/or sub-properties of some CP's properties. A CP c_1 is a specialization of a CP c if c_1 imports c, and at least one ontology element from c_1 specializes an ontology element from c.

generalization: A CP c_1 is a generalization of a CP c if c_1 imports c, and at least one ontology element from c_1 generalizes an element from c.

composition: consists of associating classes (properties) of one CP with classes (properties) of other CPs, by means of some OWL axiom.

expansion: consists of adding new classes, properties and axioms to the ontology to the aim of covering the requirements that are not addressed by the reused CPs.

CPs come from the experience of ontology engineers in modeling *foundational* (cf. chapter **??**), *upper-level*, *core* [15], or *domain* ontologies. Informally, the distinction between these kinds of ontologies relates to the degree by which an ontology covers the domain of interest, cf. chapter 0 for details. Assuming the above distinctions, there are four main ways of creating CPs, which can be summarized as follows:
Reengineering from other data models A CP can be the result of a reengineering process applied to different conceptual modeling languages, primitives, and styles. Knowledge resources that can be reengineered to produce candidate CPs are database schemas, knowledge organization systems such as thesauri, and lexica. For more references, the reader can refer to [20] that describes a reengineering approach for creating CPs starting from UML diagrams [35], workflow patterns [49], and data model patterns [27].
Specialization/Composition of other CPs A CP can be created by composing other CPs, or by specializing another CP, (both composition and specialization can be combined with expansion, see below).
Extraction from reference ontologies A CP can be *extracted* from an existing ontology, which acts as the "source" ontology. In this case, the CP corresponds to a fragment of the source ontology, which constitutes its axiomatic background context. A CP is axiomatized according to the fragment it extracts. E.g., the **co-participation** CP depends on a set of axioms from the DOLCE ontology [9], which state that an event has at least one participant, that co-participation requires two participants in a same event, that participants must participate at least partly at the same time, etc. If a modeller

specializes the **co-participation** CP for representing e.g. an academic lecture or a football match, the reasoning services will operate with reference to the **co-participation** axioms, without the need for encoding them again. However, a CP is autonomous, and only the axioms that have been extracted from the reference ontology are actually used by an ontology that reuses a CP. Therefore, reasoning services do not need to also process the general axiomatic context from the reference ontology.
Creation by combining extraction, specialization, generalization, and expansion The definition of a CP can be the result of an extraction (see above), followed by specialization and/or generalization of some ontology elements, and expansion[15].

4.2 How to use Content Ontology Design Patterns

Supporting reuse and alleviating difficulties in ontology design activities are the main goals of setting up a catalogue of CPs. In order to be able to reuse CPs, two main functionalities must be ensured: *selection* and *application*.
Selection of CPs corresponds to finding the most appropriate CP for the actual domain modeling problem. Hence, selection includes search and evaluation of available CPs. This task can be performed by applying procedures for ontology selection [41, 28] and evaluation [13] (cf. chapter 14).
Informally, a GUC i.e. the *intent* of a CP, must match an actual use case. Once a CP has been selected, it has to be applied to the domain ontology. Typically, application is performed by means of import, specialization, composition, or expansion (see section 4.1).
In realistic design projects, the operations are usually combined as it is shown by the example of section 5.

Several situations of matching between GUCs and actual use cases can occur, each associated with a different approach to using CPs. The following summary assumes a *manual* (re)use of CPs. However, an initial library of CPs is already available [36], and tool support to their selection and usage can take into account the principles informally explained in the summary below as base requirements.
Precise or redundant matching. The CP matches a GUC, which is either more complex or directly usable to describe the local use case: the CP has only to be *imported* in the domain ontology.
Broader matching. The CP matches a GUC that is more general than the local use case: the CP's catalogue entry may contain reference to less general CPs that specialize it. If none of them is appropriate, the CP has firstly to be *imported*, then it has to be *specialized* in order to cover the domain part to be represented.
Narrower matching. The CP matches a GUC that is more specific than the local use case: the CP's catalogue entry may contain references to more general

[15] See [37] for more details.

CPs. If none of them is appropriate, a the CP has firstly to be *imported*, then it has to be *generalized* according to the local requirements.
Partial matching. The CP partly matches a GUC that does not cover all aspects of the local use case (it is simpler): the CP's catalogue entry may contain references to CPs it is a component of. If none of such compound CPs is appropriate, the local use case has to be partitioned into smaller pieces. One of these pieces will be covered by the selected CP. For the other pieces, other CPs have to be selected. All selected CPs have to be *imported* and *composed*.
In all the above situation, *expansion* is performed when needed.

5 Use case example in the music industry domain

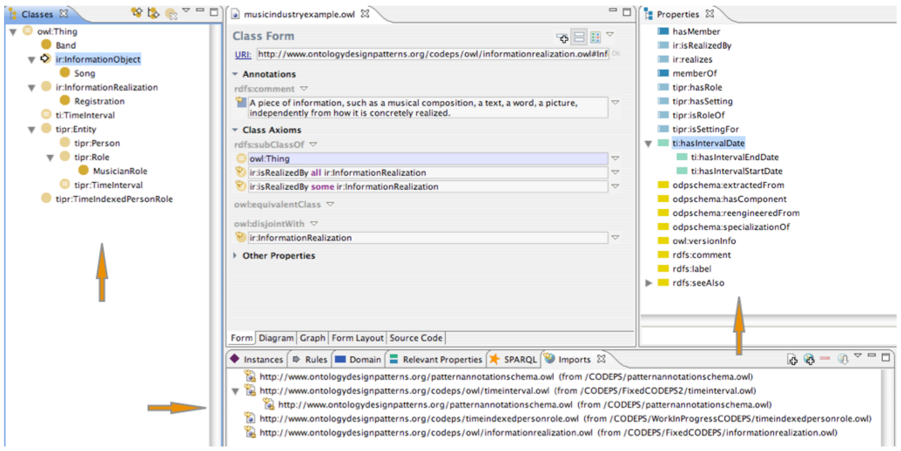
As an example of usage we design a small fragment of an ontology for the music industry domain. The ontology fragment has to address the following competency questions:

Which recordings of a certain song do exist in our archive?
Who did play a certain musician role in a given band during a certain period?

The first competency question requires to distinguish between a song and its recording, while the second competency question highlights the issue of assigning a given musician role e.g., singer, guitar player, etc., to a person who is member of a certain band, at a given period of time. The intent of the **information realization** is related to the first competency question with a *broader matching.* The intent of the **time indexed person role** partially and broadly matches the second competency question. Hence, we select these two CPs as building blocks for our ontology[16].
We proceed by importing and composing the two selected CPs in our ontology (the **information realization** CP is associated with the prefix *ir:*; the **time indexed person role** CP is associated with the prefix *tipr:*). Additionally, we might want to import the **time interval** CP[17] that allows us to assign a date to the time interval. In order to complete our ontology fragment we create: the class `Song` that specializes `ir:InformationObject`, the class `Recording` that specializes `ir:InformationRealization`, the class `MusicianRole` that specializes `tipr:Role`, the class `Band`, and the object property `memberOf` (and its inverse) with explicit domain i.e., `tipr:Person`, and range i.e., `Band`. A screenshot of the resulting ontology fragment is shown in Figure 4[18]. On the

[16] Notice that the second requirement would also require to represent membership relation between a person and a band. The **collection entity** CP available at http://www.ontologydesignpatterns.org/cp/owl/collectionentity.owl addresses membership. We do not include the description of this CP and its usage for the sake of brevity.
[17] Available at http://www.ontologydesignpatterns.org/cp/owl/timeinterval.owl
[18] The screenshot shows the TopBraid Composer interface, see http://www.topbraidcomposer.com

left side of the picture ontology classes are shown, on the right side there are ontology properties, while at the bottom there are the imported CPs. Notice



```
owl:Thing
  Band
  ir:InformationObject
    Song
  ir:InformationRealization
    Registration
  ti:TimeInterval
  tipr:Entity
    tipr:Person
    tipr:Role
      MusicianRole
    tipr:TimeInterval
  tipr:TimeIndexedPersonRole
```

```
musicindustryexample.owl
Class Form
URI: http://www.ontologydesignpatterns.org/codeps/owl/informationrealization.owl#Inf
Annotations
rdfs:comment
  A piece of information, such as a musical composition, a text, a word, a picture,
  independently from how it is concretely realized.
Class Axioms
rdfs:subClassOf
  owl:Thing
  ir:isRealizedBy all ir:InformationRealization
  ir:isRealizedBy some ir:InformationRealization
owl:equivalentClass
owl:disjointWith
  ir:InformationRealization
Other Properties
```

```
hasMember
ir:isRealizedBy
ir:realizes
memberOf
tipr:hasRole
tipr:hasSetting
tipr:isRoleOf
tipr:isSettingFor
ti:hasIntervalDate
  ti:hasIntervalEndDate
  ti:hasIntervalStartDate
odpschema:extractedFrom
odpschema:hasComponent
odpschema:reengineeredFrom
odpschema:specializationOf
owl:versionInfo
rdfs:comment
rdfs:label
rdfs:seeAlso
```

```
http://www.ontologydesignpatterns.org/patternannotationschema.owl (from /CODEPS/patternannotationschema.owl)
http://www.ontologydesignpatterns.org/codeps/owl/timeinterval.owl (from /CODEPS/FixedCODEPS2/timeinterval.owl)
  http://www.ontologydesignpatterns.org/patternannotationschema.owl (from /CODEPS/patternannotationschema.owl)
http://www.ontologydesignpatterns.org/codeps/timeindexedpersonrole.owl (from /CODEPS/WorkInProgressCODEPS/timeindexedpersonrole.owl)
http://www.ontologydesignpatterns.org/codeps/owl/informationrealization.owl (from /CODEPS/FixedCODEPS/informationrealization.owl)
```

Fig. 4. The music industry example.

that CPs can be very useful when they address issues in a specific domain. For this reason, an ontology fragment like this one might be proposed as a CP[19] if it is associated with a successful application in an ontology design project.

6 Conclusion and remarks

Ontology design is a crucial research area for semantic technologies. Many bottlenecks in the wide adoption of semantic technologies depend on the difficulty of understanding ontologies and on the scarcity of tools supporting their lifecycle, from creation to adaptation, reuse, and management. The lessons learnt until now, either from the early adoption of Semantic Web solutions or from local, organizational applications, put a lot of emphasis on the need for simple, modular ontologies that are accessible and understandable by typical computer scientist and field experts, and on the dependability of these ontologies on existing knowledge resources.[20]
In this chapter, we have described a breed of components, called Ontology Design Patterns (OPs), and tools that will support ontology design at the level that is more natural to domain experts and laymen, i.e. the level at which small, expertise-aware components can be assembled as easy-to-apply, easy-to-customize building blocks.

[19] See [36] area of proposed CP.
[20] An interesting review of evaluation, selection and reuse methods in ontology engineering is in [40].

The quality of these components is expected to be evaluated with respect to known good practices, as well as in the large testbed of organizational or web-scale open rating systems. In order to allow the maximum transparency and flexibility, OPs are supplied with a rich set of metadata for their explanation, rationale declaration, use case history, evaluation criteria, etc.
In this chapter, we have sketched a typology of OPs, then focused on Content Ontology Design Patterns (which are most beneficial to ontology design) in terms of their background, definition, communication means, related work beyond ontology engineering, exemplification, creation, and usage principles. There is still a lot of work to be carried out for populating repositories of patterns, discoverying or extracting them from existing ontologies, assisting users in their application, defining a robust semantics and algebra for them, etc. (cf. [42]). The larger context of ontology design research is still very young, and many ideas are just emerging, for example in (semi-)automatizing the creation and evaluation of ontologies, based only on informal documentation from users, a set of software components, and a repository of design patterns. In a larger report [37] and by setting up the ontologydesignpatterns.org web portal [36], we make some steps towards the open issues. Moreover, we have designed a set of experiments that are going to be performed in order to show OPs' effectiveness e.g., in teaching ontology design, lower the cost of ontology projects, etc. However, some initial experiences with PhD students classes, and the employment of CPs in of our own recent projects have provided us with concrete proof of the benefits deriving from their application.

References

1. Christopher Alexander. *The Timeless Way of Building.* Oxford Press, 1979.
2. Richard Arndt, Raphael Troncy, Steffen Staab, Lynda Hardman, and Miroslav Vacura. COMM: Designing a Well-Founded Multimedia Ontology for the Web. In *Proceedings of the 4th European Semantic Web Conference (ISCW'07)*, Busan Korea, November 2007. Springer.
3. Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The Berkeley FrameNet project. In Christian Boitet and Pete Whitelock, editors, *Proceedings of the Thirty-Sixth Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics*, pages 86–90, San Francisco, California, 1998. Morgan Kaufmann Publishers.
4. Eva Blomqvist. Fully automatic construction of enterprise ontologies using design patterns: Initial method and first experiences. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, zalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *OTM Conferences (2)*, volume 3761 of *Lecture Notes in Computer Science*, pages 1314–1329. Springer, 2005.
5. Ronald J. Brachman. A Structural Paradigm for Representing Knowledge. Ph.d. thesis, Harvard University, USA, 1977.
6. D. Brickley and L. Miller. FOAF Vocabulary Specification. Working draft, 2005.

7. Peter Clark, John Thompson, and Bruce Porter. Knowledge patterns. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 591–600, San Francisco, 2000. Morgan Kaufmann.
8. Mathieu d'Aquin, Peter Haase, Sebastian Rudolph, Jerome Euzenat, Antoine Zimmermann, Martin Dzbor, Marta Iglesias, Yves Jacques, Caterina Caracciolo, Carlos Buil Aranda, and Jose Manuel Gomez. NeOn Formalisms for Modularization: Syntax, Semantics, Algebra. Deliverable D1.1.3, NeOn project, 2008.
9. DOLCE - Project Home Page. http://dolce.semanticweb.org.
10. Nicholas Drummond, Alan Rector, Robert Stevens, Georgina Moulton, Matthew Horridge, Hai Wang, and Julian Sedenberg. Putting owl in order: Patterns for sequences in owl. In *OWL Experiences and Directions (OWLEd 2006)*, Athens Georgia, 2006.
11. Didier Dubois, Christopher A. Welty, and Mary-Anne Williams, editors. *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, June 2-5, 2004*. AAAI Press, 2004.
12. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, March 1995. ISBN-10: 0201633612 ISBN-13: 978-0201633610.
13. A. Gangemi, C. Catenacci, M. Ciaramita, and Lehmann J. Modelling Ontology Evaluation and Validation. In *Proceedings of the Third European Semantic Web Conference*. Springer, 2006.
14. Aldo Gangemi. Ontology Design Patterns for Semantic Web Content. In *M. Musen et al. (eds.): Proceedings of the Fourth International Semantic Web Conference*, Galway, Ireland, 2005. Springer.
15. Aldo Gangemi and Stefano Borgo, editors. *Proceedings of the EKAW*04 Workshop on Core Ontologies in Ontology Engineering, Northamptonshire (UK)*, volume 118. CEUR-WS, October 2004.
16. Aldo Gangemi, Carola Catenacci, and Massimo Battaglia. Inflammation ontology design pattern: an exercise in building a core biomedical ontology with descriptions and situations. In Domenico Maria Pisanelli, editor, *Ontologies in Medicine*. IOS Press, Amsterdam, 2004.
17. Aldo Gangemi, Frehiwot Fisseha, Johannes Keizer, Jos Lehmann, Anita Liang, Ian Pettman, Margherita Sini, and Marc Taconet. A Core Ontology of Fishery and its Use in the FOS Project. In A. Gangemi and S. Borgo, editors, *Proceedings of the EKAW*04 Workshop on Core Ontologies in Ontology Engineering*, volume 118. CEUR-WS, 2004.
18. Aldo Gangemi, Jos Lehmann, Valentina Presutti, Malvina Nissim, and Carola Catenacci. C-ODO: an OWL meta-model for collaborative ontology design. In Harith Alani, Natasha Noy, Gerd Stumme, Peter Mika, York Sure, and Denny Vrandecic, editors, *Workshop on Social and Collaborative Construction of Structured Knowledge (CKC 2007) at WWW 2007*, Banff, Canada, 2007.
19. Aldo Gangemi, Domenico Maria Pisanelli, and Geri Steve. An ontological framework to represent norm dynamics. In R. Winkels, editor, *Proceedings of the 2001 Jurix Conference, Workshop on Legal Ontologies*, Amsterdam, 2001.
20. Aldo Gangemi and Valentina Presutti. Ontology design for interaction in a reasonable enterprise. In Peter Rittgen, editor, *Handbook of Ontologies for Business Interaction*. IGI Global, Hershey, PA, November 2007.

21. J.-M. Le Goff and I. Willers. Design patterns for description-driven systems, 1999.
22. T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
23. M. Gruninger and M. Fox. The role of competency questions in enterprise engineering, 1994.
24. Giancarlo Guizzardi and Gerd Wagner. A unified foundational ontology and some applications of it in business modeling. In *CAiSE Workshops (3)*, pages 129–143, 2004.
25. Peter Haase, Saartje Brockmans, Raul Palma, Jerome Euzenat, and Mathieu d'Aquin. D1.1.2 updated version of the networked ontology model. Deliverable D1.1.2, Neon Project, 2007.
26. Neil B. Harrison, Paris Avgeriou, and Uwe Zdlin. Using patterns to capture architectural decisions. *Software*, 24(4):38–45, 2007.
27. David C. Hay. *Data Model Patterns.* Dorset House Publishing, 1996.
28. Tzung-Pei Hong, Wen-Chang Chang, and Jiann-Horng Lin. A two-phased ontology selection approach for semantic web. In Rajiv Khosla, Robert J. Howlett, and Lakhmi C. Jain, editors, *KES (4)*, volume 3684 of *Lecture Notes in Computer Science*, pages 403–409. Springer, 2005.
29. David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using dpml. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 3–11, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.
30. David Maplesden, John G. Hosking, and John C. Grundy. A visual language for design pattern modelling and instantiation. In *HCC*, pages 338–339. IEEE Computer Society, 2001.
31. Alistar Miles and Dan Brickley. SKOS Core Vocabulary Specification. Technical report, World Wide Web Consortium (W3C), November 2005. http://www.w3.org/TR/2005/WD-swbp-skos-core-spec-20051102/.
32. E. Motta and W. Lu. A library of components for classification problem solving. ibrow project ist-1999-19005: An intelligent brokering service for knowledge-component reuse on the world- wide web. Technical report, KMI, 2000.
33. Natasha Noy and Alan Rector. Defining N-ary Relations on the Semantic Web: Use With Individuals. Technical report, W3C, 2005. http://www.w3.org/TR/swbp-n-aryRelations/ (2004).
34. Daniel Oberle, Peter Mika, Aldo Gangemi, and Marta Sabou. Foundations for service ontologies: Aligning OWL-S to DOLCE. In *Proceedings of the World Wide Web Conference (WWW2004)*, volume Semantic Web Track, 2004.
35. Object Management Group (OMG). Unified modeling language specification: Version 2, revised final adopted specification (ptc/04-10-02), 2004.
36. Ontology design patterns. http://www.ontologydesignpatterns.org.
37. Valentina Presutti, Aldo Gangemi, Stefano David, Guadalupe Aguado de Cea, Mari-Carmen Suarez Figueroa, Elena Montiel-Ponsoda, and María Poveda. Library of design patterns for collaborative development of networked ontologies. Deliverable D1.1.3, NeOn project, 2008.
38. Alan Rector and Jeremy Rogers. Patterns, properties and minimizing commitment: Reconstruction of the galen upper ontology in owl. In Aldo Gangemi and Stefano Borgo, editors, *Proceedings of the EKAW*04 Workshop on Core Ontologies in Ontology Engineering.* CEUR, 2004.

39. Jacqueline Rene Reich. Ontological design patterns: Metadata of molecular biological ontologies, information and knowledge. In Mohamed T. Ibrahim, Josef Küng, and Norman Revell, editors, *DEXA*, volume 1873 of *Lecture Notes in Computer Science*, pages 698–709. Springer, 2000.
40. Marta Sabou, Sofia Angeletou, Mathieu dAquin, Jesus Barrasa, Klaas Dellschaft, Aldo Gangemi, Jos Lehmann, Holger Lewen, Diana Maynard, Dunja Mladenic, Malvina Nissim, WimPeters, Valentina Presutti, and BorisVillazon. Methods for selection and integration of reusable components from formal or informal user specifications. Deliverable D2.2.1, NeOn project, 2007.
41. Marta Sabou, Vanessa Lopez, and Enrico Motta. Ontology selection for the real semantic web: How to cover the queen's birthday dinner? In Steffen Staab and Vojtech Svatek, editors, *EKAW*, volume 4248 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2006.
42. Jorge Santos and Steffen Staab. Fonte: factorizing ontology engineering complexity. In *K-CAP '03: Proceedings of the 2nd international conference on Knowledge capture*, pages 146–153, New York, NY, USA, 2003. ACM.
43. Stefan Schulz, Anand Kumar, and Thomas Bittner. Biomedical ontologies: what part-of is and isn't. *J. of Biomedical Informatics*, 39(3):350–361, 2006.
44. Dmitri Soshnikov. Ontological design patterns for distributed frame hierarchy. In *Proceedings of the 5th International Workshop on Computer Science and Information Technologies*, Ufa, Russia, 2003.
45. S. Staab, M. Erdmann, and A. Maedche. Engineering ontologies using semantic patterns, 2001.
46. Robert Stevens, na Aranguren Mikel Ega Katy Wolstencroft, Ulrike Sattler, Nick Drummond, Matthew Horridge, and Alan Rector. Using owl to model biological knowledge. *Int. J. Hum.-Comput. Stud.*, 65(7):583–594, 2007.
47. Mari Carmen Suarez-Figueroa, Saartje Brockmans, Aldo Gangemi, Asuncion Gomez-Perez, Jos Lehmann, Holger Lewen, Valentina Presutti, and Marta Sabou. Neon modelling components. Deliverable D5.1.1, NeOn project, 2007.
48. Vojtech Svatek. Design patterns for semantic web ontologies: Motivation and discussion. In *Proceedings of the 7th Conference on Business Information Systems*, Poznan, 2004.
49. W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.
50. Denny Vrandecic and Aldo Gangemi. Unit tests for ontologies. In Mustafa Jarrar, Claude Ostyn, Werner Ceusters, and Andreas Persidis, editors, *Proceedings of the 1st International Workshop on Ontology content and evaluation in Enterprise*, LNCS, Montpellier, France, OCT 2006. Springer.
51. Denny Vrandecic and York Sure. How to design better ontology metrics. In Wolfgang May and Michael Kifer, editors, *Proceedings of the 4th European Semantic Web Conference (ESWC'07)*, Innsbruck, Austria, JUN 2007. Springer. to appear.
52. Denny Vrandecic, York Sure, Raul Palma, and Francisco Santana. Ontology repository and content evaluation. Deliverable D1.2.10v2, KnowledgeWeb project, 2007.

Index