Chapter 3
Pattern-Based Ontology Design

Valentina Presutti, Eva Blomqvist, Enrico Daga, and Aldo Gangemi

Abstract In this chapter, we present ontology design patterns (ODPs), which are reusable modeling solutions that encode modeling best practices. ODPs are the main tool for performing pattern-based design of ontologies, which is an approach to ontology development that emphasizes reuse and promotes the development of a common "language" for sharing knowledge about ontology design best practices. We put specific focus on content ODPs (CPs) and show how they can be used within a particular methodology. CPs are domain-dependent patterns, the requirements of which are expressed by means of competency questions, contextual statements, and reasoning requirements. The eXtreme Design (XD) methodology is an iterative and incremental process, which is characterized by a test-driven and collaborative development approach. In this chapter, we exemplify the XD methodology for the specific case of CP reuse. The XD methodology is also supported by a set of software components named XD Tools, compatible with the NeOn Toolkit, which assist users in the process of pattern-based design.

3.1 Introduction

One of the most challenging and neglected areas of ontology design is reusability, which is getting more and more important partly due to the increased spread of the Linked Data concept (Bizer et al. 2009). The notion of "pattern" has proved useful in design, as exemplified in diverse areas, such as software engineering (Gamma et al. 1994). In this chapter, we introduce the notion of ontology design patterns (ODPs) along with a description of their different types and characteristics.

V. Presutti (✉) • E. Blomqvist • E. Daga • A. Gangemi
Semantic Technologies Lab, Institute of Cognitive Sciences and Technologies (National Research Council – CNR), Via Nomentana 56, 00161 Rome, Italy
e-mail: valentina.presutti@cnr.it; eva.blomqvist@istc.cnr.it; enrico.daga@cnr.it; aldo.gangemi@cnr.it

M.C. Suárez-Figueroa et al. (eds.), *Ontology Engineering in a Networked World*,
DOI 10.1007/978-3-642-24794-1_3,

Then we focus on content ODPs (CPs), which are domain-dependent practices of modeling, encoded as reusable computational components.

ODPs have recently been the subject of a series of workshops (Blomqvist et al. 2009b, 2010b), and they are collected in online repositories such as the ODP portal[1]. Section 3.2 defines and describes ODPs, their types and characteristics, while in Sect. 3.3, we describe how CPs can be reused by means of a set of operations, such as import, specialization, and composition. In the second part of the chapter, i.e., Sect. 3.4, we introduce a pattern-based ontology design approach and describe a particular iterative and incremental method named eXtreme Design (XD), supporting this practice with a collaborative and test-driven approach. At the end of Sect. 3.4, we show a set of tools that provide software support to XD in the NeOn Toolkit environment, before we summarize some conclusions in Sect. 3.5.

3.2 What Are Ontology Design Patterns (ODPs)?

During the past decade, as remarked by (Gangemi and Presutti 2009), an average user that is trying to build or reuse an ontology, or an existing knowledge resource, has typically been left with very 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, for instance, a large set of web ontologies that are evaluated (usually in an implicit way, e.g., by inspecting them) against the intended domain and tasks of the ontology that is needed. The selected ontology (if any) is reused, and then an adaptation process is started in order to cope with the implicit requirements underlying the ontology project that originally created the reused ontology[2]. This scenario is costly in many cases. As noted by (Rector and Stevens 2008), usability of large OWL ontologies from a human perspective is often low, and automatic selection mechanisms do not help with the adaptation process.

Another typical scenario 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 can be higher than developing a new ontology from scratch. On the other hand, the success of very simple and small ontologies, such as FOAF[3] and SKOS (Miles and Bechhofer 2009), shows the potential of

[1] http://www.ontologydesignpatterns.org

[2] Even in cases when ontology requirements are explicitly expressed, e.g., as described in Chap. 5, there are commonly other implicit domain assumptions that need to be addressed at reuse time. In our experience, it is also quite rare that explicit requirements are distributed together with their corresponding ontology.

[3] See the FOAF project website: http://www.foaf-project.org/

really portable or "sustainable" ontologies. These lessons learned support a 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 has been experienced in software engineering), it is suggested to support reusability on the design side specifically. We need a way to express commonly applicable solutions and "best practices" and what ontological requirements they solve (see Chap. 5); this is where ODPs come into play. An ODP is a modeling solution to a recurrent ontology design problem (Gangemi and Presutti 2009). However, with the term ODP we refer to a wide range of modeling solution types. ODPs can be grouped into six types, or families, each addressing different kinds of modeling problems:

Structural ODPs include *Logical ODPs* and *Architectural ODPs*. *Logical ODPs* are compositions of logical constructs that solve a problem of expressivity. They help solving design problems when the used representation language does not directly support certain logical constructs, such as representing *n*-ary relations in OWL (Noy and Rector 2004). *Architectural ODPs* are defined in terms of compositions of *Logical ODPs* and affect the overall shape of the ontology, e.g., a certain OWL 2 profile could be viewed as an *Architectural ODP*.

Correspondence ODPs include *Re-engineering ODPs* and *Alignment ODPs*. *Re-engineering ODPs* provide designers with solutions to the problem of transforming a conceptual model, which can be either an ontology or a non-ontological resource[4]. to an ontology, e.g., transforming an OWL ontology in order to make it comply with a certain vocabulary, transforming a classification scheme to an OWL ontology, and so on. *Alignment ODPs* are patterns for creating semantic associations between two existing ontologies. They provide designers with solutions to align two ontologies without changing the logical types of the ontology entities involved, e.g., relating two ontologies both defining the concept "author," one by a class and the other by a property (Scharffe and Fensel 2008).

Reasoning ODPs are procedures that perform automatic inference. Examples of *Reasoning ODPs* are so-called normalizations (Vrandečić and Sure 2007). Other *Reasoning ODPs* include common reasoning tasks, such as *classification*, *subsumption*, *inheritance*, *materialization*, *de-anonymizing*, etc.

Presentation ODPs deal with usability and readability of ontologies from a human perspective. They are best practices facilitating ontology evaluation and selection, hence supporting reuse. Examples of *Presentation ODPs* are so-called *Naming ODPs,* which identify best practices for naming, i.e., naming conventions (Svátek et al. 2009).

Lexico-Syntactic ODPs are linguistic structures consisting of a sequence of types of words associated with an assessment of the meaning they express (Aguado de Cea et al. 2009). For example, the sequence of two noun phrases connected by

[4] For further details, and a definition of "non-ontological resource", see Chap. 6 of this book.

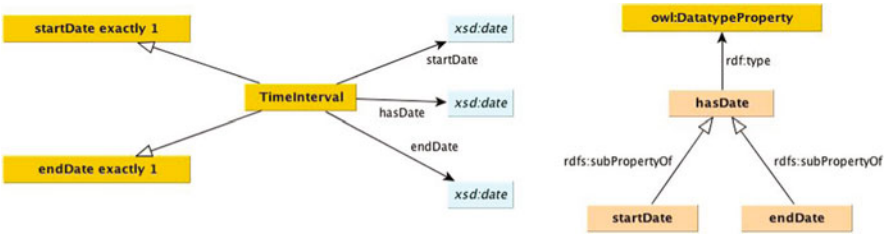

Fig. 3.1 UML-like diagram showing the OWL encoding of the *time interval* CP taken from the online catalog of CPs (http://www.ontologydesignpatterns.org/wiki/Submissions:TimeInterval)

the verb *be* such as "Dolphins *are* warm blooded mammals" often identifies a "subClassOf" relation between a class that represents dolphins and a class that represents warm-blooded mammals.

Content (or domain) ODPs (CPs) are instantiations (and compositions) of *Logical ODPs*. They have an explicit non-logical vocabulary for a specific domain of interest, i.e., they are content (domain) dependent, although the domain might be very general. An example of a Content ODP is depicted in Fig. 3.1. It represents the concept of a "time interval" as a class of things characterized by an arbitrary number of "dates" (i.e., points in time), but which has exactly one start date and one end date.

Much more important than the type of a pattern is its nature of being a reusable modeling solution. Ideally, an ontology project can be completely developed by reusing existing solutions, i.e., ODPs, by appropriately combining them – however, this ideal situation will most likely be very rare in practice, whereby we need to combine the approach presented here with the ones targeted in other chapters of this book. An interesting question is nevertheless: How can we make ODP reuse as easy and useful as possible? In order to identify candidate ODPs for reuse in a certain ontology project, ODPs and the specific ontology design problems to be addressed have to be comparable, i.e., need to be described in a similar way. For instance, we need to know what requirements a certain ODP helps us to solve, as well as what requirements are present in our current ontology design project.

Figure 3.2 depicts the idea of pattern-based design. The ontology development project is divided into two spaces: (1) the problem space, which contains a set of requirements (explicitly represented, e.g., as competency questions (CQs), as discussed in Chap. 5) that describe the ontology design problems to be addressed, and (2) the solution space, which contains all available ODPs, where each ODP should be well documented, e.g., described through what ontological requirements it solves. First, the requirements of the current project (problem space) are compared with the requirements of the available ODPs (solution space), and a set of candidate matching ODPs are identified. Second, the most appropriate ODPs among the candidate ones are selected for reuse, as illustrated by "dropping them into the project basket" in Fig. 3.2.

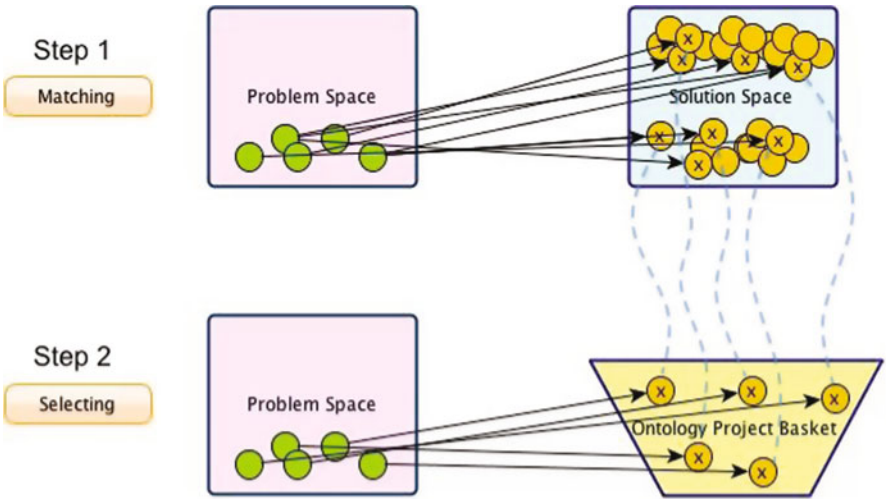

Fig. 3.2 The idea of pattern-based design. The ontology project is divided into the *problem* and *solution spaces*. The *problem space* contains a set of requirements (see Chap. 5), while the *solution space* contains a set of ODPs. The two spaces are compared in order to identify ODPs matching the requirements (matches are illustrated by *arrows*). A number of those ODPs are selected for reuse in the ontology project, i.e., dropped in the project basket

One of the most important raison d'être of ODPs is to enable this matching and selection task for supporting reuse. Hence, regardless of its type, an ODP is associated with a set of requirements, explicitly represented, which describes the problem it provides a solution for. For example, the requirements associated with the CP shown in Fig. 3.1 refer to the problem of representing time intervals, their start and end points. If expressed in the form of CQs, the requirements of that CP include "What is the starting point of a particular interval?" and "What is the end point of a particular interval?". Another example is the Logical ODP for expressing "*n*-ary relations," the requirements of which indicate the issue of representing relations with *n* arguments through a logical language including only primitives for expressing binary relations, e.g., a language such as OWL. The format and extent of formality of requirement representation depends on the type of ODP. In this chapter, we will focus on content ODPs (CPs) and how to use them for developing ontologies through the application of a particular pattern-based method, as well as its specific tool support.

3.3 Content Ontology Design Patterns (CPs)

CPs solve design problems for the domain classes and properties that populate an ontology; therefore, they solve content – domain-specific – problems (Gangemi and Presutti 2009). According to the general notion of ODP (see Sect. 3.2), each CP is

associated with a set of requirements, which represents the problem it provides a solution for. Such requirements are usually represented in three different forms: (1) competency questions (CQs), i.e., based on work by (Gruninger and Fox 1994), (2) contextual statements, i.e., general axioms that hold within the domain, and (3) reasoning requirements. CQs indicate typical queries that a knowledge base will be able to answer if it is based on that CP. Contextual statements are general axioms that apply within the domain, which indicate conditions that hold for (and between) certain concepts encoded by the CP. Finally, reasoning requirements indicate what inferences are enabled by the CP, e.g., if it perhaps allows some form of classification or consistency checking of facts.

The *time interval* CP shown earlier, in Fig. 3.1 (Sect. 3.2), which is a very simple but useful CP, is associated with the following competency questions:

- *When does a certain time interval start?*
- *When does a certain time interval end?*
- *What are the points in time that belong to a certain time interval?*

Typically, a CP can also be associated with a set of SPARQL queries that formally encode its competency questions. This is very useful for an ontology designer who wants to test an ontology containing a CP against sample data since the SPARQL queries can be used to test the coverage of the CQs.

The *time interval* CP (see Fig. 3.1, Sect. 3.2) is also associated with the following contextual statement:

- *A time interval always has exactly one starting point and exactly one end point.*

This requirement is, in the OWL realization of the CP, addressed by cardinality restrictions on the data type properties that identify the start and the end of a time interval.

Finally, this CP is also associated with the following reasoning requirements:

- *The start and end dates of a time interval belong to the interval.*
- *Two time intervals with the same start and end dates should be recognized to be the same interval.*

The first reasoning requirement is, in the OWL realization of the CP, addressed by defining the two data type properties `startDate` and `endDate` as sub-properties of `hasDate`, which is the property indicating that something belongs to the interval. This enables ontologies reusing the CP to also include the start and end date of an interval, when the model is queried for dates belonging to the interval, using the `hasDate` property. The second reasoning requirement is addressed by defining a `hasKey[startDate, endDate]` axiom, on the class `TimeInterval`. This enables an inference engine to infer that the `owl:sameAs` property holds between two instances of `TimeInterval` whenever they have the same start and end date values.

Where do CPs come from? This is a highly relevant question since we need a considerable catalog of CPs in order for them to be useful in practice. A CP can emerge from existing conceptual models as well as from data. It can be extracted

from *foundational* (Masolo et al. 2005), *core* (Gangemi and Borgo 2004), or *domain* ontologies, re-engineered from other conceptual models (e.g., data model patterns (Hay 2000)). Informally, the distinction between foundational, core, and domain ontologies relates to the generality of the domain they address and to the extent of domain coverage: (1) *foundational ontologies*, e.g., DOLCE[5] and SUMO (Niles and Pease 2001), axiomatize general concepts and relations and are reusable across any domain; (2) *core ontologies* (Masolo et al. 2005), such as the Core Ontology of Fishery (Gangemi et al. 2004) and the Core Legal Ontology (Gangemi et al. 2005), focus on a specific domain without being restricted to specific applications or specific sub-areas. The latter can be built as extensions of foundational ontologies or based on general principles and well-founded methodologies; and (3) *domain ontologies*, such as the Gene ontology[6] and the Unified Medical Language System (UMLS)[7], deal extensively with a specific domain of interest, deepen the coverage of a certain area of a domain or address a specific use case within a domain. Informally, such general ontologies can be viewed as compositions of numerous CPs, and by modularizing such ontologies, i.e., decoupling certain "pieces" from the rest of their often large overall structure, the formal representation of those CPs can be extracted.

CPs can also be extracted from Linked Data (Bizer et al. 2009), i.e., in a more bottom-up fashion, where they emerge from the way data is actually modeled. By analyzing recurring semantic structures (if any) within the same, as well as across different, datasets addressing some domain of interest, CPs may be detected.

CPs can also be created by composing or specializing other CPs or by expanding them (see Sect. 3.3.1 describing operations on CPs). Figure 3.3 shows a composed CP. This CP allows to represent the relation between an information object, such as a novel, and its realizations, e.g., a book, an HTML page, etc., and to index this relation based on the place *where* and time *when* it holds. This CP reuses the *time interval* CP shown in Fig. 3.1 (Sect. 3.2) and combines it with two other CPs, the *information realization* CP[8] and the *place* CP[9]. The composition is realized by specializing a fourth CP, the *situation* CP[10]. which is a CP corresponding to the *n-ary relation* Logical ODP.

Since CPs can be represented as reusable building blocks, e.g., OWL modules, a natural question is how they are distinguished from any other small ontology. CPs show a number of pragmatic characteristics that allow to distinguish them from other ontologies. CPs are:

[5] DOLCE – Project Home Page: http://dolce.semanticweb.org

[6] See http://www.geneontology.org/

[7] See http://www.nlm.nih.gov/research/umls/

[8] http://ontologydesignpatterns.org/wiki/Submissions:Information_realization

[9] http://ontologydesignpatterns.org/wiki/Submissions:Place

[10] http://ontologydesignpatterns.org/wiki/Submissions:Situation

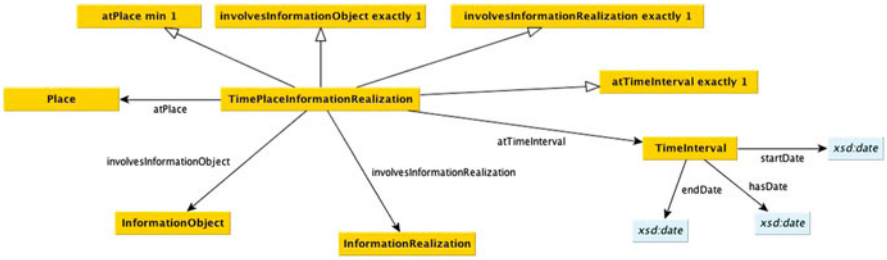

Fig. 3.3 UML-like diagram showing the OWL encoding of the *time and place indexed information realization* CP present in the online catalog of CPs

- *Computational components*. They are represented and encoded in a computational logic language, e.g., OWL, so that they can be processed and reused as building blocks in ontology design, e.g., through the OWL import construct.
- *Small, autonomous components*. Smallness and autonomy of CPs facilitate the design of ontology networks, because they enforce modularization; by composing CPs, designers can better govern the complexity of the whole resulting ontology as opposed to governing a monolithic ontology.
- *Inference enabling components*. Each CP allows some logical conclusions to be drawn from the model. This means that a single element, e.g., a single class without any associated axioms, cannot be a CP since it does not enable any inferences (even simple ones) to be made.
- *Hierarchical components*. All CPs participate in a partial order, where the ordering relation is called *specialization* (see Sect. 3.3.1). Specialization requires that at least one entity of the more specific pattern, e.g., a class or property, is subsumed by at least one entity of another, more general, pattern. Figure 3.4 shows an example of CP specialization: (a) shows the *part-of* CP[11], which defines a transitive property between objects for representing parthood relationships between them; (b) shows the *componency* CP[12], which specializes *part-of* by defining object properties for representing direct parts of objects as sub-properties of the transitive parthood relation.
- *Cognitively relevant components*. CP visualization must be intuitive and compact and should catch relevant, "core" notions of a domain[13].
- *Best practices* of ontology modeling. How to evaluate the quality of a CP, e.g., to determine if it is truly a *best practice* is currently an open issue (Hammar and Sandkuhl 2010); hence at the moment, the quality of a CP can only be assessed through the personal experience of ontology designers and through its provenance. Additional criteria are evidence from reusability across different projects and large-scale applications such as Linked Data.

[11] http://ontologydesignpatterns.org/wiki/Submissions:PartOf

[12] http://ontologydesignpatterns.org/wiki/Submissions:Componency

[13] Independently of the generality at which a CP is singled out, it must contain the central notions that "make rational thinking move" for an expert in a given domain for a given task.

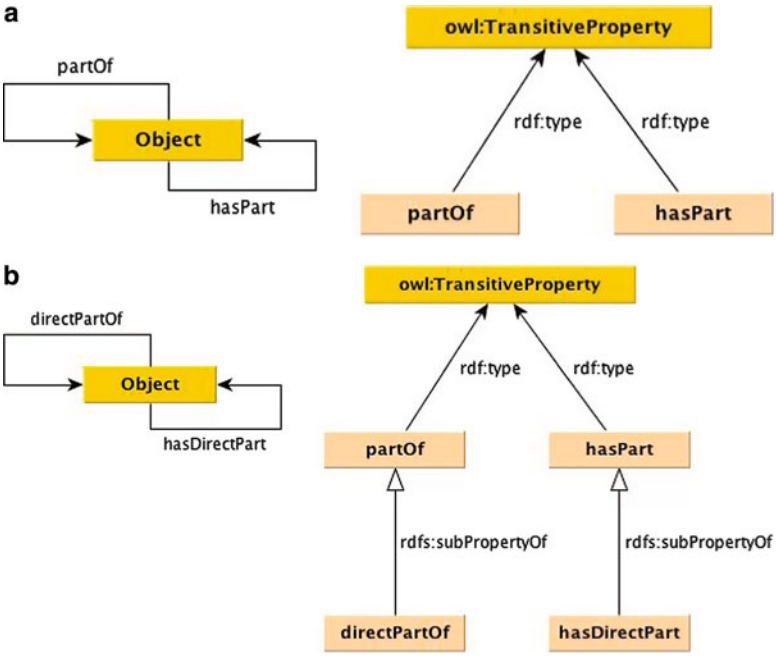

Fig. 3.4 Example of CP specialization: (**a**) depicts the *part-of* CP, which is specialized by the *componency* CP, shown in (**b**)

Additionally, CPs often match linguistic structures called frames. This could be formulated as an additional characteristic of being *linguistically relevant*, and the essence of most CPs can be expressed quite straightforward in natural language. The richest repository of frames is FrameNet (Baker et al. 1998). Informally, a frame is a lexically founded ontology design pattern. Frames typically encode argument structures for verbs, e.g., the frame *Desiring* defines associations between elements (or "semantic roles") such as `Experiencer`, `Event`, `FocalParticipant`, `LocationOfEvent`, etc. Frames can be used for validating CPs with respect to lexical coverage, for lexicalizing them, and can be re-engineered in order to populate a CP catalog such as the ODP portal.

As opposed to the concept of CP, there is that of *AntiCP*. AntiCPs are ontologies that implement wrong modeling practices, e.g., examples of bad practices or common mistakes. In other words, they are based on erroneous assumptions or rationales. For example, modeling transitive parthood relationships through subsumption, e.g., `City rdfs:subClassOf Country`, is considered an AntiCP. AntiCPs produce the side effect of inferring wrong or undesired knowledge, e.g., `Rome rdf:type Country`, or of preventing the capability to infer the desired knowledge. It is important to distinguish between ontologies that are not CPs and AntiCPs, i.e., only a subset of ontologies that are *not* CPs are *AntiCPs*.

3.3.1 Operations on Content Patterns

CPs are a special kind of ontologies, as discussed above, and their creation and usage rely on a set of operations that can be summarized as follows:

Cloning consists of duplicating an ontology entity (possibly into a new namespace), which can be reused in a CP or used as a prototype for the definition of a new ontology entity defined in a CP. This operation is, for instance, used when extracting CPs from foundational and core ontologies, i.e., a part of the larger structure becomes a CP through being cloned and given a new namespace.

Composition relates two CPs and results in a new ontology (which could in turn be a CP – as seen in Fig. 3.3, Sect. 3.3). The resulting ontology includes the union of the sets of ontology entities and axioms from the two CPs plus the ontology entities and axioms that are defined locally in the new ontology in order to relate the two CPs, e.g., disjointness axioms. Figure 3.5 depicts an example of a CP composition. At the left of the figure, the two CPs are shown separately, one CP (top left) represents membership relationships between persons and music bands, the other CP (bottom left) models objects and the roles they play. The axioms that are added for composing the two CPs are shown at the right side of the figure. The class `Person` is defined to be subclass of `Object`, and both `Person` and `Band` are defined to be disjoint with `Role`.

Specialization defines a new ontology (which could in turn be a CP) by specializing entities of an existing one, e.g., a CP. Specialization introduces a partial order between CPs, based on subsumption relations holding between their respective ontological entities. Specialization relies on `rdfs:subClassOf` and `owl:subPropertyOf` constructs. Figure 3.4 in Sect. 3.3, as mentioned previously, shows an example of specialization: the *component* CP (Fig. 3.4b) specializes *part-of* (Fig. 3.4a) by defining the object properties `directPart` and `directPartOf` as sub-properties of `hasPart` and `partOf`, respectively.

Import is the basic mechanism for explicit CP reuse, as well as a way to reuse ontologies in general. It is also the only operation described here that is directly

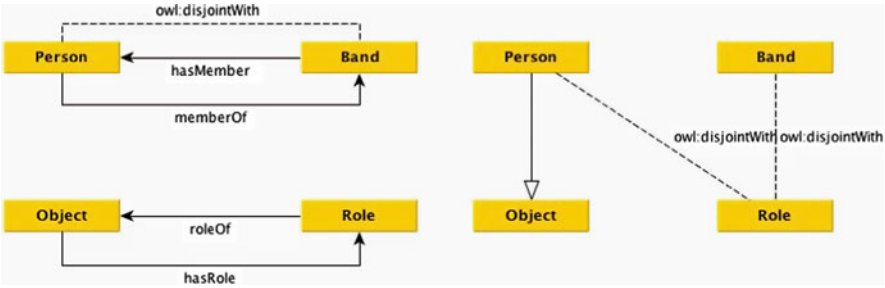

Fig. 3.5 Example of CP composition: A CP representing membership relationships between persons and bands (*top left*) is composed with the CP *object-role* (*bottom left*), which represents objects and the roles they play, and the result can be seen to the *right*

supported in the OWL vocabulary, through `owl:imports`. By importing a CP, the ontology includes all the axioms of the CP and hence ensures the set of inferences that the CP enables in its corresponding knowledge base.

3.4 eXtreme Design: An Agile Methodology for Pattern-Based Ontology Development

With the name *eXtreme Design* (XD), we refer to a family of methods that support the pattern-based design approach as depicted in Fig. 3.2 (Sect. 3.2), i.e., the matching between problem and solution spaces in order to reuse solution components, such as ODPs (Presutti et al. 2009). When XD is based on CPs, the problem space is expressed by means of *competency questions* (CQs), *contextual statements,* and *reasoning requirements*, as described at the beginning of Sect. 3.3, and the solution space contains CPs and their associated requirements, i.e., similarly expressed through CQs, contextual statements, and reasoning requirements. Hence, the matching process is performed through finding similarities between CQs as well as between the other two types of requirements, as exemplified in Fig. 3.6.

In the following sections, we describe the XD method, inspired by software engineering's *eXtreme Programming* (XP) (Shore and Warden 2007) and *experience factory* (Basili et al. 1994), for building ontologies through intensive CP reuse. XD is test-driven, i.e., testing is a central part of the development; it applies a divide-and-conquer approach similarly to XP and promotes pair design, which is analogous to pair programming.

3.4.1 eXtreme Design Principles

Similarly to XP, XD has evolved around a set of main principles. The principles both describe the essence of XD and act as guidelines when performing the design process.

The first principle is named *customer involvement and feedback*. Ideally, the customer should be involved in the ontology development team continuously.

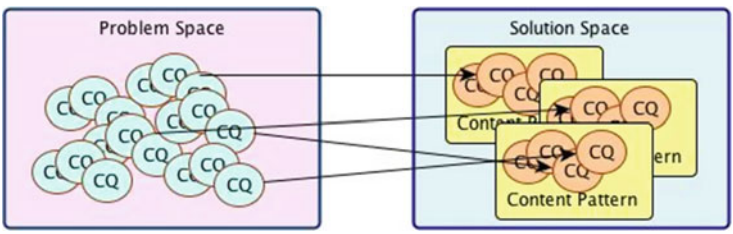

Fig. 3.6 The XD (eXtreme Design) approach with CPs, exemplified for CQ matching

This means that the customer should identify representatives that can be easily contacted during the development for quick feedback. Such representatives have to be aware of all parts, and needs, of the project. Here, depending on the project configuration, the "customer" could be either the organization containing the end users of the system to be built (including the ontology) or simply the software developers needing the ontology in order to perform some particular functionality in the overall system.

The second principle states that all requirements should be based on *customer stories*, from which CQs, contextual statements, and reasoning requirements are derived. The customer representatives describe the ontology requirements and the ontology tasks in terms of small stories. Designers work on those small stories and transform them into more rigorous and precise requirements, e.g., in the form of CQs, contextual statements, and reasoning requirements.

Next, an important principle is that of *iterative development*. XD is an iterative and incremental process. Each iteration produces a number of modules that contribute to an incremental release, produced through an integration phase.

Test-driven design means, in the case of XD, that testing is used as an integrated means for completing the modules. Stories, CQs, reasoning requirements, and contextual statements are used in order to develop unit tests, e.g., CQs can be transformed into SPARQL queries. By deciding how the query should be formed, a developer is actually partly designing the model, hence, the notion "test-driven" design. The ontology module representing a customer story can be passed to the integration phase, i.e., to be included in the next release, only if all its associated unit tests run successfully. This principle also enforces the *task-oriented* approach of the method, i.e., the principle that modules should realize exactly what is required (their intended task), nothing more and nothing less.

It has to be noted that ontology unit testing, first introduced by (Vrandečić and Gangemi 2006), has a different meaning than software unit testing. An ontology module developed for addressing (part of) a user story is tested by developing unit tests, i.e., dedicated ontology modules containing sample facts and appropriately documented with testing metadata, each importing the ontology module to be tested, based on one of the following three approaches: (1) through *verification tests* to test the fulfillment of basic requirements, i.e., SPARQL queries based on CQs that are run against valid sample data in order to check if expected results are returned by the SPARQL engine, (2) *inference tests*, i.e., through inference materialization performed on sample data which is expected to cause certain inferences to be materialized in accordance with the reasoning requirements and (3) through *stress tests*, e.g., through consistency checking performed on invalid sample data violating the contextual statements, thus expecting to provoke inconsistencies. While (1) and (2) are mainly intended for verifying the correct implementation of requirements, (3) could be viewed as more similar to the kind of software testing when a system is fed random or erroneous data, to make sure such cases are handled correctly, and there are no unexpected side effects or crashes.

One of the core principles of XD is *ODP reuse*, which inherently leads to a *modular design*. Iterations are based on identifying reusable CPs through matching

CQs and other requirements. Every time there is a positive match, the identified CPs are considered for reuse. If the solution space does not provide an adequate ready-to-use CP, a specific solution is developed in a modular way, preferably in the form of a new CP so that it can be shared with the team (and ideally on the web) for future reuse. This principle favors the creation of a common "language" based on shared patterns and eases both the understandability and the integration of developed modules. In addition, the *divide-and-conquer* paradigm leads to a natural modularisation of the problem, which facilitates a distributed ontology development, and assists in scoping the modeling issues that are addressed within a single iteration.

To handle this incremental ontology development, *collaboration* and *integration* are two essential principles. Integration is a key aspect of XD, as the ontology is developed in a distributed, modular, and collaborative way. Collaboration and continuous sharing of knowledge is needed when running an XD project. The result of each iteration, i.e., one or more ontology modules, is integrated with the rest of the ontology modules before releasing an increment. Typically, a sub-team of designers is devoted to the integration task.

As mentioned previously, *task-oriented design* is another main principle. The XD approach is based on developing a task-oriented ontology, covering only part of a domain of knowledge according to a specific application task. This is opposed to the more philosophically oriented approach of formal ontology design, where the aim is to comprehensively cover a certain domain of knowledge. XD proposes to provide solutions to the exact requirements stated, in the sense that the concepts should be defined according to the intended task of the ontology, rather than in some common sense notion of their "true" nature. Each XD iteration focuses on a specific part of the domain requirements, expressed in terms of a user story.

On the more organizational side, XD promotes *pair design*. The team of designers is organized in pairs. This practice is analogous to the pair programming of XP. While pair programming has empirically been proven efficient in software development, it still remains to rigorously test the efficiency for ontology engineering. Currently, this has to be considered a hypothesis, based on experience and observations made through collecting feedback of trainees and developers, through informal discussions and questionnaires after the execution of XD with different teams. Most of them felt that they benefit from on-the-fly brainstorming, and perceived to improve the effect of learning-by-doing within the pair design setting.

3.4.2 The eXtreme Design Process

Figure 3.7 shows the complete XD process for CP reuse. The process starts with the XD team, including the customer representatives, making themselves familiar with the knowledge domain, with the aim of identifying the scope of the ontology project, based on the desired application tasks (*Step 1. Project initiation and scoping*). The objective is twofold: (1) to make the customer representatives

(domain experts) aware of what the XD team expects from them and (2) to provide the ontology design team with an overview of the problem from a domain expert perspective, its scope, and agree on initial terminology. The result of this step is the setup of a collaborative environment where the customer representatives and ontology designers will share documentation and collect argumentation and motivations of modeling issues, including terminology, e.g., through deploying a wiki for the project. Following this starting activity, the XD team identifies the sources of CPs to be used during the ontology project (*Step 2. Identify CP catalogs*); however, such a set can be extended during the process.

The customer representatives are then invited to write stories, preferably from real, documented scenarios, which act as samples of the typical facts that should be stored in the resulting ontology, and exemplify how these facts are connected and used (*Step 3. Collecting requirement stories*). All stories are organized in terms of priority, and possible dependencies between them are identified and made explicit. Each story is described by means of a small card, like the one depicted in Table 3.1, which includes the unique title of the story, a list of other stories that it depends on, a description in natural language, i.e., the story itself, and a priority value. The customer can add stories during the whole project life cycle, depending on how formal contracts and other commitments are formulated. Nevertheless, if a new requirement emerges, new stories can be written.

Once a sufficient number of stories for starting the development have been collected, each pair of designers selects a story that will be the focus of their work for the next iteration (*Step 4. Eliciting requirements and constructing module(s) from CPs*). The selection is based on the experience and competencies of the design pair and on the priority of the story. A new wiki page for the story is created, and its content is set up based on the information reported on the card. By performing this task, a pair enters a development iteration (the dashed rectangle in Fig. 3.7, which is detailed in Fig. 3.8).

Once a story has been completely modeled, it is carefully documented and released internally for integration into the next release (*Step 5. Releasing module(s)*). This task constitutes the end of a story iteration for a pair, and the result is one or more ontology modules, i.e., small ontologies. Before releasing a module, it is important to make sure that all tests run successfully and that the module is well documented, both in the shared wiki as well as through annotations in the module itself. All ontological elements have to have appropriate labels, they have to be commented (as well as the module itself), and the module should be associated with a description of its purpose, the requirements it solves, and even links to the unit

Table 3.1 A requirement story card, here exemplified through a story from the fishery domain

Title	Tuna observation
Depends on	Exploitation values, tuna areas
Description	In 2004, the resource of species “tuna” in water area 24 was observed to be fully exploited in the tropical zone at pelagic depth
Priority	High

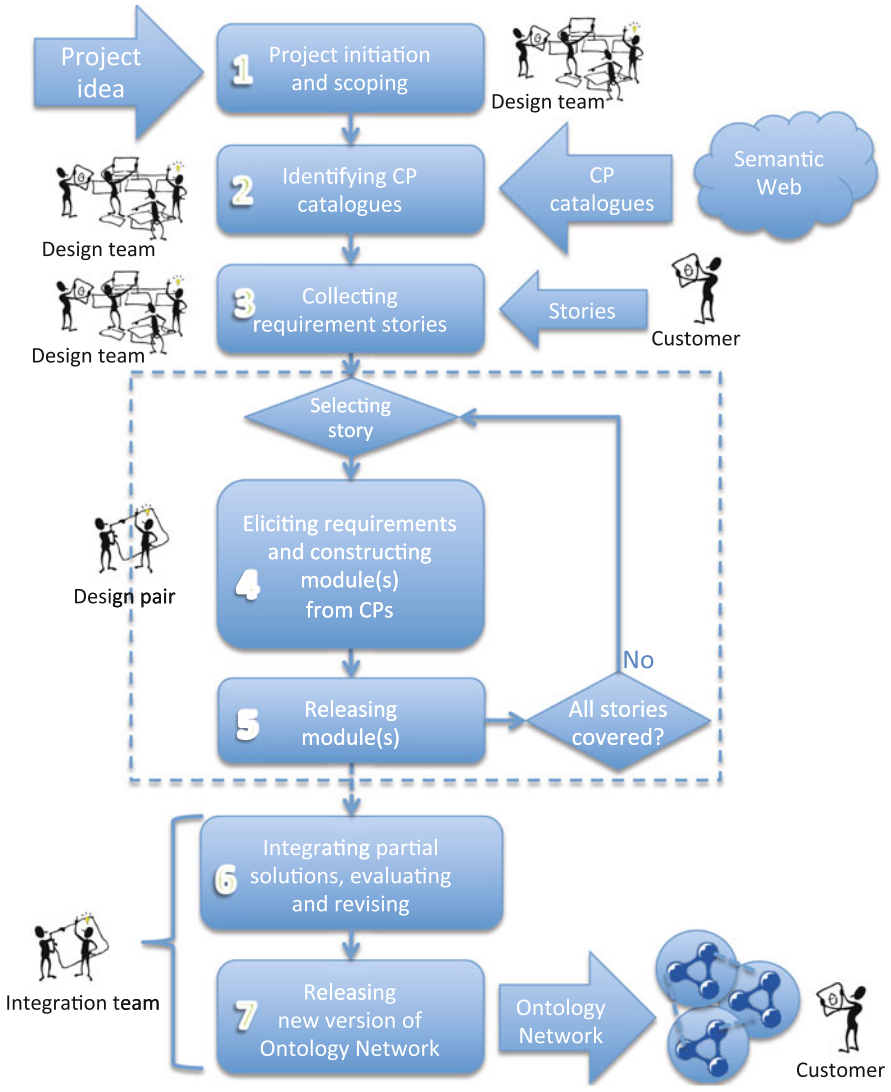

Fig. 3.7 The overall XD process, for CP reuse

tests that have been used. The modules are assigned a URI and are shared by the whole team. If a module can be publicly shared, and is considered highly reusable, it can be published in open web registries, such as the ODP portal.

Once a new module is released, it has to be integrated with all the others that constitute the current version of the ontology network (*Step 6. Integrating partial solutions, evaluating, and revising*). Usually, one pair is in charge of performing the integration and related tests. New unit tests are defined for the integrated ontology network, and all existing ones (unit tests of individual modules) are again executed

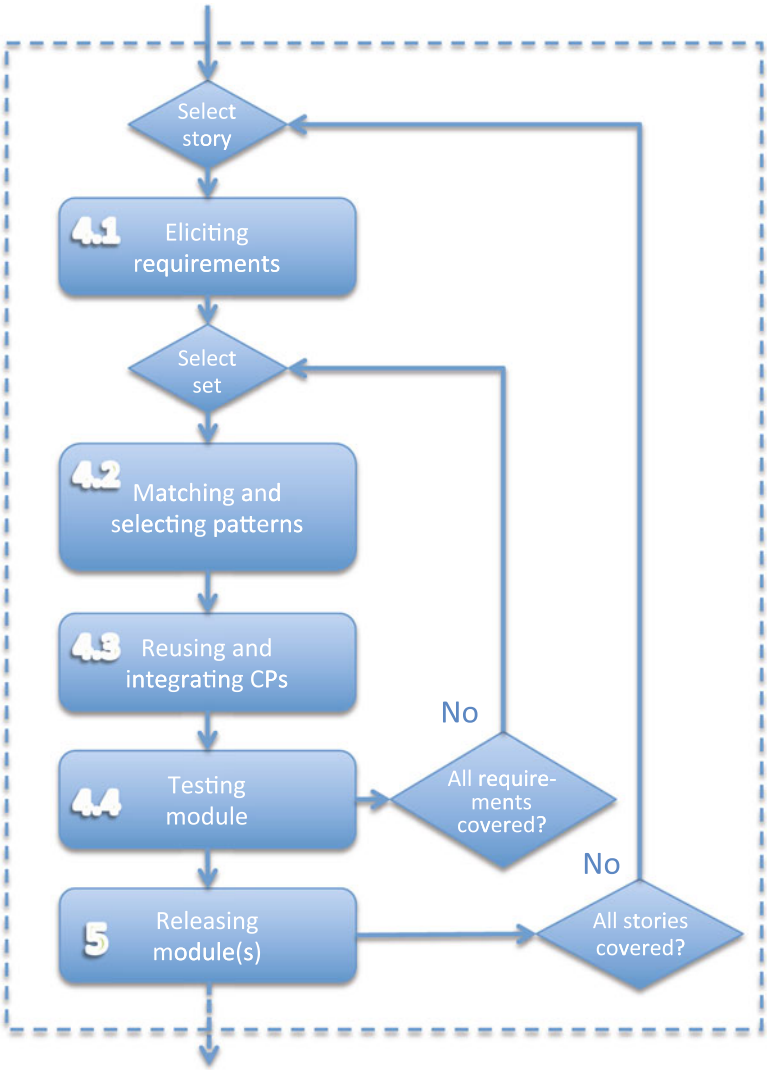

Fig. 3.8 Detailed breakdown of sub-steps in the design pair iteration (steps 4–5 in Fig. 3.7)

as regression tests before moving to next task. All contextual statements and reasoning requirements are taken into account, and all necessary alignment axioms are defined. The modules are now under the complete control and editing of the team in charge of the integration, and refactoring of the ontology modules may be performed in case inconsistent modeling choices are discovered. Integration can be done in multiple fashions, and an integration policy should be defined at the start of the project. For instance, if decoupling of modules is an essential feature of the resulting ontology network, then a minimum of refactoring should be performed in

order to remove overlap between modules, instead integration should simply align the modules. On the other hand, in some cases, a coherent and non-redundant model is desired, whereby an alternative policy would be to refactor the modules, remove as many redundant definitions as possible and instead add import dependencies between them. The products of this step are new unit tests and alignment axioms, and possibly a set of changes to the ontology modules (results of refactoring), all properly documented in the wiki.

When all unit tests run successfully during the integration step, a new incremental version of the ontology network can be released (*Step 7. Releasing new version of ontology network*). The ontology is given a new version number, it is appropriately documented, and it is associated with its own version of the wiki documentation. It is important to note that the process depicted in Fig. 3.7 is usually not a sequential one, i.e., in most cases the arrows indicate an input/output dependency rather than a sequence of actions. For instance, the integration and release steps will be ongoing activities during the complete project (as soon as the first modules are ready); hence, integration will be performed in parallel with Steps 4–5 where new modules are produced, to create a series of incremental releases.

Step 4 of the XD process identifies the core iteration performed by a design pair, which is focused on the development of the ontology module(s) representing one user story. Figure 3.8 depicts the main steps of a single iteration (i.e., a detailed breakdown of the steps within the dashed rectangle of Fig. 3.7).

First, the development pair analyzes the selected user story and derives a set of CQs, contextual statements and reasoning requirements from it (*Step 4.1. Eliciting requirements*). In order to do that, designers could involve the customer for having feedback and clarifications. For example, the story "Tuna observation" (see Table 3.1) can be transformed into the following CQs, which are added to the story's wiki page:

- CQ_1: What is the exploitation state observed and the vertical distance in a given climatic zone for a certain resource?
- CQ_2: What resources have been observed during a certain period in a certain water area?

Additionally, assume that the following contextual statements and reasoning requirements are derived based on a discussion with the customer representative:

- A resource contains one or more species.
- Species are associated to vertical distances. As a consequence, the vertical distance of a resource is inferred through the vertical distance of its species.

The iteration continues by further breaking down the task, before starting to address it through modeling. This is done by selecting one of the competency questions, or a small set of them that constitutes a coherent modeling issue, and then start matching them to the competency questions associated with available CPs in order to identify candidates for reuse (*Step 4.2. Matching and selecting patterns*). In our example, let CQ_1 and CQ_2 be the selected competency questions. Candidate CPs for reuse would be *situation* and *time interval*. The competency question of

situation – "What entities are in the setting of a certain situation?" – can be said to match the observation of the resource and the parameters that are in the setting of that observation. Additionally, the *time interval* CP may be seen as matching the question of what period a certain observation was made (CQ_2), although this could also be solved with just a simple data type property.

The following step is to select which of those patterns should actually be used for solving the modeling problem. In our case, there are only two patterns, and neither is an alternative solution to the other, but in many cases, this step involves making some modeling choices, i.e., deciding which pattern is most suitable for the particular case. Nevertheless, in our example, we still need to decide if both patterns are really needed or if they add too much overhead to our model. For instance, we may decide that *time interval* adds too many extra elements to the model since perhaps our customer simply wants to store the year of the observation, rather than an exact period of dates, in which case we will only select *situation*.

After selecting a set of CPs, it is time to start modeling, i.e., *reusing* the CPs (*Step 4.3. Reusing and integrating CPs*). The term "reuse" here refers to the application of typical operations that can be applied to CPs, i.e., *import*, *specialization* and *composition* (see Sect. 3.3.1). In some cases, one may also decide to clone a CP, e.g., if it is desirable not to rely on imports external to the project, which would result in replicating the modeling solution, but without importing the available building block. The latter has both advantages, e.g., reducing the size of the module in case the complete transitive closure of CP imports is not cloned, and disadvantages, e.g., the loss of a common "language" by not referring to the pattern explicitly and reduced support for automatic alignment with other pattern-based modules.

In our example, we import and specialize *situation* in order to address CQ_1, as shown in Fig. 3.9. Our particular situation is an "observation," and the thing observed is an "aquatic resource." Additionally, the exploitation state, climatic zone, and vertical distance of the observation are also involved in the setting. Thereby, we add a subclass of `situation:Situation` named `AquaticResourceObservation` and add the other entities as subclasses of `owl:Thing`. In addition, we define sub-properties of the `situation:isSettingFor` and

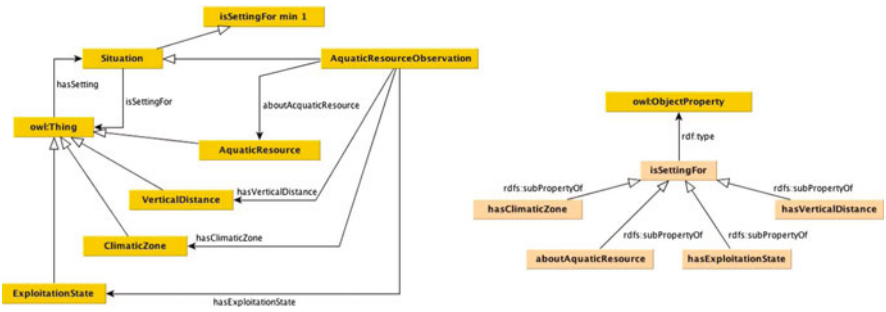

Fig. 3.9 Specialization of the situation CP for modeling aquatic resource observations, as for addressing CQ_1

its inverse, for connecting the observations to the resources and the different parameters. After iterating over all selected CPs (in our example, only one pattern was selected) and integrating them into the current module, the module also has to be extended to cover the complete set of CQs. In our example, no pattern was selected to solve the time period issue in CQ_2; hence, a data type property has to be added to the module in order to cover the complete CQ.

The goal of the following task (*Step 4.4. Testing module*) is to validate the ontology module against the requirements it is supposed to address, i.e., CQs, contextual statements, and reasoning requirements, through developing and executing verification tests, inference tests, and stress tests (see description of test types in Sect. 3.4.1). The ontology modules are revised until all unit tests run successfully. All unit tests are documented in the project wiki and are properly linked to their motivating user story, and requirement(s), in order to document the testing activity as well as to preserve the unit tests for the integration process. In our example, a unit verification test associated with CQ_2 could be the following SPARQL query, retrieving the exploitation state (?exp), vertical distance (?dist), climatic zone (?zone), and resources (?resource) of available observations (?obs):

```
SELECT ?exp ?dist ?resource ?zone
WHERE {

   ?obs a:AquaticResourceObservation.
   ?obs aboutAquaticResource ?resource.
   ?obs hasClimaticZone ?zone
   ?obs hasExploitationState ?exp.
   ?obs hasVerticalDistance ?dist

}
```

If all requirements derived from the story have been solved, and all tests run successfully, the design pair proceeds to internally release the module(s) (*Step 5. Releasing module(s)*), which are then ready for integration in the current overall increment iteration.

3.4.3 Example: A Music Industry Ontology

To illustrate the process of creating ontologies through XD, a small hypothetical project is described in this section. The domain, which is *music industry*, should be intuitive to most readers. The example is not intended as a case for validating the methodology, but merely as an illustration how it could be used in practice.

Step 1 – Project initiation and scoping. Let us assume that an ontology is needed in an online community platform for people who want to discuss music, share news, playlists, and music recommendations. The ontology will be used to store and retrieve information about music recordings and artists, as well as to reason

over musical genres. As ontology developers, we are working closely together with the software developers, implementing the online community software; however, we also have access to some future administrators of the community, who are used as the "customer representatives", i.e., domain experts, during the project. A wiki is set up for the development project, where all information is stored and shared, both between developers and with the customer representatives.

Step 2 – Identifying CP catalogs. Let us assume that we decide to focus on CP reuse and to mainly reuse CPs from the ODP portal.

Step 3 – Collecting requirement stories. As soon as the project environment is set up, we ask the customer representatives to start entering their stories into the wiki. Some stories become examples of typical information that is to be stored by the ontology, while other stories focus more on reasoning tasks of the ontology, depending on how the customer representatives formulate them. Each story is entered into the story template, e.g., given a title and priority, and as soon as several stories are in place, they can be related to each other. A collected story can be seen in Table 3.2, and another one in Table 3.3. The "Albums"-story depends on the "Recordings of songs" story, since the notion of recorded track appears also in the story about albums but is the main focus of "Recordings of songs."

Step 4 – Eliciting requirements and constructing modules from CPs. At this point, the design team is divided into pairs, in order to develop the ontology modules using pair design. One pair is dedicated to the integration task, i.e., proceed directly to prepare Steps 6–7, while the rest of the pairs choose their first stories from the pool of collected ones. Each story with high priority has to be solved before the lower priority ones are addressed. Each pair then starts to elicit requirements from their chosen story, i.e., tries to derive CQs, contextual statements, and reasoning requirements.

Table 3.2 A requirement story from the music domain concerning albums

Title	Albums
Depends on	Recordings of songs
Description	An album is a collection of recorded tracks. The genre(s) of an album should be derived based on the genres of the tracks it contains
Priority	High

Table 3.3 A requirement story from the music domain concerning songs and recordings

Title	Recordings of songs
Depends on	
Description	Songs are recorded by artists. Many artists can record the same song. In the web interface, users will click on songs and get to see the artists that have recorded them and links to information on those recordings
Priority	High

Let us imagine that you are part of the design pair who picks up the story in Table 3.2. First, you start analyzing the story itself, to see if there are any obvious CQs, indicating information that should be stored and retrieved. The most obvious CQs are:

1. What are the tracks of this album?
2. What is the genre of this track or album?

There could however be other CQs possible; hence, the final list needs to be agreed with a customer representative, in order to ensure appropriate coverage of the domain and task and to avoid misinterpretations of the story. In addition, it is evident from the way the story is written that a reasoning requirement is needed, i.e., the following:

- An album should be automatically assigned all the genres of it contained tracks.

In other cases, it may not be as self-evident what needs to be possible to infer; however, in many cases, software requirements and interactions with the customer can clarify such issues.

Additionally, contextual statements can be proposed based on common sense knowledge, e.g., in our case:

- An album always has at least one track.

Other contextual statements may be given by the customer representative or be implicit in the software requirements, e.g., limitations set by the way the software will use the ontology. We have thus collected two CQs, one contextual statement and one reasoning requirement, based on the story in Table 3.2 and our interaction with the customer representatives.

Next, the pair proceeds to select a subset of the requirements, which represent some particular modeling issue. When analyzing the CQs, we note that the first one is focused on the album as a collection of tracks, while the second one adds the notion of genre. These are actually quite separate concerns, and in order to decouple these modeling issues, we decide to create one module for each CQ.

Choosing to start with the first CQ and the contextual statement, we now need to match the CQ to the requirements covered by the CPs in the ODP portal. When searching the portal's CP submission table[14], we find that there are several interesting CPs, e.g., there is the *collection* CP[15] for representing membership, and the *part of* CP for part-whole relations. In this case, both patterns have matching CQs, so the choice is instead based on how we wish to view the album, i.e., as an object divided into parts or as a collection that is the sum of its members. One of the main differences between the patterns is that the *part of* CP

[14] http://ontologydesignpatterns.org/wiki/Submissions:ContentOPs

[15] http://ontologydesignpatterns.org/wiki/Submissions:Collection

defines the part-whole relation as transitive, while the membership relation in the *collection* CP is not. Since we are not interested in creating a hierarchy of parts, we decide on the *collection* CP and document this choice in the project wiki (including our argumentation).

Then it is time to start modeling. Since we are creating a new module each time, we start by creating an empty ontology, with a new namespace (following a namespace convention agreed in Step 1). Next, we import the OWL building block of the *collection* CP into our empty ontology and start specializing it. As a subclass of `collection:Collection`, we create a new class `Album`, and then another new class called `Track` (subclass of `owl:Thing`). To complete the specialization, we create sub-properties of the pattern properties, with more domain-specific names, e.g., `containsTrack` and `containedInAlbum`, set them to be inverses, and define domain and range axioms. Figure 3.10 illustrates the result of this process. Each entity we create is commented, and given a label, and we additionally extend the specialized CP, by adding the contextual statement as a cardinality restriction over the `containsTrack` property on the `Album` class.

When the design pair is satisfied, it is time to test the module they have created. First, we formulate the CQ as a SPARQL query. Most often, missing parts are discovered already when formulating the query since the query formulation involves an inspection of the model. However, a new ontology (i.e., a "test case") is created, importing the ontology to be tested, and some test instances are added in the test case ontology. If the SPARQL query gives the expected result, based on our test data, then the test is successful. We can proceed to perform some stress testing. In this case, we should add data that violates some constraint, e.g., a contextual statement, and see that the ontology is able to detect the problem, e.g., through finding an inconsistency, and that there are no undesired

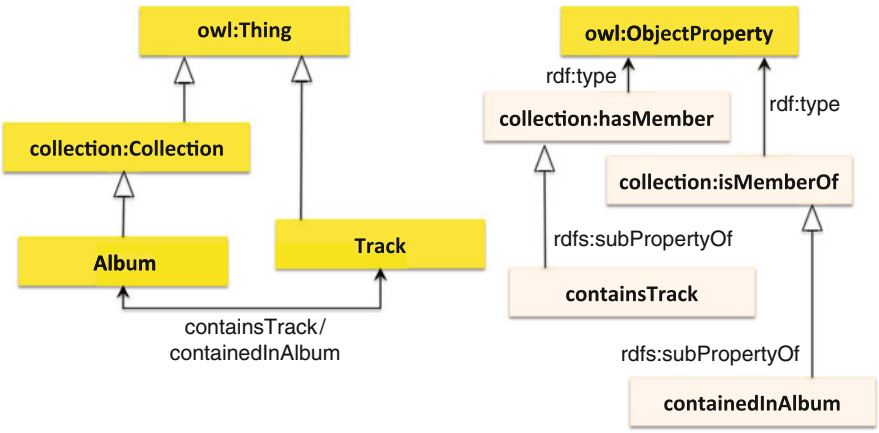

Fig. 3.10 Specialization of the *collection* CP

side effects. In our case, however, the open-world assumption of OWL makes it hard to detect violations of our contextual statement.

When all tests run successfully, and the module is fully annotated and documented in the wiki, it is time to proceed with the next set of requirements, i.e., the second CQ and the reasoning requirement. Similarly as before, we start by matching the requirements to the list of CPs in the ODP portal. This time, we do not immediately find a match, i.e., there is no pattern for music genres; however, there are patterns for expressing descriptions and parameters of a concept. Nevertheless, let us assume that we find these too abstract for our case, and instead choose to create the model on our own.

Just as in the previous iteration, we start by creating a new empty ontology with its own namespace. However, this time we realize that we need the tracks and albums that we just modeled in the previous module; hence, we import it into our new ontology module. Then we add the class `Genre` and a property `hasGenre` (including its inverse `genreOf`). The domain of `hasGenre` is set to the union of `Track` and `Album`, while the range is set to the `Genre` class. In addition, to solve the reasoning requirement, we add a property chain definition to the `hasGenre` property, stating that `hasGenre` can be derived from the combination of the `hasTrack` and `hasGenre` properties, meaning that if an album has a track which in turn has a certain genre, then that album should also be directly connected to the same genre.

Testing this time involves testing the CQ using one or more SPARQL queries but also to test the inferences produced based on the property chain, i.e., to confirm that the reasoning requirement is fulfilled. To do the latter, we create a new empty ontology, import our module to be tested and add some test data that should produce the correct inference. For instance, an album instance can be added, then associated to a track (through `hasTrack`), and the track's genre set to `rock` (through `hasGenre`). When the inferences are materialized, we expect to see that the album is now also associated with the genre `rock`. As soon as all tests run successfully, and the ontology module is appropriately commented, we are now ready to release the complete solution of the customer story in Table 3.2, consisting of our two ontology modules.

Step 5 – Releasing module(s). The modules, and all their wiki documentation, are now made available to the pair in charge of integration.

Step 6 – Integrating partial solutions, evaluating, and revising. As soon as the integration team have more than one solution to work with, i.e., more than one story is covered, they start integrating the modules. Integration is a crucial part and involves a trade-off between refactoring, to reduce overlap between modules, and keeping the decoupling of modules to facilitate later changes and reusability of individual modules. In some cases, integration is quite easy, e.g., the modules can directly be imported into one new ontology, and tested together, without any additional modeling, while in other cases, the integration means to add some "glue" to resolve conflicts and make sure that the requirements of the stories treated so far can be covered all together. However, the use of CPs facilitates the integration since it makes explicit the modeling

choices made, assures that the development team has a shared vocabulary for talking about modeling choices, and in some cases even makes the integration semi-automatic, i.e., if the same CP is imported in several modules they are inherently aligned.

While the integration pair starts their task, our design pair can now go back to the list of remaining user stories, and select a new one, to start another development iteration. This process is continued until no more stories are to be covered.

Step 7 – Releasing new version of ontology network. After each new module has been integrated into the resulting ontology (i.e., ontology network), a new release is created, letting the customer and other parties, e.g., software developers, review and test the ontology at all stages of development.

3.4.4 Tool Support

In this section, we briefly present the ODP portal[16] and the eXtreme Design Tools (XD Tools), two resources that support XD. The ODP portal is a semantic wiki dedicated to best practices of ontology design for the semantic web, with particular focus on ODPs. The ODP portal supports the life cycle of ODPs, i.e., from their proposal to their evaluation and possible certification. CP wiki pages can be created automatically in the wiki by providing, as input, the CP OWL file properly annotated. Currently, the ODP portal supports the life cycle of *Content ODPs, Re-engineering ODPs, Alignment ODPs, Logical ODPs, Architectural ODPs,* and *Lexico-syntactic ODPs*. The ODP portal is associated with a registry of CPs[17].

While the ODP portal is meant to give community support to XD, the XD Tools are meant to assist the execution of the XD methodology. XD Tools are a set of software components released as an Eclipse plugin, accessible through a perspective – eXtreme Design – compatible with Eclipse-based ontology design environments, such as the NeOn Toolkit. Currently, XD Tools are comprised of five main components that allow a user to browse a registry of CPs, search and import them into a local ontology project. Although specialization is possible through native NeOn Toolkit functionalities, XD Tools feature a wizard for specializing a CP, for usability reasons. As a special feature, a service for analyzing an ontology with respect to general modeling best practices is also included.

Figure 3.11 gives an overview of the XD Tools interface as it appears in the NeOn Toolkit. The *ODP Registry view* (bottom left of Fig. 3.11 – enlarged view in Fig. 3.12) exposes a tree-like view of a CP registry that can be browsed by a user. The default registry used is the ODP portal registry, but others can be added through customizing the plugin. When a CP is selected, the *ODP Details view*

[16] The ODP portal main page, http://www.ontologydesignpattern.org

[17] The ODP Portal pattern registry can be downloaded at: http://ontologydesignpatterns.org/schemas/registry.owl

(to the right of the registry view in both figures) shows a description of it, based on the annotations stored in the CP's OWL file. By right clicking on a CP, its OWL file can be downloaded through the "Get" command and put in a local ontology project. The *ODP Selector view* (bottom right of Fig. 3.11 – enlarged view in Fig. 3.13) provides a search service over the CP registry. By clicking on the "Search" icon (highlighted by a small circle in Fig. 3.13), a user can type a natural language query, e.g., a competency question, in a text field, and submit it to a set of search services that return ranked lists of CPs, from which the user can select the most appropriate one(s).

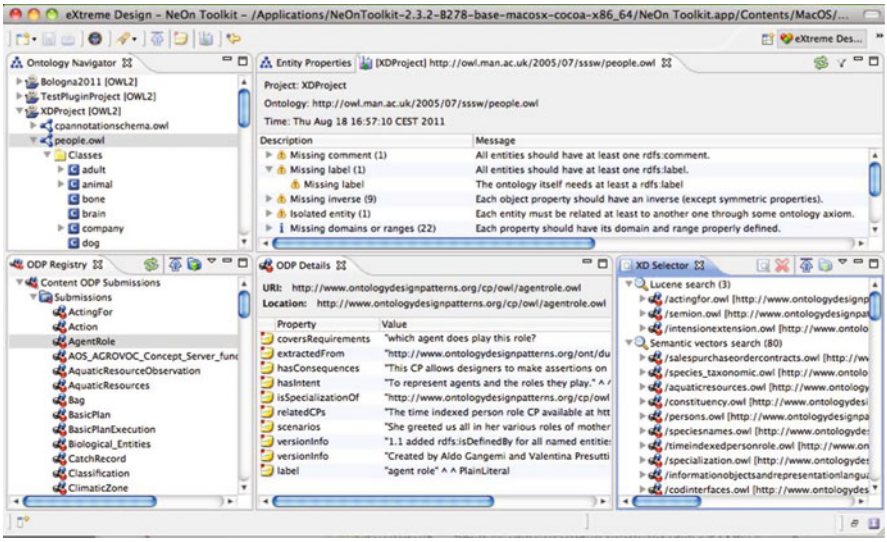

Fig. 3.11 Screenshot overview of XD Tools

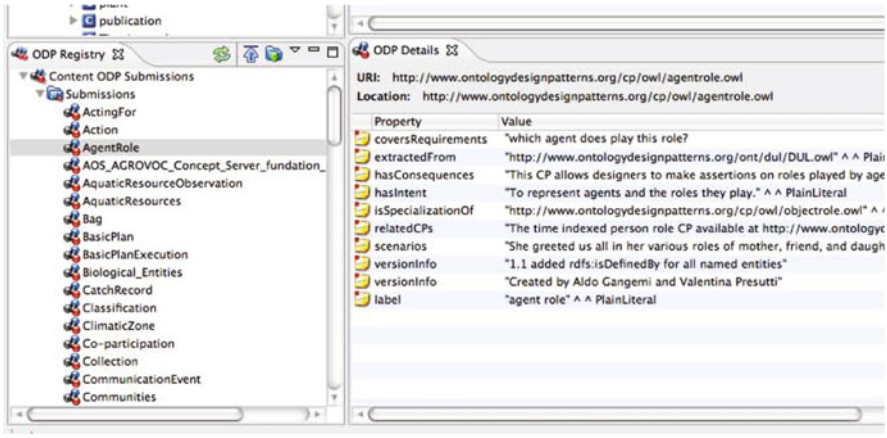

Fig. 3.12 Screenshot depicting the ODP Registry view (*left*) and the ODP Details view (*right*)

The *XD Analyzer view* (top right of Fig. 3.11 – enlarged view in Fig. 3.14) can be run through a contextual menu on a selected ontology and shows a list of messages, each associated with a best practice criterion. A message indicates whether the ontology satisfies a certain criterion or not. The XD Analyzer has a pluggable architecture, allowing for easy extension of the set of heuristics that express "best practices." Three levels of messages can be produced: errors, warnings, and suggestions (i.e., proposals for improvement). An *error* is, for instance, a missing type, i.e., all instances should have an explicit class as its type (could be `owl:Thing`).

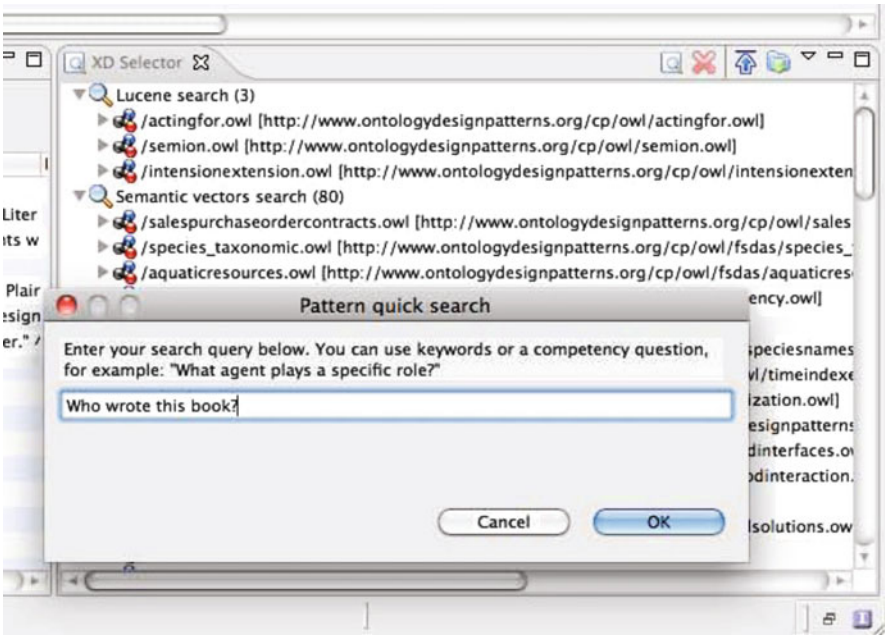

Fig. 3.13 Screenshot depicting the ODP Selector view and its search query interface

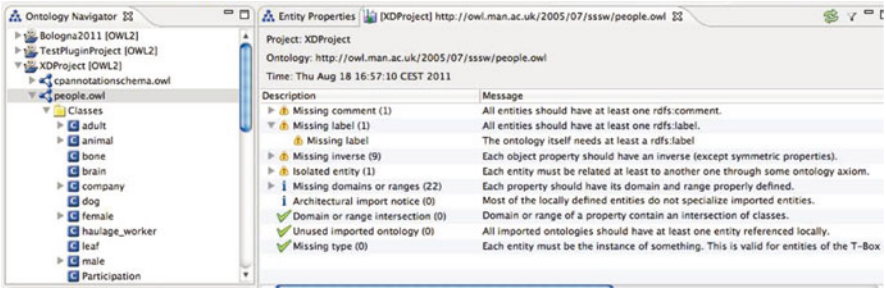

Fig. 3.14 Screenshot depicting the XD Analyzer, showing the results of the analysis in the list to the right. Yellow triangles indicate warnings, while "i" stands for suggestion. The number of occurrences is given within brackets

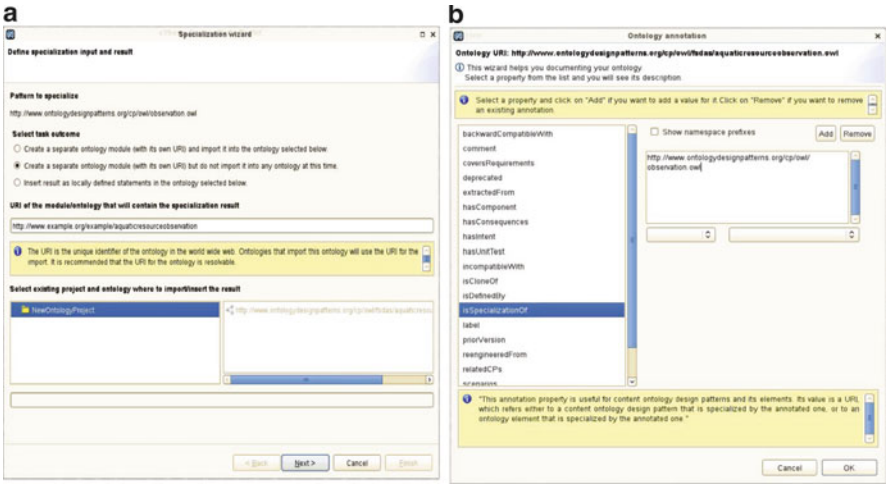

Fig. 3.15 Part (**a**) shows the XD Specialization wizard and (**b**) the XD Annotation dialog

Examples of *warnings* are missing labels and comments, as well as proposals to create an inverse for each object property that has no inverse so far in the analyzed ontology. A *suggestion* could be a message to check the object properties that lack domain and range definitions, i.e., such definitions are not mandatory in a well-designed ontology, since they could be replaced by other axioms; however, if they are missing, it could also indicate that the developer has forgot to add them.

XD Tools also include a wizard for guiding users in the process of specializing a CP. Figure 3.15a shows the *Specialization wizard*. CP specialization, as the primary step of their reuse, can be challenging for an inexperienced user if it is done one element at a time, without guidance. From a user perspective, CP specialization has the following steps: (1) import the pattern into the working ontology, (2) define subclasses/sub-properties for each of the (most specific) pattern entities needed and (3) add any additional appropriate axioms. The specialization wizard provided by XD Tools guides the user through this process. Finally, XD Tools provide a so-called *Annotation dialog* – depicted in Fig. 3.15b – which supports annotation, i.e., documentation, of an ontology based on customizable annotation vocabularies.

In addition, XD Tools provide several help functions, such as inline info boxes, help sections in the Eclipse help center and "cheat sheets" describing the XD methodology for CP reuse.

3.5 Conclusion

In this chapter, we have presented ontology design patterns (ODPs), which are reusable modeling solutions that encode modeling best practices, by briefly discussing their different types and characteristics. ODPs are the main tool for

performing pattern-based design of ontologies, which is an approach to ontology development that emphasizes reuse and promotes the development of a common "language" for sharing knowledge about ontology design best practices. ODPs are associated with a set of requirements that are explicitly expressed in order to favor their selection through a matching procedure. Content ODPs (CPs) have been the main focus of this chapter, which has shown through some examples how they can be used for building an ontology according to a set of elicited requirements. CPs are domain-dependent patterns, the requirements of which are expressed by means of competency questions, contextual statements, and reasoning requirements. In order to reuse CPs, we have defined a set of operations that include importing, specializing, and composing them to the aim of building a new ontology (or ontology network).

In the second part of the chapter, we have described an agile methodology for pattern-based ontology design named eXtreme Design (XD), an iterative and incremental process, which is characterized by a test-driven and collaborative development approach. The XD methodology is supported by a set of software components named XD Tools, which assist users in the process of pattern-based design.

The XD methodology has been tested in numerous ontology development projects, including user-based experiments conducted in controlled environments. The results of those experiments have been reported by Blomqvist and colleagues (2009a) and by Blomqvist and colleagues (2010a). The participants were, for instance, asked to assess how useful ODPs and XD were and how much overhead it added to their work processes, as well if XD felt like a natural way of working, i.e., if they were already working in a similar way before being introduced to the methodology. Overall, the methodology was received well, and the participants felt that it was a very natural way of working, without adding any unnecessary restrictions to the process. Nevertheless, the objective evaluation of their modeling results showed that the quality increased drastically, in particular with respect to a number of common mistakes, when introducing the methodology. This could be attributed to the testing focus of the methodology that enforces a rigorous evaluation of each solution before its release. So even though the participants felt that the methodology added nothing new, it actually helped them to structure their work and provide better and more rigorously tested ontologies.

References

Aguado de Cea G, Gómez-Pérez A, Montiel-Ponsoda E, Suárez-Figueroa MC (2009) Using linguistic patterns to enhance ontology development. In: Dietz J (ed) Proceedings of the international conference on knowledge engineering and ontology development (KEOD), Funchal, pp 206–213

Baker CF, Fillmore CJ, Lowe JB (1998) The Berkeley FrameNet project. In: Boitet C, Whitelock P (eds) Proceedings of the 36th annual meeting of the Association for Computational Linguistics

and 17th international conference on computational linguistics, vol 1. Association for Computational Linguistics, Stroudsburg, PA, USA, pp 86–90
Basili V, Caldiera G, Rombach D (1994) The experience factory. In: Marciniak J (ed) Encyclopedia of software engineering. Wiley, New York, pp 469–476
Bizer C, Heath T, Berners-Lee T (2009) Linked data – the story so far. Int J Semant Web Inf Syst 5 (3):1–22
Blomqvist E, Gangemi A, Presutti V (2009a) Experiments on pattern-based ontology design. In: Proceeding of K-CAP 2009, Los Angeles. ACM, New York
Blomqvist E, Sandkuhl K, Scharffe F, Svatek V (2009b) Proceedings of the workshop on ontology patterns (WOP 2009), collocated with the 8th international semantic web conference (ISWC-2009), Washington, DC, USA, 25 Oct, 2009, vol 516. CEUR
Blomqvist E, Presutti V, Daga E, Gangemi A (2010a) Experimenting with eXtreme design. In: Proceedings of EKAW2010 – knowledge engineering and management by the masses, LNCS 6317. Springer, Berlin/Heidelberg/New York
Blomqvist E, Chaudhri V, Corcho O, Presutti V, Sandkuhl K (2010b) Proceedings of the 2nd international workshop on ontology patterns – WOP2010, vol 671. CEUR
Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
Gangemi A, Borgo S (2004) Core ontologies in ontology engineering 2004. (Un) Successful cases and best practices for ontology engineering: reusing well-founded ontologies for domain content specification. In: Proceedings of the EKAW*04 workshop on core ontologies in ontology engineering, Northamptonshire (UK), 8 Oct, 2004, vol 118. CEUR
Gangemi A, Presutti V (2009) Ontology design patterns. In: Staab S, Studer R (eds) Handbook on ontologies, 2nd edn. Springer, Berlin, pp 221–243
Gangemi A, Fisseha F, Keizer J, Lehmann J, Liang A, Pettman I, Sini M, Taconet M (2004) A core ontology of fishery and its use in the FOS project. In: EKAW 2004 workshop on core ontologies in ontology engineering, Northamptonshire. CEUR
Gangemi A, Sagri MT, Tiscornia D (2005) A constructive framework for legal ontologies. In: Law and the semantic web. Legal ontologies, methodologies, legal information retrieval, and applications. 3369. Springer, Berlin/Heidelberg/New York
Gruninger M, Fox MS (1994) The role of competency questions in enterprise eEngineering. In: IFIP WG5.7 workshop on benchmarking – theory and practice, Trondheim
Hammar K, Sandkuhl K (2010) The state of ontology pattern research: a systematic review of ISWC, ESWC and ASWC 2005–2009. In: Blomqvist E, Chaudhri VK, Corcho O, Presutti V, Sandkuhl K (eds) Proceedings of the 2nd International workshop on ontology patterns – WOP2010. Workshop at the 9th international semantic web conference (ISWC2010) – ISWC 2010 workshops, vol VIII. Shanghai, China, 8 Nov, 2010, vol 671. CEUR
Hay DC (2000) Data model patterns: conventions of thought. Dorset House Publishing, New York
Masolo C, Borgo S, Gangemi A, Guarino N, Oltramari A (2005) The wonderweb library of foundational ontologies. Wonderweb deliverable D18. Laboratory for applied ontology (ISTC-CNR)
Miles A, Bechhofer S (2009) SKOS simple knowledge organization system reference. W3C
Niles I, Pease A (2001) Towards a standard upper ontology. In: Welty C, Smith B (eds) 2nd international conference on formal ontology in information systems (FOIS-2001), Ogunquit
Noy N, Rector A (2004) Defining N-ary relations on the semantic web: use with individuals. W3C
Presutti V, Daga E, Gangemi A, Blomqvist E (2009) eXtreme design with content ontology design patterns. In: Blomqvist E, Sandkuhl K, Scharffe F, Svatek V (eds) Proceedings of the workshop on ontology patterns (WOP 2009), collocated with the 8th international semantic web conference (ISWC-2009), Washington, DC, USA, 25 Oct 2009, vol 516. CEUR
Rector A, Stevens R (2008) Barriers to the use of OWL in knowledge driven applications. In: Dolbear C, Ruttenberg A, Sattler U (eds) Proceedings of the fifth OWLED workshop on OWL: experiences and directions collocated with the 7th international semantic web conference (ISWC-2008) Karlsruhe, Germany, 26–27 Oct 2008, vol 432. CEUR

Scharffe F, Fensel D (2008) Correspondence patterns for ontology alignment. In: Gangemi A, Euzenat J (eds) Proceedings of the 16th international conference, EKAW 2008, Acitrezza, Italy. 5268. Springer, Berlin/Heidelberg/New York, pp 83–92

Shore J, Warden S (2007) The art of agile development. O'Reilly, Farnham

Svátek V, Sváb-Zamazal O, Presutti V (2009) Ontology naming pattern sauce for (human and computer) gourmets. In: Workshop on ontology patterns at ISWC'09, Washington DC, 2009. 516. CEUR

Vrandečić D, Gangemi A (2006) Unit tests for ontologies. In: Proceedings of the 1st international workshop on ontology content and evaluation in enterprise. Springer, Berlin/Heidelberg/New York

Vrandečić D, Sure Y (2007) How to design better ontology metrics. In: May W, Kifer M (eds) 4th European semantic web conference (ESWC'07). Springer, Berlin/Heidelberg/New York