NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 — "Semantic-based knowledge and content systems"

D2.5.1: A Library of Ontology Design Patterns: reusable solutions for collaborative design of networked ontologies.

Deliverable Co-ordinator: **Valentina Presutti**

Deliverable Co-ordinating Institution: **CNR**

Other Authors: **Aldo Gangemi (CNR), Stefano David (CNR), Guadalupe Aguado de Cea (UPM), Mari Carmen Suárez-Figueroa (UPM), Elena Montiel-Ponsoda (UPM), María Poveda (UPM) - (see the executive summary for details on respective contributions).**

This deliverable collects the work made so far in the task 2.5 of WP2 on reusable modeling solutions for collaborative ontology design.

Document Identifier:	NEON/2007/D2.5.1/v1.2	Date due:	February 15th, 2008
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	February 13th, 2008
Project start date	March 1, 2006	Version:	v1.2
Project duration:	4 years	State:	Final
		Distribution:	Public

2006–2008 © Copyright lies with the respective authors and their institutions.

NeOn Consortium

This document is part of the NeOn research project funded by the IST Programme of the Commission of the European Communities by the grant number IST-2005-027595. The following partners are involved in the project:

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, e.motta}@open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB Englerstrasse 11 D-76128 Karlsruhe, Germany Contact person: Peter Haase E-mail address: pha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.ump.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Jesús Contreras E-mail address: jcontreras@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SL–1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 665 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier, France Contact person: Jérôme Euzenat E-mail address: jerome.euzenat@inrialpes.fr	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield, United Kingdom Contact person: Hamish Cunningham E-mail address: hamish@dcs.shef.ac.uk
Universität Kolenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Marino della Battaglia 44 – 00185 Roma-Lazio Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Food and Agriculture Organization of the United Nations (FAO)** Viale delle Terme di Caracalla 00100 Rome Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@fao.org
Atos Origin S.A. (ATOS) Calle de Albarracín, 25 28037 Madrid Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com	**Laboratorios KIN, S.A. (KIN)** C/Ciudad de Granada, 123 08018 Barcelona Spain Contact person: Antonio López E-mail address: alopez@kin.es

2006–2008 © Copyright lies with the respective authors and their institutions.

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to the writing of this document or its parts:

- Partner 1: CNR
- Partner 2: UPM
- Partner 3: UKO-LD
- Partner 4: INRIA

Change Log

Version	Date	Amended by	Changes
1.0	10-01-2008	Valentina Presutti	Draft sent to the internal reviewer
1.1	01-02-2008	Valentina Presutti	Pre-Final version completed (clarification of definitions, correction of errors and addition of examples)
1.2	13-02-2008	Valentina Presutti and Aldo Gangemi	Final version completed (all review issues addressed and contribution from UPM partner added. Title updated. Positioning within WP2 tasks. Some DOLCE-related patterns validation)
1.3	15-02-2008	Valentina Presutti and Aldo Gangemi	Some formatting updates to the final version

Executive Summary

This deliverable collects the work made so far in the task 2.5 of WP2 on reusable modeling solutions for collaborative ontology design. The focus of 2.5 is on the *reusability* aspects of collaboration, rather than on specific modeling support for collaboration proper.
The work presented assumes a broad notion of *ontology design pattern*, which, also exploiting the similarities with software engineering patterns, encompasses solutions ranging from logical to content modeling, from mapping to lexico-syntactic presentation, from naming to reengineering, etc.
Ontology design patterns so characterized constitute a repository of solutions to recurrent modeling problems, and comply with the vision of collaborative ontology design formalized in D2.1.1.
The main focus of this deliverable is on *content* ontology design patterns, while the next version will contain more material on the other types of patterns. The deliverable consists of three parts: the first part (chapters 1 to 3) explains the basic notions, pattern types, the historical background, and a minimal semantics for pattern operations, also providing examples, diagrams, and instantiations.
The second part (chapter 4) is a catalog of content patterns, which is neither exhaustive nor definitive, but gives a hint of what is being incrementally populated on a dedicated public portal, http://www.ontologydesignpatterns.org, where patterns can be proposed, discussed, and recommended. The template used for the presentation has been optimized for the web and defined in an OWL annotation schema; it is the same used on the semantic web portal.
The third part (Appendix A) is an appendix that includes a presentation of catalog patterns according to a template tailored for software engineers (but less appropriate for the web), and is compatible with the preliminary presentation of patterns in D5.1.1.
T2.5 work is related to experimental and user study work that is being carried out in WP5, as well as to the modularization work that is being carried out in WP1. A preliminary (now obsolete) version of the Appendix was in fact presented in D5.1.1, while experimental settings for ontology design patterns evaluation are presented in D5.6.1. A preliminary liaison between content patterns and modularization semantics is presented in D1.1.3.
Ongoing and future work in T2.5 will focus on the pattern portal and evaluation, while theoretical refinements will be mainly related to the ontology modularization semantics and algebra presented in D1.1.3, OMV and Oyster (WP1-WP6), and the evaluation, selection and reengineering methods from T2.2.

Contribution details
CNR has coordinated the editing of the deliverable, and has contributed the first (except section 2.2.5)) and second parts.
UPM has contributed the Appendix (A) and section 2.2.5.

2006–2008 © Copyright lies with the respective authors and their institutions.

Contents

1 Introduction 13
1.1 Background . . . 15

2 Ontology Design Patterns 16
2.1 Definition of Ontology Design Pattern . . . 16
2.2 Types of Ontology Design Patterns . . . 18
2.2.1 Structural OPs . . . 19
2.2.2 Correspondence OPs . . . 25
2.2.3 Reasoning OPs . . . 27
2.2.4 Presentation OPs . . . 28
2.2.5 Lexico-Syntactic OPs . . . 29
2.2.6 Content OPs (CPs) . . . 34

3 Creation and Usage of Content Ontology Design Patterns 36
3.1 Requirements and use cases . . . 36
3.2 Operations . . . 37
3.2.1 Covering . . . 37
3.2.2 Clone . . . 38
3.2.3 Composition . . . 41
3.2.4 Specialization and Generalization . . . 43
3.2.5 Expansion . . . 44
3.2.6 Import . . . 45
3.3 Definition of Content Ontology Design Pattern (CP) . . . 45
3.3.1 Ontological resources that are not CPs . . . 47
3.3.2 Content Ontology Design Anti-pattern (AntiCP) . . . 47
3.4 Where do CPs come from? . . . 47
3.4.1 Examples of CP creation . . . 49
3.5 The CP annotation schema . . . 51
3.6 How to use content ontology design patterns . . . 53
3.6.1 Matching between *intent* and use case . . . 54
3.6.2 Pattern-based ontology evaluation . . . 55

4 Catalogue of Content Ontology Design Patterns **60**
4.1 General 61
4.1.1 Types of entities 61
4.1.2 Description 63
4.1.3 Situation 64
4.1.4 Classification 66
4.1.5 N-ary Classification 67
4.1.6 Description and Situation 68
4.1.7 Object Role 69
4.2 Parts and collections 71
4.2.1 Part of 71
4.2.2 Time Indexed Part of 71
4.2.3 Componency 73
4.2.4 Constituency 75
4.2.5 Collection Entity 76
4.3 Semiotics 77
4.3.1 Intension Extension 77
4.3.2 Information Realization 79
4.4 Quantities and Dimensions 80
4.4.1 Region 80
4.4.2 Region Overlap 82
4.4.3 Parameter 83
4.4.4 Parameter Region 85
4.5 Participation 86
4.5.1 Participation 86
4.5.2 Co-participation 87
4.5.3 N-ary Participation 89
4.6 Organization, Management, and Scheduling 90
4.6.1 Precedence 90
4.6.2 Agent Role 91
4.6.3 Task Role 93
4.6.4 Time-Indexed Person Role 94
4.6.5 Basic Plan Description 95
4.6.6 Basic Plan Execution 97
4.6.7 Basic Plan 99
4.7 Business 101
4.7.1 Price 101
4.7.2 Sales and Purchase Order Contracts 102

2006–2008 © Copyright lies with the respective authors and their institutions.

4.8 Time . . . 104
4.8.1 Time Interval . . . 104
4.9 Space . . . 106
4.9.1 Move . . . 106
4.10 Life Science . . . 108
4.10.1 Linnean Taxonomy . . . 108
4.11 Multimedia . . . 109
4.11.1 Multimedia Data Segment Decomposition . . . 109
4.12 Towards a Web Portal of CPs . . . 111

A Software Engineering templates **113**
A.1 General . . . 113
A.2 Parts and collections . . . 123
A.3 Semiotics . . . 130
A.4 Quantities and Dimensions . . . 134
A.5 Participation . . . 138
A.6 Organization, Management, and Scheduling . . . 143
A.7 Business . . . 159
A.8 Time . . . 166
A.9 Space . . . 168
A.10 Life Science . . . 171
A.11 Multimedia . . . 175

Bibliography **179**

List of Tables

2006–2008 © Copyright lies with the respective authors and their institutions.

List of Figures

2.1 Ontology Design Pattern definition . . . 18
2.2 Ontology Design Pattern types . . . 19
2.3 The n-ary relation Logical OP . . . 21
2.4 An example of usage of the n-ary relation Logical OP . . . 21
2.5 The classes as values Logical OP . . . 22
2.6 An example of usage of the classes as values Logical OP . . . 22
2.7 The transitive reduction Logical OP . . . 23
2.8 An example of usage of the transitive reduction Logical OP . . . 24
2.9 The relation between datatype values Logical OP . . . 25
2.10 An example of usage of the relation between datatype values Logical OP . . . 25
2.11 The modular architecture Architectural OP . . . 26
2.12 Relation between CPs and Logical OPs. . . . 35

3.1 The agent role CP . . . 38
3.2 The time indexed person role CP . . . 39
3.3 The definition of the `DUL:Agent` class. . . . 39
3.4 The definition of the `AgentClone` class. . . . 40
3.5 The definition of the `DUL:SocialAgent` class. . . . 41
3.6 The result of a partial clone of the `DUL:SocialAgent` class. . . . 42
3.7 The agent role CP . . . 43
3.8 The task role CP . . . 43
3.9 The composition of **agent role** and **task role** CPs . . . 44
3.10 Containment vs Subsumption: an example of AntiCP . . . 48
3.11 The CP extraction process. . . . 50
3.12 The information realization CP extraction from Dolce Ultra Lite ontology. . . . 50
3.13 Ontology elements extracted and specialized from the Dolce Ultra Lite ontology. . . . 52
3.14 The music industry example. . . . 55

4.1 The types of entities CP . . . 62
4.2 The types of entities example scenario. . . . 62
4.3 The description CP . . . 63

4.4 The description example scenario: preparation of a coffee. 64
4.5 The situation CP 65
4.6 The situation example scenario: a particular preparation of a coffee. 65
4.7 The classification CP 66
4.8 The classification example scenario: classification of an operating system. 66
4.9 The n-ary classification CP 67
4.10 The n-ary classification example scenario: different operating systems. 68
4.11 The description and situation CP 69
4.12 The object role CP 70
4.13 The object role example scenario: an old glass used as a flower pot. 70
4.14 The part of CP 72
4.15 The part of example scenario: brain and heart are parts of the human body, substantia nigra is part of brain. Notice that by transitivity, also substantia nigra is inferred as part of human body (blue association denotes the inferred relation). 72
4.16 The time indexed part of CP 72
4.17 The time indexed part of example scenario: Toyota Yaris pneumatics changing over time. . . . 73
4.18 The componency CP 74
4.19 The componency example scenario 74
4.20 The constituency CP 75
4.21 The constituency example scenario 75
4.22 The collection entity CP 76
4.23 The collection entity example scenario: members of the Laboratory for Applied Ontology. . . . 77
4.24 The intension extension CP. 78
4.25 The intension extension example scenario. 78
4.26 The information realization CP 80
4.27 The information realization scenario: the *I Me Mine* John Lennon's biography. 80
4.28 The region CP 81
4.29 The region example scenario: number of wheels. 81
4.30 The region CP 82
4.31 The region example scenario 83
4.32 The parameter CP 84
4.33 The parameter example scenario: phases of a book. 84
4.34 The parameter region CP 85
4.35 The parameter region example scenario 86
4.36 The participation CP. 87
4.37 The participation example scenario 87
4.38 The co-participation CP 88
4.39 The co-participation example scenario: co-participation in the ISWC 2007 conference. 88
4.40 The n-ary participation CP 89

2006–2008 © Copyright lies with the respective authors and their institutions.

4.41 The n-ary participation example scenario: co-participation of objects to a certain event at a certain time, place, etc. . . . 90
4.42 The precedence CP. . . . 91
4.43 The precedence CP . . . 91
4.44 The agent role CP . . . 92
4.45 The agent role example scenario . . . 92
4.46 The task role CP . . . 93
4.47 The task role example scenario: system configuration (administrator) and content writing (author). . . . 94
4.48 The time indexed person role CP . . . 95
4.49 The time indexed person role example scenario. . . . 96
4.50 The basic plans description CP . . . 96
4.51 The basic plan description example scenario. . . . 97
4.52 The basic plan execution CP . . . 98
4.53 The basic plans execution example scenario: the ISWC paper writing. . . . 99
4.54 The basic plans CP . . . 100
4.55 The price CP . . . 101
4.56 The price example scenario: the price of macbook pro 2.2. GHz expressed in Euro and Dollars. 102
4.57 The sales and purchase order contracts CP . . . 103
4.58 The sales and purchase order contracts example scenario: a purchase of Queen Greatest hits album. . . . 104
4.59 The time interval CP . . . 105
4.60 The time interval example scenario . . . 105
4.61 The move CP . . . 106
4.62 The move example scenario . . . 106
4.63 The Linnean taxonomy CP . . . 108
4.64 The Linnean taxonomy example . . . 109
4.65 The multimedia data segment decomposition CP . . . 110
4.66 The ODP publishing and quality workflow. . . . 112

Chapter 1

Introduction

In the NeOn Deliverable 2.1.1, a complex model of collaborative ontology design (C-ODO) has been provided, including basic distinctions and associations between *ontology projects*, *design workflows*, *design rationales*, *design solutions*, etc.
The notion of “pattern” has proved useful in design, as exemplified in diverse areas such as architecture, software engineering, etc. Within C-ODO, *Ontology design Patterns* (OP) play a crucial role, since they summarize the good practices to be applied within design solutions, and keep track of the design rationales that have motivated their adoption. Following a line of research dating back at least to 2003, in this deliverable we provide a library of OPs, accompanied by a minimal theoretical framework and methods to apply them in realistic ontology projects.
Although preliminarily, we will also liaise the catalogue and the meta-model for OPs to related NeOn work, notably the networked ontology model (WP1) and the meta-models for mapping and modularity (WP1), the reengineering models and methods (WP2), and the NeOn methodology (WP5).
In the remainder of this introduction, we provide a motivation why patterns are important in ontology design.

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. In this deliverable, we focus on patterns for ontology design [Gan05, GLP+07].
Despite the original ontology engineering approach, when ontologies were seen as “portable” components [Gru93], and the key role played by ontologies in the semantic web and interoperability issues, 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 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

2006–2008 © Copyright lies with the respective authors and their institutions.

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 [BM05] and SKOS [MB05] 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 propose 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 deliverable, we will provide a typology of OPs, but we will mainly focus on a catalogue of 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 [GLP+07]. We call these basic building blocks to be used in ontology design *Content Ontology design Patterns* (Content OPs, [Gan05]). Content OPs 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 design patterns, with appropriate dependencies between them.

Throughout experiences in ontology engineering projects[1] at the Laboratory for Applied Ontology (LOA),[2] 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 **participation** pattern (including objects taking part in events) emerges in domain ontologies as different as enterprise models [GF94], legal norms [GPS01], sofware management [OMGS04], biochemical pathways [GCB04], and fishery techniques [GFK+04].
Other, more complex patterns have also emerged in the same disparate domains: the **role-task** pattern, the **information-realization** pattern, the **description-situation** pattern, etc.
Moreover, since Content OPs are strictly related to small use cases, they are transparent with respect to the rationales applied to the design of a certain ontology. Content OPs 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. [VG06]) that are typical of the tasks addressed by a designer. Furthermore, mapping and composition of patterns can facilitate ontology mapping and alignment/merging: two ontologies drafted according to Content OPs can be mapped in an easier way: Content OP hierarchies are more stable and more easily maintainable than local, partial, scattered ontologies. Finally, Content OPs can be also used in training and educational contexts for ontology engineers.
Content OPs are a very beneficial kind of patterns for ontology design, because they provide solutions to domain-oriented problems, but there are other types of OPs, which will be also introduced in this chapter for a general overview.
The content of this deliverable is organized as follows: section 1.1 gives some background notions; chapter 2 defines what an OP is and introduces a typology of OPs; chapter 3 focuses on Content OPs, defines the typical operations used for, and describes ways to create and work with Content OPs. Chapter 4 contains a catalogue of Content OPs. Finally, the catalogue is presented by means of a software engineering-oriented template.

[1] For example, in the projects *IKF*: http://www.ikfproject.com/About.htm, *FOS*: http://www.fao.org/agris/aos/, *WonderWeb*: http://wonderweb.semanticweb.org, and *Metokis*: http://metokis.salzburgresearch.at
[2] http://www.loa-cnr.it

1.1 Background

The term “pattern” appears in English in the 14th century and derives from Middle Latin “patronus” (meaning “patron”, and, metonymically, “exemplar”, i.e. something proposed for imitation).[3] As Webster’s puts it, a pattern has a set of senses that show a reasonable degree of similarity (see italics): a) a *form* or *model* proposed for *imitation*, b) something *designed* or used as a *model* for *making things*, c) a *model* for making a *mold*, d) an artistic, musical, literary, or mechanical *design* or *form*, e) a natural or chance *configuration*, etc., and, f) a *discernible coherent system* based on the *intended interrelationship* of component parts’.
In the seventies, the architect and mathematician Christopher Alexander introduced the term “design pattern” for shared guidelines that help solve design problems. In [Ale79] 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 context.
Taking seriously the architectural metaphor, the notion has been eagerly endorsed by software engineering [GW99, GHJV95, MHG02], and DBMS applications with so-called data model patterns [Hay96]. 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 [CTP00, GW04, Rei00, DWW04, Sva04, Sos03].[4] While traditional design patterns appear more like a collection of shortcuts and suggestions related to a class of context-bound problems and success stories, in recent work there is a tendency towards a more formal encoding of design patterns (notably [BFL98, GW04, MHG01]), and even towards using ontology patterns to encode software engineering patterns [OMGS04].
Ontology engineering literature has tackled the notion of design pattern at least since [CTP00, Rei00], while in the context of Semantic Web research and application the notion has been introduced by [GCB04, RR04, Sva04]. In particular, [GCB04, Sva04] take a foundational approach that anticipates that presented in [Gan05] (which is closely related to this chapter). Some work [Blo05, BS05] 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 patterns, e.g. for content objects and multimedia [GBCL05, ATS$^+$07], software components [Obe06], business modelling and interaction [GP07], etc. Moreover, the root of what we call here *Logical OPs* can be found in [Bra77], 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).

[3] Cf. Online Etymology Dictionary: http://www.etymonline.com

[4] In software engineering, formal approaches to design patterns, based on dedicated ontologies, are being investigated, e.g. in so-called *semantic middleware* [OMGS04]

2006–2008 © Copyright lies with the respective authors and their institutions.

Chapter 2

Ontology Design Patterns

In this chapter, we define the concept of Ontology Design Pattern (OP) and provide its ontological context according to the C-ODO ontology [CLN+07]. We identify OP types, group them into families, and provide some examples.
In this chapter, we use the following formatting conventions: terms represented by either `typewriter` font (in ordinary text), or by *`typewriter italic`* font (within text of definitions) refer to ontology elements. Each term includes a prefix, which identifies the ontology namespace it belongs to.

2.1 Definition of Ontology Design Pattern

The goal of this section is to give a precise definition of OP. C-ODO includes formal definitions for the notion of OP in terms of concepts from the DOLCE Ultra-Lite ontology[1]. The OP definition is at the end of this section. Before, we clarify the meaning of each ontology element involved in that definition. The namespaces we use below and their corresponding prefixes are the followings:

- the default namespace is http://www.loa-cnr.it/ontologies/OD/odSolutions.owl#
- `dul:` is for http://www.loa-cnr.it/ontologies/DUL.owl#
- `oddata:` is for http://www.loa-cnr.it/ontologies/OD/odData.owl#
- `owl:` is for http://www.w3.org/2002/07/owl#
- `rdf:` is for http://www.w3.org/1999/02/22-rdf-syntax-ns#
- `rdfs:` is for http://www.w3.org/200/01/rdf-schema#

An OP is an *information object*, which is defined as follows.

Definition 1 (Information Object) *An information object i.e. `dul:InformationObject` is a piece of information about some entity that can be interpreted by an agent. It is encoded within some system for information encoding, and realized by some entity. Consequently, information objects are dependent from an encoding as well as from a concrete realization.*

Examples of information objects are: a musical composition, a text, a word, a picture. From a communication perspective, an information object can play the role of 'message'. From a semiotic perspective, it plays the role of 'expression'. Information objects can express a description (the ontological equivalent of a meaning/conceptualization)[2], which is defined as follows.

[1] http://www.loa-cnr.it/ontologies/DUL.owl
[2] see also sections 4.3.2 and 4.3.1

Definition 2 (Description) *A description (`dul:Description`) represents a conceptualization (e.g. a mental object or state), hence it is dependent on some agent and communicable. Descriptions can also be taken as reifications of cognitive contexts or logical n-ary relations. Descriptions use concepts, are expressed by information objects and can be satisfied by situations.*

A description can be thought also as a *descriptive context* that defines concepts in order to model a relational context out of a set of data or observations. For example, a plan is a description of some actions to be executed by agents in a certain way, with certain parameters; a diagnosis is a description that provides an interpretation for a set of observed entities, etc.[3]. The factual counterpart of descriptions are *situations*, which are defined as follows.

Definition 3 (Situation) *A situation (`dul:Situation`) is a setting for entities, typically consistent with a Description.*

A situation can be seen as a *relational context* created by an observer on the basis of a description. For example, the execution of a plan is a setting for some actions executed by agents according to certain parameters and expected workflows from a plan, a diagnosed situation is a context of observed entities that is interpreted on the basis of a diagnosis, etc. Two descriptions of a same situation are possible. The time and space (and possibly other values) of a situation are the time and space of the events and objects occurring in the setting.[4].

OPs express a design pattern schema, which is a description that can be satisfied by some design solution. *Design pattern schemas* are defined as follows.

Definition 4 (Design Pattern Schema) *A design pattern schema (`DesignPatternSchema`) is the description of an OP. It includes the roles, tasks, and parameters needed in order to solve an ontology design issue.*

For example, a design pattern schema for content creation provides instructions to clone, choose, compose, or specialize an existing ontology or collection of ontology elements, possibly according to good practices. Structural descriptions of OPs have simpler, non-procedural descriptions, which are called here *skins*.

Definition 5 (Design Pattern Skin) *A design pattern skin (`DesignPatternSkin`) is a design pattern schema describing the pure structure of an OP. It identifies the elements to be put into an ontology that is supposed to reuse an OP.*

Design pattern schemas and design pattern skins define *element roles* that only can be played by ontology elements. For example, a skin for a **partof** OP would be defined e.g. with one element role for a mereological relation, and two *element roles* e.g., *part*, and *whole*, for two entities of a same category (e.g. either objects or events). *Ontology elements* and *element roles* are defined below.

Definition 6 (Ontology Element) *An ontology element (`oddata:OntologyElement`) is an (identified) element from an ontology. Ontology elements are also modelled in the Ontology Definition Metamodel (ODM) [ea05], the Ontology Metadata Vocabulary (OMV) [HBP$^+$07], etc.*

Examples of ontology elements include e.g. OWL classes, OWL properties, and OWL restrictions.

Definition 7 (Element Role) *An element role (`ElementRole`) is a concept that classifies an ontology element.*

[3]see also sections 4.1.2 and 4.1.6
[4]see also sections 4.1.3 and 4.1.6

2006–2008 © Copyright lies with the respective authors and their institutions.

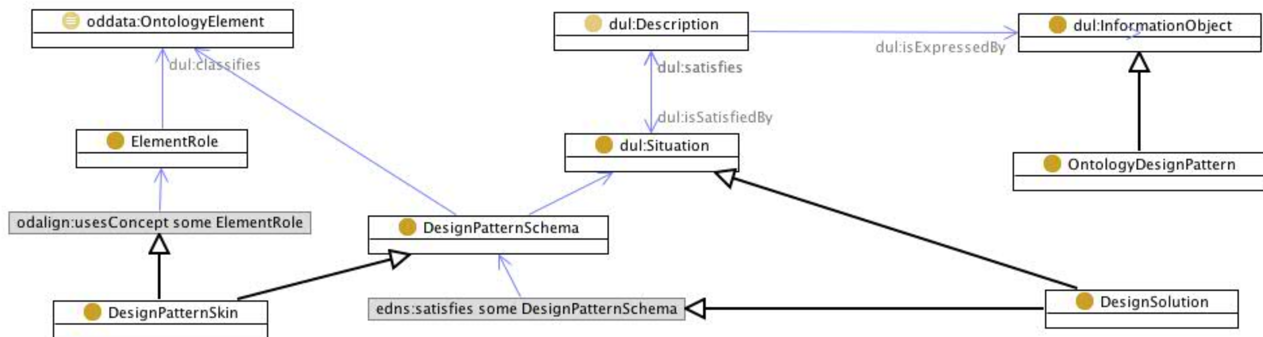


Figure 2.1: Ontology Design Pattern definition

Design pattern schemas and design pattern skins can be satisfied by *design solutions*, which are defined as follows.

Definition 8 (Design Solution) *A design solution (`DesignSolution`) is a situation that includes only formal expressions (at least one Content OP or logical construct), and their relations.*

For example, the use of the **part of** Content OP (also presented in section 4.2.1), in order to model the parts of a country is a design solution. It is a setting for the **partOf** relation (which is an ontology element) and its arguments.
Finally, in agreement with above definitions, and as depicted in Figure 2.1, we define:

Definition 9 (Ontology Design Pattern) *An OP (`OntologyDesignPattern`) is a modeling solution to solve a recurrent ontology design problem. It is an `dul:InformationObject` that `dul:expresses` a `DesignPatternSchema` (or skin). Such schema can only be satisfied by `DesignSolution`s. Design solutions provide the setting for `oddata:OntologyElement`s that play some `ElementRole`(s) from the schema.*

The concept of OP abstracts from the particular formalism that is used in order to represent them. However, the representation formalism is a key aspect for their usage. In the context of this deliverable, we deal with OPs for the OWL representation language in the DL species. For this reason, we refer to OWL DL semantics [OWL04] to define the notions of *OWL ontology element* and *OWL DL axiom*, which will be used in the rest of the document.

Definition 10 (OWL Ontology Element, oe) *An OWL Ontology Element is an identified element in an OWL DL ontology, which `dul:isClassifiedBy` one of the following: `owl:Class`, `owl:DatatypeProperty`, `owl:ObjectProperty`, OWL Individual, or OWL Property Value.*

Definition 11 (OWL DL Axiom, ax) *An OWL DL Axiom is an assertion, which is built by means of one of the following elements: `rdfs:subClassOf`, `owl:equivalentClass`, `owl:disjointWith`, `owl:complementOf`, `rdfs:subPropertyOf`, `owl:equivalentProperty`, `owl:hasValue`, `owl:inverseOf`, `rdfs:domain`, `rdfs:range`, `owl:sameAs`, `owl:oneOf`, and `owl:differentFrom`.*

2.2 Types of Ontology Design Patterns

Our definition of OP includes a wide range of modeling solution types.

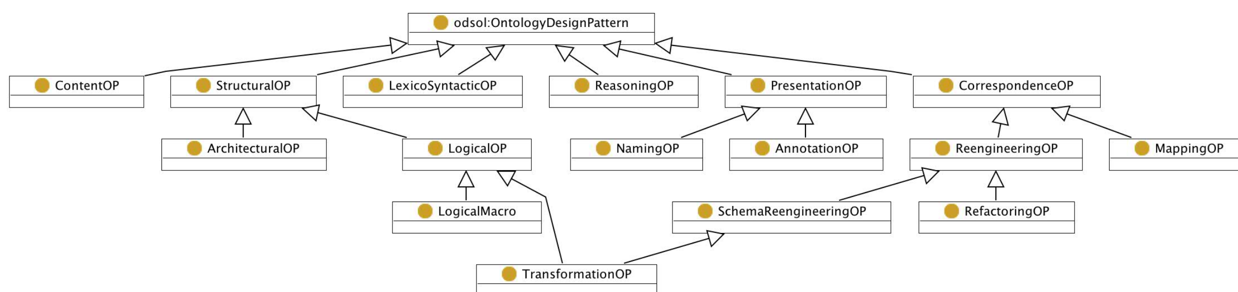


Figure 2.2: Ontology Design Pattern types

One might expect OPs to be easily comparable to software engineering design patterns. The same analogy has been done with architecture, linguistics, and other disciplines. Ontology engineering and software engineering show many similarities from the pragmatic viewpoint, but they are quite different from the theoretical viewpoint. Therefore, we use comparisons between ontology engineering and software engineering for clarifying concepts and intuitions behind the definitions, but we do not take theoretical aspects of OP as dependent on those of software engineering (or other fields).

Our concept of "pattern" is associable with that of "good/best practice" in software engineering, hence it includes a wider range of solution types. For example, naming conventions in software engineering are considered good practices, but they are not design patterns.[5] In ontology engineering "naming" is an important design activity, which can have a strong impact on the usage of the ontology e.g., for selection, mapping, etc. That's why we classify ontology naming conventions as OPs.

In order to prevent confusion, we distinguish the different types of OPs by grouping them into six families (cf. Figure 2.2): *Structural* OPs, *Correspondence* OPs, *Content* OPs (CPs), *Reasoning* OPs, *Presentation* OPs, and *Lexico-Syntactic* OPs . Each family addresses different kinds of problems, and can be represented with different levels of formality.

Structural OPs can be either Logical OPs or Architectural OPs. Correspondence OPs can be either Reengineering OPs, or Mapping OPs. CPs can be distinguished in terms of the domain they represent. Reasoning OPs are typical reasoning procedures. Presentation OPs relate to ontology usability from a user perspective; e.g., we distinguish between Naming OPs and Annotation OPs. Lexico-Syntactic OP are linguistic structures or schemas that permit to generalize and extract some conclusions about the meaning they express.

Each family of OPs can be associated with a catalogue. This deliverable presents a catalogue of CPs (chapter 4) that will be extended in next versions, and a sample catalogue including entries for each type of OPs (a preliminary catalogue of Logical OPs is also contained in [SFBG+07].[6] The next version of this deliverable will also contain at least a catalogue of Reasoning OPs.)

As a graphical representation of Logical OPs and CPs, we use an OWL profile for UML, where UML generalization (an arrow with a large end) is used to represent `owl:subClassOf`, UML association (an arrow with a small end) is used to represent `owl:ObjectProperty`, UML class (a large box) is used to represent `owl:Class`, and a special, UML-object-like box is used to represent `owl:Restriction`.

2.2.1 Structural OPs

Structural OPs includes Logical OPs and Architectural OPs. Logical OPs are compositions of logical constructs that solve a problem of expressivity, while Architectural OPs are defined in terms of composition of Logical OPs that are used in order to affect the overall shape of the ontology; i.e., an Architectural OP identifies a composition of Logical OPs that are to be exclusively used in the design of an ontology.

[5] Actually, software engineering design patterns are always associated to naming conventions, which can be then considered as part of the design pattern.

[6] The definition of Logical OP has changed from the time of writing of [SFBG+07], however some of the Logical OPs included there still comply with the new definition given in this deliverable.

Logical Ontology Design Patterns

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 Logical OPs 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. A Logical OP can be applied more than once in the same ontology in order to solve the same modeling problem. In this context we deal with OWL DL and we define Logical OPs as follows.

Definition 12 (Logical Ontology Design Pattern) *A logical design pattern is a formal expression, whose only parts are expressions from the logical vocabulary of OWL DL, that solve a problem of expressivity.*

We can informally divide Logical OPs into two types:

- *Logical macros* compose OWL DL constructs; e.g. the **universal+existential** OWL macro, described in [Vra05], allows to model a recurrent intuitive quantification, as from the following example: “all horses have parents that are horses”, requiring both an *owl:allValuesFrom* restriction (only horses can be parents of horses) and a *owl:someValuesFrom* restriction (all horses have parents).
- *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 [NA05] with respect to OWL, is a transformation pattern from first-order logic to OWL DL. Other Logical OPs are documented in [SFBG$^+$07, NA05].

In this section, we include four samples of Logical OPs: **n-ary relation**, **classes as values**, **transitive reduction**, and **relation between datatypes**. Each description is accompanied by two diagrams.

- The first diagram shows the meta model elements needed for representing the Logical OP in OWL DL. Such elements are defined as an extension of the OWL ODM ontology, which encodes OWL DL constructs in a metamodel[7]. Such ontology is referred to by the prefix `a`.
- The second diagram shows an example of usage for the Logical OP.

N-ary relation

The aim of this Logical OP is to solve the issue of modeling n-ary relations in an ontology expressed in OWL DL, which only natively supports the expressivity for binary relations. Figure 2.3 shows a diagram depicting the metamodel extension for the **n-ary relation** pattern. This Logical OP consists of defining a class that represents a relation with $n_{\geq 1}$ arguments. Such a class is an instance of the metamodel class `NaryRelationClass`. Although applicable to relations with any arity, `NaryRelationClass` is typically instantiated for relations with three or more arguments. An instance of `NaryRelationClass` has to be the domain of object properties that represent the projections of the n-ary relation, i.e., instances of the metamodel class `NaryRelationProjection`. `NaryRelationProjection` instances are object

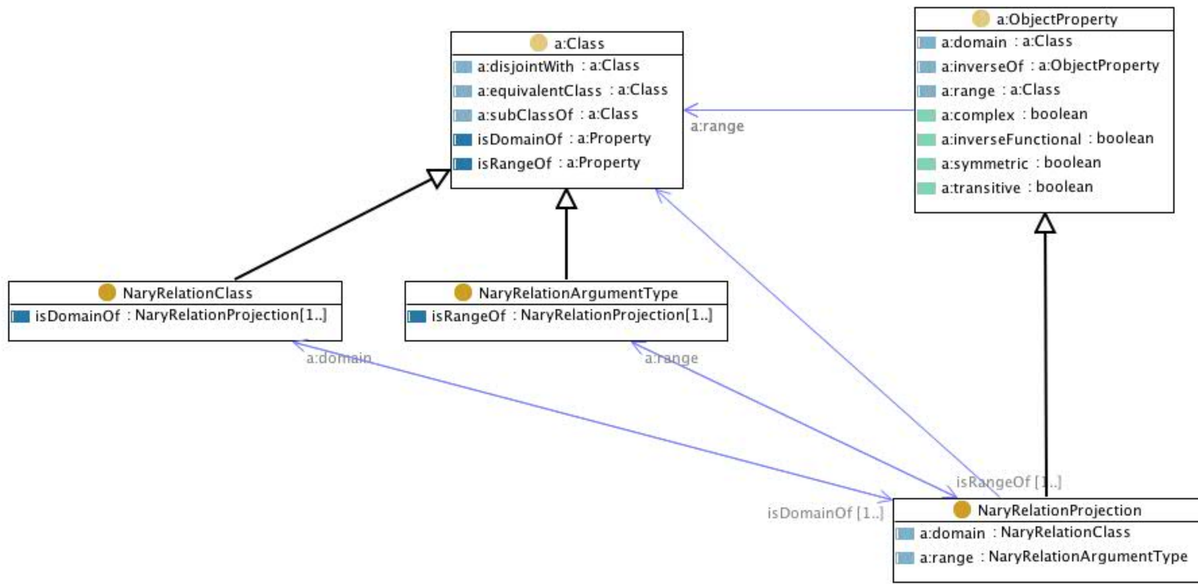


Figure 2.3: The n-ary relation Logical OP

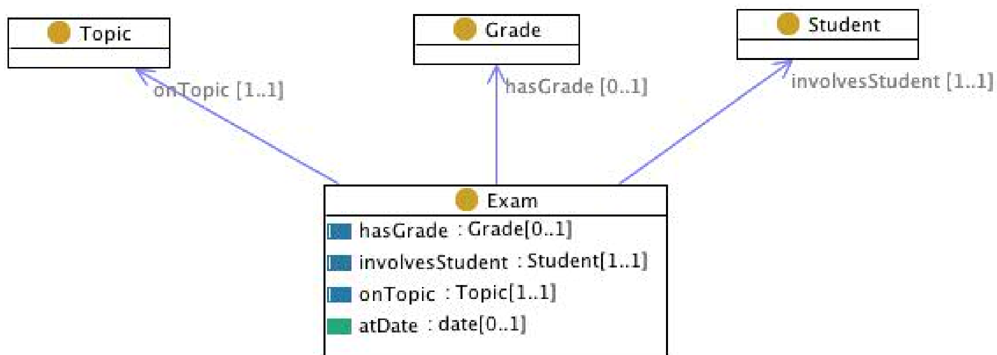


Figure 2.4: An example of usage of the n-ary relation Logical OP

properties that only can have as their range a class of the type `NaryRelationArgumentType`, which represents the reification of the type of the arguments from a n-ary relation.

An example is the *exam* relation, which involves four arguments: the topic of the exam, the student that gives the exam, the grade that the student obtains, and the date at which the exam take place. Figure 2.4 depicts the relation *exam* represented by the **n-ary relation** Logical OP. We introduce the class `Exam` that represents the relation. For each argument of the relation we introduce either an object or a datatype property. For each object property we define a class that represents the type of the value for the object property range. For the example, we introduce the classes `Topic`, `Grade`, and `Student`, and the object properties `onTopic`, `hasGrade`, and `involvesStudent`. Finally, the `atDate` datatype property is used to express the date of the exam. All object properties and the datatype properties are constrained to have one at most or exactly one value for a given exam. Several CPs in chapter 4 (e.g. situation, time-indexed part of) instantiate this Logical OP.

[7]The OWL ODM ontology is available at http://www.loa-cnr.it/codeps/owl/owl10a.owl, which is an extension/revision of http://owlodm.ontoware.org/OWL1.0.owl

2006–2008 © Copyright lies with the respective authors and their institutions.

Classes as values

The aim of this Logical OP is to express that a class is the value for a relation. This is not possible in OWL DL, since a class cannot be part of the ABox. Figure 2.5 shows a diagram depicting the metamodel extension for the **classes as values** Logical OP. The metamodel class `ReifiedClass` includes the reifications of classes that are introduced in a model. An individual from `ReifiedClass` is the subject of an `owl:hasValue` restriction that has to be defined in order to relate the class to its reification. Such a restriction is a class of type `HasClassAsValue`. The individual that reifies the class can be used as representing the value for the class.

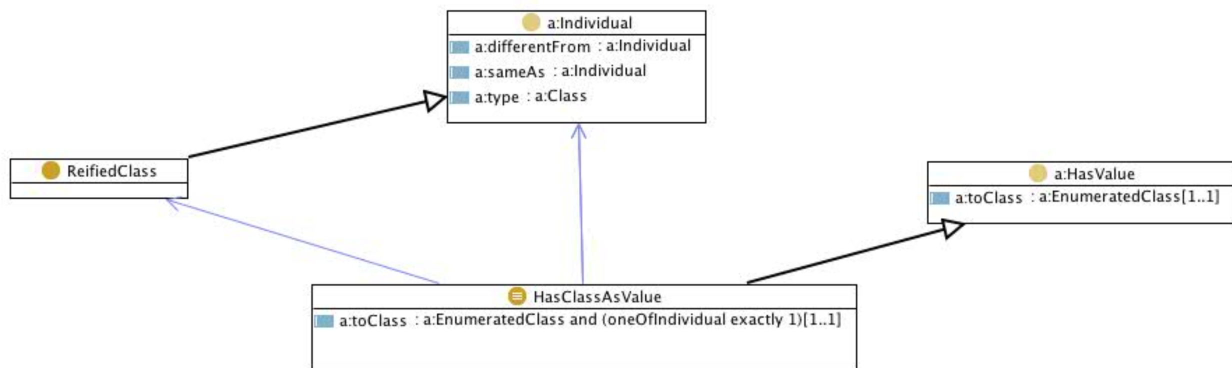


Figure 2.5: The classes as values Logical OP

For example, consider an ontology that introduces a class representing the students of a university, while you want to assert that the subject of a certain regulation are "the students". Figure 2.6 depicts how to solve this design issue by adopting the **classes as values** Logical OP. The class `Student` is the class of all the students of the university, while the class `Subject` models the subject of a regulation e.g., the collection of all students from the university. The class `Regulation` represents the regulations of the university. The `isAssociatedWithSubject` object property associates each element of a certain class with a qualified instance of the class `Subject`, by means of a `owl:hasValue` restriction. The object property `hasSubject` associates a regulation with its subject, which uniquely identifies a class of the model. In the case of students, this can be e.g. the individual `students`, that represents students as an individual from the class `Subject`.

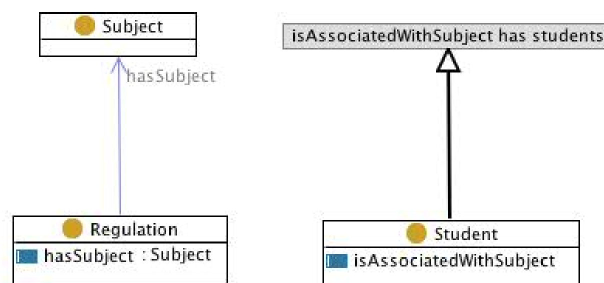


Figure 2.6: An example of usage of the classes as values Logical OP

Several CPs in chapter 4 (e.g. classification, agent-role) instantiate this Logical OP.

Transitive reduction

The aim of the **transitive reduction** Logical OP is to represent the transitive reduction of an object property in an OWL model, without defining that object property as transitive. This is obtained by defining the object

property as sub-property of another object property, which is a transitive property. The result is that the transitive closure of the sub property is the transitive closure of its super property. Figure 2.7 shows a diagram depicting the metamodel extension for this Logical OP. The class `TransitiveProperty` is the set of all transitive object properties. The class `TransitivelyReducedProperty` represents the object properties, the transitive closure of which is given by the transitive closure of one or more instances of `TransitiveProperty`. As an example of application of this pattern, the reader can refer to e.g. the `componency` CP in section 4.2.3. Consider also the following as an example of requirements for a domain

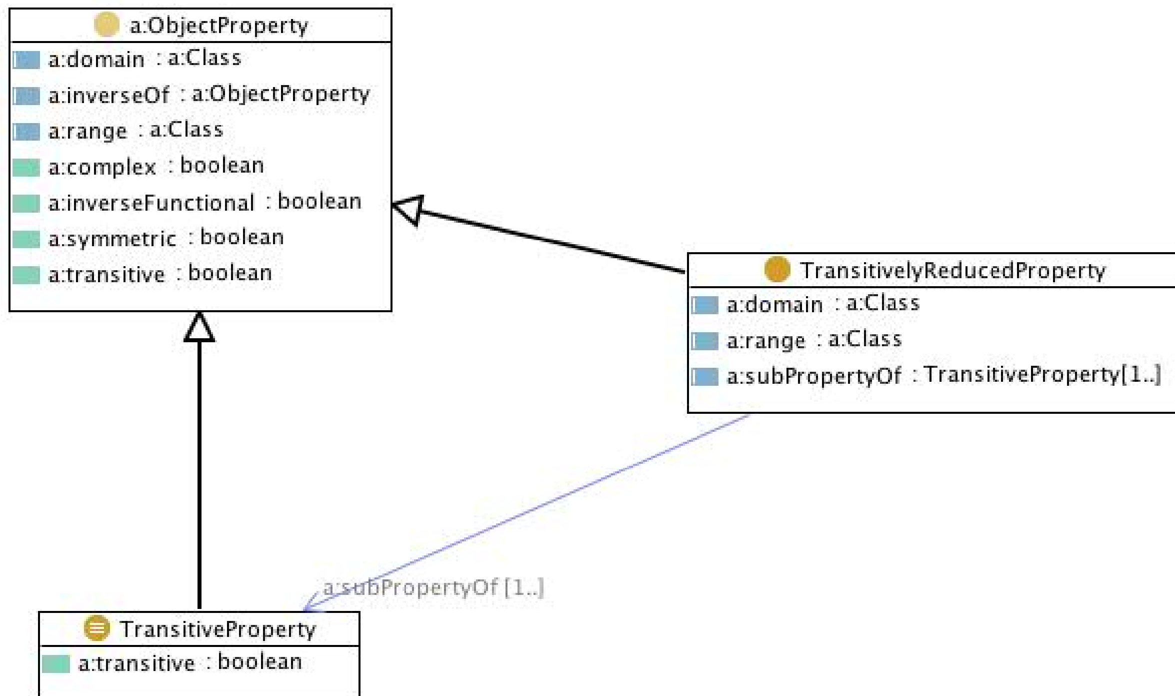


Figure 2.7: The transitive reduction Logical OP

problem: a project is composed only of procedures, which in turn are composed only of activities. Activities of a certain procedure are said to be part of the project the procedure is a component of. Certain responsibilities and roles apply only to components of the project, while others apply also to sub-components. Now, consider activities *a1* and *a2* that compose the procedure *pr1*, and the activities *a3* and *a4* that compose the procedure *pr2*. The project *p* is composed of *pr1* and *pr2*.
If we apply the transitive reduction pattern to componentOf relation as sub-property of the partOf relation, with partOf as transitive and componentOf as transitively reduced, the previous assumptions let us infer that *pr1* and *pr2* are part of *p* as well as *a1*, *a2*, *a3*, and *a4*. Furthermore, *a1* and *a2* are part of *pr1*, and *a3* and *a4* are part of *pr2*. Figure 2.8 depicts an ontology that addresses the above requirements, by using the **transitive reduction** Logical OP. We define the classes `Project`, `Procedure`, and `Activity` that correspond to the domain entities expressed by requirements. The `partOf` object property is transitive, while the `componentOf` object property is defined as subproperty of `partOf` and does not inherit the characteristic of being transitive. The transitive reduction of `componentOf` is the transitive closure of `partOf`.

Relation between datatype values

The aim of this Logical OP is to relate two datatype values, a relation that is not expressible in OWL DL. As an example, consider the need of modeling the fact that a certain data value is *preferred to* another in

2006–2008 © Copyright lies with the respective authors and their institutions.

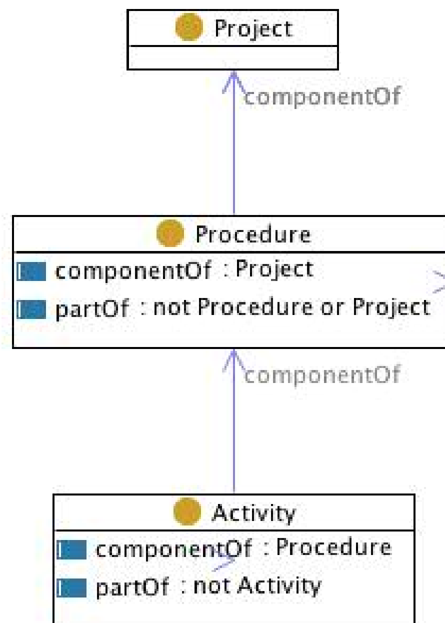


Figure 2.8: An example of usage of the transitive reduction Logical OP

a certain context. Figure 2.9 shows a diagram depicting the metamodel extension for this Logical OP. The instances of the metamodel class `DataValueClass` are classes defined in the model in order to represent certain data values. Such classes are in the domain of one instance of `RepresentsDataValue`, which is the class of the datatype properties that relate classes of type `DataValueClass` with the value they are meant to represent. Finally, `RelationBetweenDataValue` is the class of object properties that relate two classes of type `DataValueClass` and that express the relation between two datatype values.
A variant of this Logical OP is to have a `NaryRelationClass` that reifies the `RelationBetweenDataValue` (see above), when the data values to be related are more than two. For example, consider you want to represent that a certain name is preferred to another one by a couple who is waiting for a baby. Figure 2.10 shows an ontology that addresses this requirement. We define the class `Name`, the instances of which represent names. Each of them is related to a `xsd:string`. At the bottom of the figure there is an example of instantiation. `name1` is associated with the string `Linda`. while `name2` to the string `Arianna`, respectively. The `preferredTo` relationship enables us to express that Linda is preferred to Arianna.

Architectural Ontology Design Patterns

Architectural OPs affect the overall shape of the ontology: their aim is to constrain ‘how the ontology should look like’. Architectural OPs are defined in terms of Logical OPs, or compositions of them. They are used in the design of an ontology as a whole, by providing the composition of Logical OPs that have to be exclusively employed when designing an ontology. Examples of Architectural OPs are: **Taxonomy**, **Modular Architecture**, **Lightweight Ontology**, etc. They have been presented in [SFBG$^+$07]. OWL species as well as the different varieties of description logics are other examples of Architectural OP, as they describe a specific set of logical constructs that can be used in defining the ontology.
For example, the **Modular Architecture** Architectural OP consists of an ontology network, where the involved ontologies play the role of modules (according to definitions given in [HBP$^+$07] and in section 3.3. The modules are connected by the *import* operation[8] with one root ontology that imports all the modules.

[8] See section 3.2.6

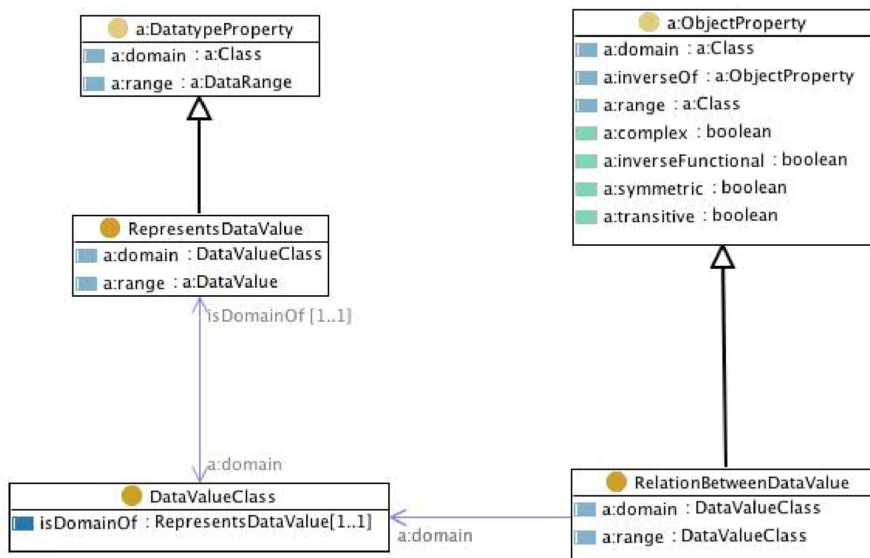


Figure 2.9: The relation between datatype values Logical OP

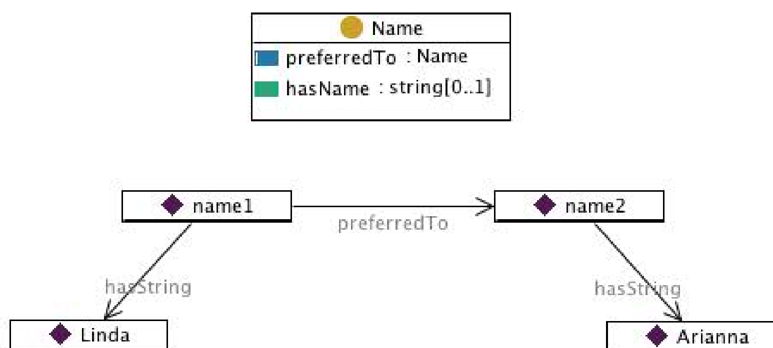


Figure 2.10: An example of usage of the relation between datatype values Logical OP

Figure 2.11 shows the structure of a modular architecture ontology.

2.2.2 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 Ontology Design Patterns

Reengineering Ontology Design Patterns (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.

2006–2008 © Copyright lies with the respective authors and their institutions.

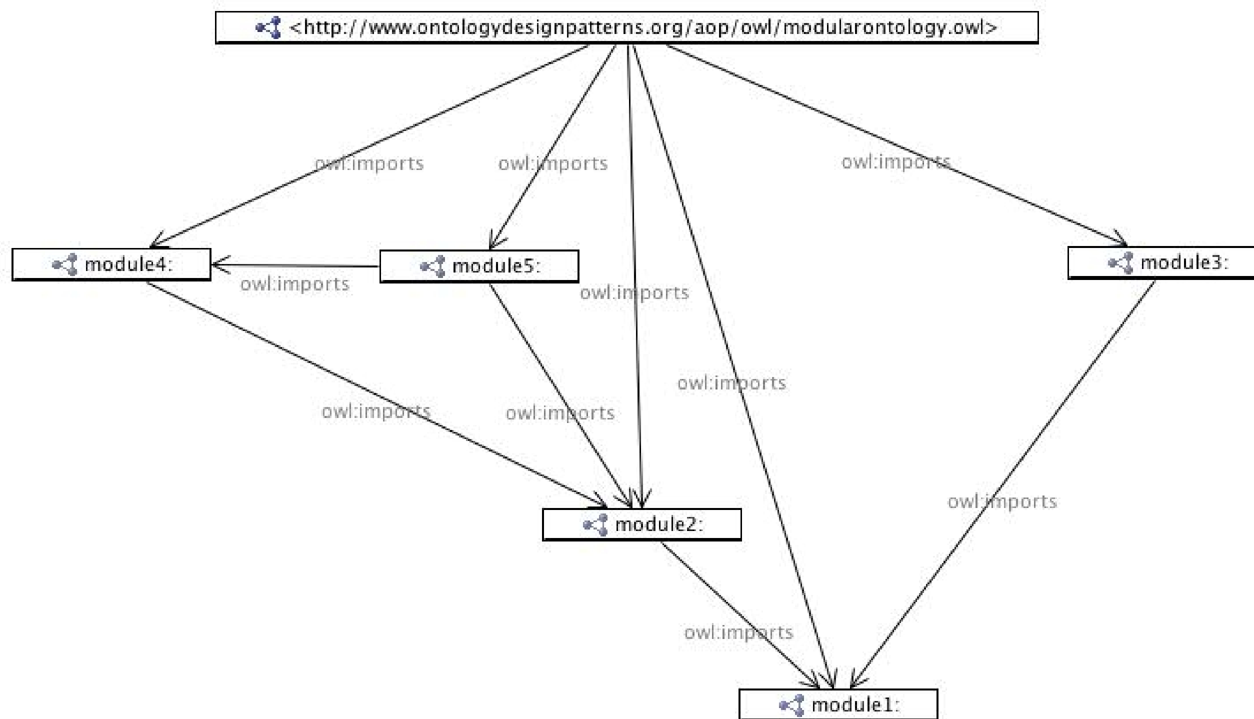


Figure 2.11: The modular architecture Architectural OP

- *Schema reengineering patterns* are rules for transforming a non-OWL DL metamodel into an OWL DL ontology. For example, consider the use of SKOS [MB05] for Knowledge Organization Systems (KOS) reengineering. The following set of rules describes a simple example of Reengineering OP for conversion of a KOS to a knowledge base (an OWL ABox), based-on the SKOS [MB05] TBox. The symbol "↦" is to be read "is transformed to".

  kos2skosABox :

$$\texttt{KOS} \mapsto \texttt{skos:ConceptSchema} \tag{2.1}$$

$$\texttt{Descriptor} \mapsto \texttt{skos:Concept} \tag{2.2}$$

$$\texttt{Broader Term} \mapsto \texttt{skos:broader} \tag{2.3}$$

$$\texttt{Related Term} \mapsto \texttt{skos:related} \tag{2.4}$$

  The rule (2.1) states that, given a KOS, it maps to an instance of the class `skos:ConceptSchema`, while the rule (2.2) maps each descriptor in the KOS to a specific instance of the class `skos:Concept`. The rule (2.3) relates to the case of having two descriptors `d1` and `d2` in the KOS, where `d1` is broader than `d2`. Given the corresponding instances of `skos:Concept` `skos:c1` and `skos:c2`, the *broader term* relationship between `d1` and `d2` maps to an object property value having the subject `skos:c1`, the object property `skos:broader`, and the object `skos:c2`. Finally, the rule (2.4) relates to the case of having two descriptors `d1` and `d2` in the KOS that are related terms. Given the corresponding instances of `skos:Concept` `skos:c1` and `skos:c2`, the *related term* relationship between `d1` and `d2` maps to an object property value having the subject `skos:c1`, the object property `skos:related`, and the object `skos:c2`.

  Transformation Logical OPs (2.2.1) 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 OWL DL,[9] while in the second case the designer wants to reengineer an existing non-OWL DL model into an OWL DL ontology.

- *Refactoring patterns* provide designers with rules for transforming, i.e. “refactoring”, 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 pattern.

Mapping Ontology Design Patterns

Mapping OPs refer to the possible semantic relations between mappable elements, as defined in [HBP$^+$07]. 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.
We also consider an additional semantic relation that we call *clonedness*, which states that the ontology element oe_1 in one ontology is the clone of an ontology element oe_2 in the other ontology. Depending on which of the two elements is the cloned one, *clonedness* can be expressed in one or the other direction. Such a semantic relation holds e.g., between an ontology element of a CP and an ontology element of the reference ontology, from which it has been cloned. More details about the *clone* operation that produces such a semantic relation between ontology elements is contained in section 3.2.2. As CPs play the role of modules with respect to the ontologies that reuse them, *clonedness* should be considered in the definition of the semantics for modular ontologies in NeOn[10].
Pragmatically speaking, clonedness is different from mapping, because clonedness is a relation emerging when *generating* a new ontology as a partial copy of the first, while mapping is established between two existing (and usually mutually independent) ontologies.

2.2.3 Reasoning OPs

At the time of writing we have investigated Reasoning OPs only at a preliminary level, although we plan to deepen this branch of OP-related work. A catalogue of Reasoning OPs could be included in the next version of this deliverable. 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 [VS07, VSPS07] five normalizations have been identified, which are summarized as follows:[11]

[9] 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.
[10] Deliverable D1.1.3 and later versions
[11] For more details the reader can refer to [VS07, VSPS07]

2006–2008 © Copyright lies with the respective authors and their institutions.

- Name all relevant classes, so no anonymous complex class descriptions are left (restriction de-anonymizing)
- Name anonymous individuals (skolem de-anonymizing)
- Materialize the subsumption hierarchy (automatic subsumption) and normalize names
- Instantiate the deepest possible class or property
- Normalize property instances (property value materialization)

When all known Reasoning OPs have been performed on an ontology, it can be considered as *normalized*.

2.2.4 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 patterns by facilitating their evaluation and selection.

Naming Ontology Design Patterns

Naming OPs are conventions about how to create names for namespace, files, and ontology elements in general (classes, properties, etc.). Naming OPs are good practices that boost ontology readability and understanding by humans, by supporting homogeneity in naming procedures. Examples of Naming OPs include:

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.

Class names: should not contain plurals, unless explicitly required by the context. Names like `Areas` is considered bad practice, if e.g. an instance of the class `Areas` is a single area, not a collection of areas. It is also recommended to use *readable* names instead of e.g. alphanumerical codes. It is also useful to include the name of the parent class as a *suffix* of the class name e.g. `MarineArea rdfs:subClassOf Area`. Furthermore, class names conventionally start with a capital letter e.g. `Area` instead of `area`.

Property names: usually start with a lower case letter e.g. `specializes` instead of `Specializes`.

Annotation Ontology Design Patterns

Annotation OPs provide annotation properties or annotation property schemas that can be used in order to improve the understandability of ontologies and their elements. Examples of Annotation OPs include:

labels: each class and property should be annotated with meaningful labels with the annotation property `rdfs:label`, with also translations in different languages.[12]

comments: each ontology and ontology element should be annotated with the rationale they are based on by means of the annotation property `rdfs:comment`, or other user-defined annotation properties. This information is crucial for “manual” selection and evaluation.

[12] More advanced practices for multilingual in ontologies are described in deliverable D2.4.1 from this workpackage.

2.2.5 Lexico-Syntactic OPs

Lexico-Syntactic OPs can be defined as 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. In this work, Lexico-Syntactic OPs have been formalized basing on some notations used for describing the syntax of languages, like the BNF notation, and on the way in which those patterns have been represented in previous studies as [Hea92, SJN04, VVS06]. The elements represented in the formalized patterns are considered to be necessary for identifying the relation of interest supposedly expressed by the pattern. For example, in one of the patterns that corresponds to the "subClassOf" relation, NP<subclass> be NP<superclass>, a Noun Phrase (abbreviated NP) should appear before the verb -represented by its basic form or lemma, be in this example- and the verb should in its turn be followed by another Noun Phrase. In this way, sentences in English like “Dolphins are warm blooded mammals” could be asserted of expressing a hyponymy-hyperonymy relation between the two Noun Phrases.
A Noun Phrase is a phrase whose main word is a noun or a pronoun, and that is optionally accompanied by a set of modifiers, as for example, determiners, adjectives, etc. Following the previous example, warm blooded mammals would constitute a NP. NPs can represent classes, properties or relations. This information has been linked to the NP for the sake of clarity. In some Lexico-Syntactic OPs we have even differentiate between super- and sub-class. Verbs expressing the conceptual relation in question are represented by its lemma or base form, be in the case of the Dolphins example. In principle, the verb form that we encounter in NL sentences should not modify the meaning of the pattern.
The elements represented in the pattern are the ones considered to be necessary for the pattern to express a certain relation. Optional elements, i.e. the ones that may appear or not without modifying the basic meaning of the pattern, have been indicated by the use of [ ], as included in Table 2. Any other element should in principle be ignored, because it does neither provide any information nor affect or modify the basic meaning expressed by the sentence in NL. Tables 1 and 2 show the rest of abbreviations and symbols used in the formalization of the Lexico-Syntactic OPs. Some abbreviations have been extracted from the annotation system of the Penn Treebank Project [San91] that corresponds to the part-of-speech tags obtained by annotating NL texts with the GATE tool.
The identification of Lexico-Syntactic OPs associated with OPs and their exploitation is an on-going research that, up to now, has resulted in the first version of a set of Lexico-Syntactic OPs related to some OWL constructs, and some Logical OPs. For describing such linguistic information we use the following template:

Lexico-Syntactic Patterns	
Formalization	NL representations of the ODP in the form of formalized patterns
Examples	NL expressions matching the Lexico-Syntactic OP in question

Abbreviations used in the Lexico-Syntactic OPs and their meaning	
CD	Cardinal Number [San91]
CATV	Verbs of classification {classify in/into, comprise in, contain in, compose of, group in/into, divide in/into, fall in/into, belong to}
CN	Class Name: generic name for class type plus preposition {class of, group of, type of, member of, subclass of, category, etc.}
NEG	No, not
NP	Noun Phrase

2006–2008 © Copyright lies with the respective authors and their institutions.

PARA	Paralinguistic symbol, such as colon (:).
PREP	Preposition
RPRO	Relative Pronoun {that, which, whose. . . }
VB	Verb, base form [Hea92]

Table 1: Abbreviations used in the formalization of Lexico-Syntactic OPs

Other symbols	
( )	Group of two or more elements
*	Repetition
[ ]	Optional element
¬	Element that should not appear in the specified position in the pattern

Table 2: Symbols used in the formalization of Lexico-Syntactic OPs

To date, we have identified Lexico-Syntactic OPs (for the English language) for the following OWL DL constructs: subClassOf, equivalence between classes, object property, subpropertyOf relation, datatype property, existential restriction, universal restriction, disjointness, union of classes. Furthermore, we have identified Lexico-Syntactic OPs for the following Logical OPs[13]: **multiple-inheritance**, **exhaustive classes**, and **n-ary relation**.

Lexico-Syntactic OP associated with rdfs:subClassOf

Lexico-Syntactic OP	
Formalization	NP<subclass> be NP<superclass> [(NP<subclass>,)* and] NP<subclass> be [CN] NP<superclass> [(NP<subclass>,)* and] NP<subclass> (group in | into | as) | (fall into) | (belong to) CN NP<superclass> NP<superclass> CATV [CD] [CN] [PARA] (NP<subclass>,)*and NP <subclass>
Examples	Amphibians are divided into three groups: frogs and toads, newts and salamanders, and caecilians.

[13]The reader can refer to [SFBG+07] for the Logical OPs that are not included in this deliverable

Lexico-Syntactic OP associated with multiple inheritance between owl:Class(es)

<table>
<tr><th colspan="2">Lexico-Syntactic OPs</th></tr>
<tr><td>Formalization</td><td>[(NP<subclass>,)* and] NP<subclass> be | CATV [CN] NP<superclass> and NP<superclass></td></tr>
<tr><td>Examples</td><td>Amphibians are water-living and land-living animals.</td></tr>
</table>

Lexico-Syntactic OP associated with owl:equivalentClass

<table>
<tr><th colspan="2">Lexico-Syntactic OP</th></tr>
<tr><td>Formalization</td><td>NP<class> be (the same as) | (equivalent to) | (equal to) | like NP<class>

NP<class> call | denominate | designate by|as | name NP<class>

NP<class> have (the same) | equal characteristic | feature | attribute | quality | property as) NP<class></td></tr>
<tr><td>Examples</td><td>4,2 onces are equivalent to 120 grams
Poison dart frogs are also called poison-arrow frogs.</td></tr>
</table>

Lexico-Syntactic OP associated with owl:ObjectProperty

<table>
<tr><th colspan="2">Lexico-Syntactic OP</th></tr>
<tr><td>Formalization</td><td>NP<class> have NP<class> [1]

NP<class> VB ¬ (be | have | CATV) NP<class></td></tr>
<tr><td>Examples</td><td>Birds have feathers.
Birds build nests</td></tr>
</table>

[1] Ambiguous pattern. In some cases, it can also express the Logical Pattern for Modelling Datatype Property or the Simple Part-Whole Relation Content Pattern, amongst others. Research in this sense is currently being conducted.

Lexico-Syntactic OP associated with rdfs:subPropertyOf

2006–2008 © Copyright lies with the respective authors and their institutions.

Lexico-Syntactic OP	
Formalization	NP<property> specialize | (be more specific than) NP<property> NP <property> contain | include | (be subproperty of) NP<property>
Examples	Feathers include contour feathers.

Lexico-Syntactic OP associated with owl:DatatypeProperty

Lexico-Syntactic OP	
Formalization	NP<class> have NP<property> [1] NP<class> have [CARD] property|ies | characteristic|s | attribute|s PARA (NP<property>,)* and NP<property> (NP<property>,)* and NP<property> be property|ies | characteristic|s | attribute|s of [CN] NP<class> NP<class> be describe | define by (NP<property>,)* and NP<property>
Examples	Some unique properties of mammals are hair, sweat glands, producing milk, and giving live birth.

[1] Ambiguous pattern. (cf. 1.2.4)

Lexico-Syntactic OP associated with owl:someValuesFrom (existential restriction)

Lexico-Syntactic OP	
Formalization	NP<class> [must] have NP<class> * Every | any | all NP<class> [must] have NP<class> *
Examples	Every bird has feathers.

Lexico-Syntactic OP associated with owl:allValuesFrom (universal restriction)

Lexico-Syntactic OP	
Formalization	NP<class> [can] just | only | exclusively | entirely have [relation|s to] NP<class> NP<class> VB just | only | exclusively | entirely NP<class>
Examples	Herbivore eat only plants.

Lexico-Syntactic OP associated with owl:unionOf

Lexico-Syntactic OP	
Formalization	NP<class> be union | sum | join of (NP<class>,)* and NP<class> NP<class> (consist of) | (made up of) | comprise | (be constitute of) [PARA] (NP<class>,)* and NP<class> (NP<class>,)* and NP<class> make up | constitute NP<class>
Examples	The deep-sea consists of four zones: the mesopelagic, the bathypelagic, the abyssopelagic and the hadal zone.

Lexico-Syntactic OP associated with owl:disjointWith

Lexico-Syntactic OP	
Formalization	NP<superclass> be | classify in|into | group in | into [either] NP<subclass> or NP<subclass> (NP<class>,)* and NP<class> (have NEG) | (do NEG have) elements | individuals | instances in common (NP<class>,)* and NP<class> do NEG share elements | individuals | instances NP<superclass> be divide | split |separate between | among NP<subclass> and NP<subclass>
Examples	Animals are either vertebrates or invertebrates.

2006–2008 © Copyright lies with the respective authors and their institutions.

Lexico-Syntactic OP associated with Exhaustive Classes Logical OP (LP-EC-01)

Lexico-Syntactic OP	
Formalization	NP<superclass> include | comprise | (be composed of) [all] [CARD] [CN] [PARA] (NP<subclass>,)* and NP<subclass> NP<superclass> divide in|into [CARD] [CN] [PARA] (NP<subclass>,)* and NP<subclass>
Examples	Vertebrates include: mammals, amphibians, reptiles, birds, and fish.

Lexico-Syntactic OP associated with n-ary relation Logical OP (LP-NR-01)

Lexico-Syntactic OP	
Formalization	NP<class> VP NP<class> [PREP] NP<class> NP<class> VP NP<class> RPRO VP NP<class>
Examples	Bird feathers provide birds insulation.

2.2.6 Content OPs (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 [Gan05].
From the theoretical perspective, CPs should be compared to *knowledge patterns*, which are conceived in [CTP00] as axiom schemas that are *invariant under signature morphisms*, i.e. when the predicates in the signature change, the pattern does not change. On the other hand, that definition would make CPs similar to Logical OPs, since no signature change would affect the pattern.
A possible improvement is to require that knowledge patterns have a non-empty signature, since Logical OPs have an empty one. But even with this addition, the unconditional invariance under signature change that is required for knowledge patterns would violate the intuition of CPs. For example, if we consider the **agent-role** CP, which has a signature including the predicates: {Agent, hasRole, Role}, and we change it by preserving *downward taxonomical ordering*, e.g.: {Person, hasRole, Employee}, the invariance of the pattern is ensured, because persons are agents and employee is a kind of role. But if we change it in less intuitive ways, by violating downward taxonomical ordering, or even giving up taxonomical ordering at all, e.g.: {Object, hasRole, Object} (upward taxonomical ordering), or {Nose, hasRole, Water} (no taxonomical ordering), the invariance of the pattern is lost.[14] For these reasons, we restrict the definition in [CTP00] for CPs, as *being invariant under signature morphisms that preserve downward taxonomical ordering*.
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). They are typically reused by performing specialization, expansion, and composition (see chapter

[14] Unless one wants to use a pattern-based ontology to achieve unusual or creative effects, as in poetry.

3).
In analogy to conceptual modeling (cf. the difference between class and use case diagrams in the Unified Modeling Language (UML) [Obj04]) and knowledge engineering (cf. the distinction between domain and task ontologies in the Unified Problem-solving Method development Language (UPML) [ML00]), modeling problems solved by CPs 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 "expert finding" scenario). A typical way of capturing use cases is by means of so called *competency questions* [GF94].
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.
Based on the above assumptions, we also characterize CPs as distinguished ontologies. They cover a specific set of competency questions (requirements), which represent the problem they are a solution for. A CP is associated with the set of ontology elements it consists of. Furthermore, CPs show certain characteristics i.e., they are: computational, small and autonomous, hierarchical, cognitively relevant, linguistically relevant, and constitute good practices for domain experts (these characteristics are explained in some detail in section 27).

CPs can be also described in terms of the relation they can have with Logical OPs, as depicted in Figure 2.12.
On the left side, LP is the set of all Logical OPs, while $\overline{\mathrm{LP}}$includes all valid combinations of Logical OPs. On

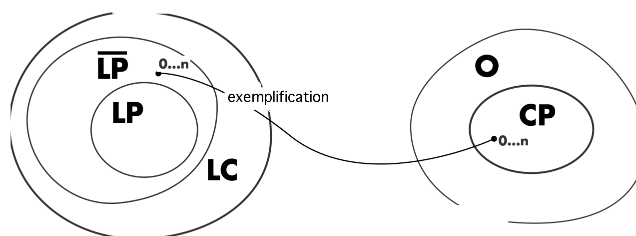


Figure 2.12: Relation between CPs and Logical OPs.

the right side of the picture, the set O identifies all possible ontologies, and CP identifies the set of all CPs. Our intuition is that there is a relation, which we call *exemplification*, between elements of CP and elements of $\overline{\mathrm{LP}}$, with cardinality (0...n) on the $\overline{\mathrm{LP}}$side, and cardinality (0...n) on the CP side. Informally, exemplification corresponds to instantiation of Logical OPs, where CPs are the instances.
The next two chapters focus on CPs. Chapter 3 contains a more precise definition of CP, and describes how CPs are created and used. Chapter 4 contains a catalogue of CPs that is structured according to the different domains they address issues for.

2006–2008 © Copyright lies with the respective authors and their institutions.

Chapter 3

Creation and Usage of Content Ontology Design Patterns

This chapter better characterizes Content OPs, and can be used as an introduction to create, manipulate, and apply them. The operations on CPs are described in section 3.2. A more precise definition of CP is given in section 2.2.6. Section 3.4 describes four approaches for the creation of CPs, while section 3.6 describes the possible situations of CP selection and usage that can occur in practice.
Definitions in this chapter are enriched (where possible) with a formal semantics that is based on the assumption that with any ontology O_1 (then also for each CP CP_1), we can associate two unique sets: the set OE_{O1} (OE_{CP1}) of all ontology elements $oe_{1...n}$ from O_1 (CP_1), and the set AX_{O1} (AX_{CP1}) of all axioms $ax_{1...n}$ from O_1 (CP_1).
CPs are components that represent, and possibly help solving a modelling problem (requirement, use case) arising across different scenarios. The terms *requirement*, *use case*, and *scenario* are used in ontology design in order to express concepts similar to those referred by the same terms when they are used in software engineering. However, there are some differences that we highlight in the next section in order to prevent confusion. Furthermore, the concept of requirement is key for defining the notion of CP.

3.1 Requirements and use cases

A software engineering *requirement* describes a user functionality that a software system under development has to address. The term requirement is also used for the document that describes user requirements, which is usually written by a software engineer using a natural language. A typical requirements document describe functionalities that may be redundant, overlapping, contradictory, or even incompatible. Software engineers use such document in order to perform a *requirement analysis* that results in the production of a *system specification*. Typically, contradictory and incompatible problems can be discovered and fixed by analyzing the system specification. Unfortunately, system specifications are often represented by using languages that do not have a complete formal semantics. Therefore, finding such problems is one of the trickiest issues in software engineering.
A *use case* represents a typical interaction between a user and a system in order to accomplish a task. It describes the possible ways that allow a user to complete that task as lists of steps, some of which are carried out by the system without the need of user interaction. Use cases encompass also the possibilities of errors and failures of the system.
Finally, a *scenario* instantiates a use case, and is based on a real case example.
Ontology engineering requirements are expressed in terms of *competency questions* (*cqs*), which express classes of queries that users might want to submit to a knowledge base. Cqs can be expressed in different ways; the most common is a set of natural language questions, which show the intended use of the system.

Requirements can be either asserted, or automatically extracted from natural language documents by exploiting Natural Language Processing (NLP) [] techniques. Questions can also be either transformed to or directly expressed as SQL-like queries by following query matching techniques.
Since competency questions are elicited by users or extracted by corpora, they can be affected by the same problems as software engineering requirements. However, in ontology engineering the analogous of software system specification is a formal theory, i.e. an ontology. This makes the task of discovering either contradictions or incompatibilities in the ontology equivalent to performing a consistency checking operation, which is included in most reasoning engines[1].
Given the above assumptions, definition 13 introduces the notion of *ontology requirement*.

Definition 13 (Ontology Requirement, Req) *A Requirement is a non-empty set of competency questions cq that are each other neither contradictory nor incompatible, while they can be either overlapping or redundant.*

$$\begin{aligned} Req \equiv \{cq_1, \ldots, cq_n\} \neq \emptyset \\ cq_i \neq cq_j, \forall i \neq j \end{aligned} \tag{3.1}$$

The notion of *use case* in ontology engineering is equivalent to that of Req, whereas a *scenario* is intended as a possible state that can be represented in the knowledge base. Intuitively, Req refers to the theory (the TBox), while scenarios refer to the instances (the ABox).

3.2 Operations

CP creation and usage rely on a common set of operations: *covering* that connects CPs and requirements, *clone* that copies an ontology element from an existing ontology, *composition* between CPs, *specialization* and *generalization* that define a partial ordering over CPs, and *expansion*, a relation between CPs on one side, and either ontology elements (oe) or axioms (ax) on the other side. The operations are defined below.

3.2.1 Covering

The *covering* operation has the purpose of expressing that a CP satisfies all cq_i in a requirement.

Definition 14 (Covering, $cov(CP, Req_1)$) *Let's Req_1 be a set of competency questions, which corresponds to a number of possible queries $q_1, \ldots, q_n$ to be submitted to a knowledge base kb. A CP covers Req_1 if it is as expressive as it is needed to store the necessary knowledge for answering $q_1, \ldots, q_n$.*

For example, consider the following competency questions:

- cq_1: who did play a specific role in a certain period?
- cq_2: which role does have a certain person at a certain time?

Let's define $Req = \{cq_1, cq_2\}$. Now consider two CPs: **agent role**, and **time indexed person role**. They are described in sections 4.6.2, and 4.6.4, respectively. For the sake of readability we show here the diagrams that describe them. Our aim is to verify if the following statements hold.

$$cov(\textbf{agent role}, Req) \tag{3.2}$$
$$cov(\textbf{time indexed person role}, Req) \tag{3.3}$$

[1]A similar approach is used in software engineering model checking techniques, but they are mostly used in academic contexts and for systems dealing with sensitive information and time constraints.

2006–2008 © Copyright lies with the respective authors and their institutions.

Figure 3.1 shows the UML diagram of the **agent role** CP. Such CP allows us to represent agents, the roles they play, and the relation between them. Persons can be modeled by specializing the class `Agent`. However, both competency questions ask for information about temporal indexing, which is not included in this CP. Therefore, the statement (3.2) is false.

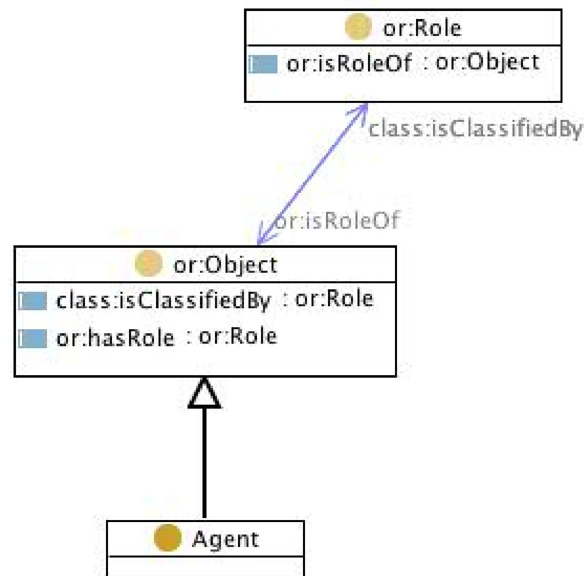


Figure 3.1: The agent role CP

Figure 3.2 shows the UML diagram of the **time indexed person role** CP. Such CP allows us to represent persons, the roles they play, and the time interval at which the relation between them holds. This CP models all the knowledge we need in order to answer cq_2 as well, therefore the statement (3.3) is true.

3.2.2 Clone

The *clone* operation consists of duplicating an ontology element, which is used as a prototype.[2] We distinguish three kinds of clones:

- *shallow clone*: consists of creating a new ontology element oe_2 by duplicating an existing ontology element oe_1. OWL restrictions of and axioms defined for oe_1 and oe_2 will be exactly the same
- *deep clone*: consists of creating a new ontology element oe_2 by duplicating an existing ontology element oe_1, and by deep-cloning a new ontology element for each one that is referred in oe_1's axiomatization, recursively
- *partial clone*: consists of deep-cloning an ontology element, but by keeping only a subset of its axioms, and of partial-cloning the kept elements, recursively.

Some ontology design tools support the shallow clone operation e.g., TopBraid Composer[3], while deep clone and partial clone are not yet supported by any existing tool.

Definition 15 (Shallow Clone) *Given an ontology element oe_1, shallow clone creates a new ontology element oe_2, which is the copy of oe_1, and assigns a new name to it. The ontology element oe_1 plays the role of* prototype*.*

[2]There is a strong analogy between the clone operation in object oriented software programming and ontology element clone operation
[3]http://www.topbraidcomposer.com/

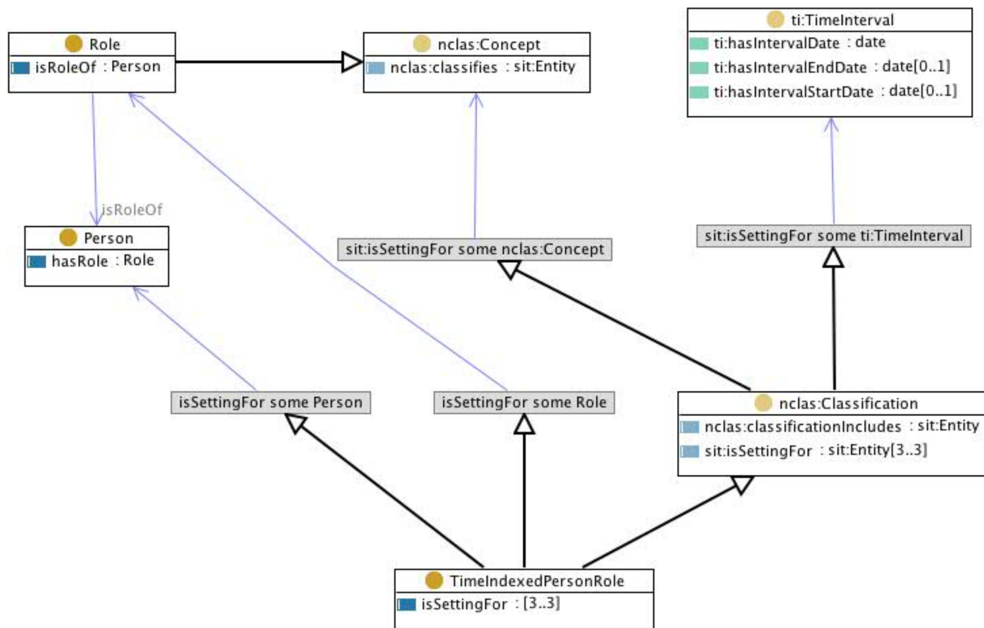


Figure 3.2: The time indexed person role CP

For example, suppose we want to perform a shallow clone of the class `DUL:Agent`. The axiomatization of `DUL:Agent` is shown in Figure 3.3.[4]. When we perform a shallow clone of the class `DUL:Agent`, the result is the creation of a new class. We assign to the new class the name `AgentClone`. Figure 3.4 depicts

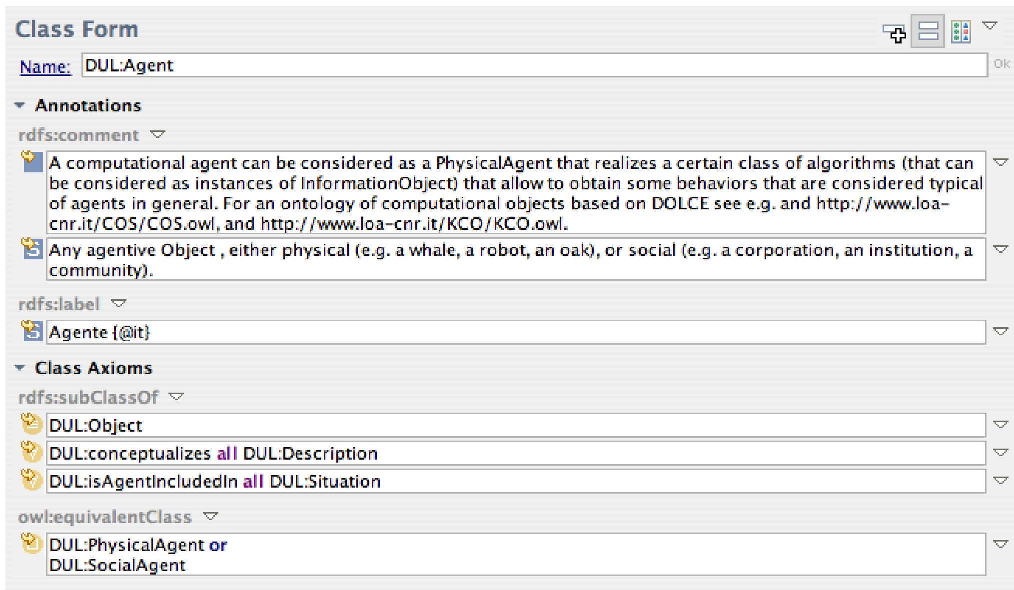


Figure 3.3: The definition of the `DUL:Agent` class.

the axiomatization of the class `AgentClone`. As the reader can notice, the class `AgentClone` has the same semantics as the `DUL:Agent` i.e., the OWL axioms defined for it involve the same ontology elements as those defined for `DUL:Agent`

Definition 16 (Deep Clone) *Given an ontology element oe_1, deep clone creates a new ontology element oe_2, which is the copy of oe_1, and assigns a new name to it. Furthermore, all ontology elements that are*

[4]The figure depicts a screenshot of the ontology editor with the detail of the axioms defined for the class `DUL:Agent`

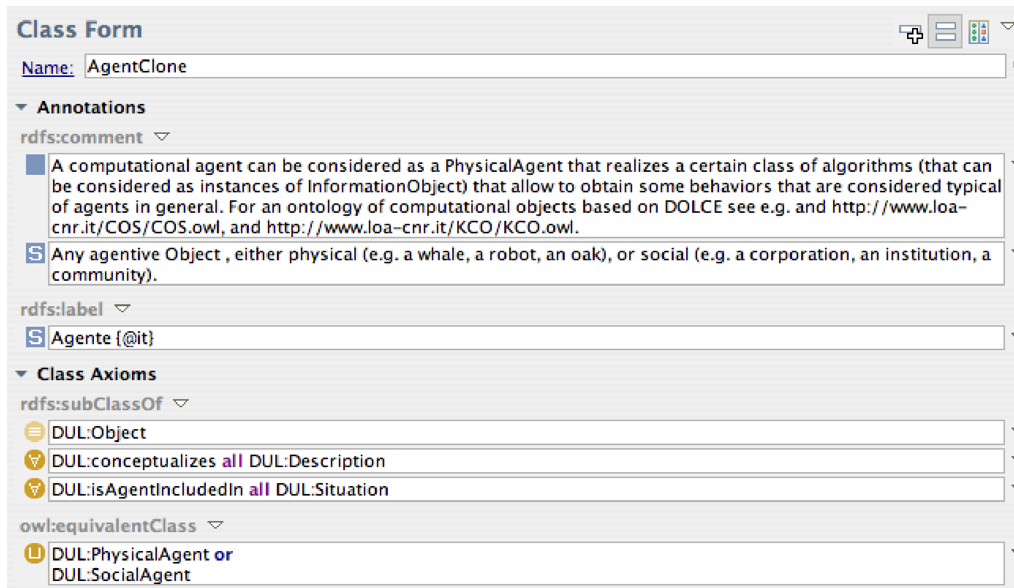


Figure 3.4: The definition of the `AgentClone` class.

involved in the axiomatization of oe_1 are cloned by means of a deep clone, recursively. All axioms of the copies are updated in order to involve the copies instead of the prototypes. The result is a new ontology that does not have any explicitly formalized dependence on the first one.

In order to better understanding how deep clone works, the reader can refer to the mechanism of object-oriented deep clone, typically used in the *Prototype* software engineering design pattern [GHJV95].

Definition 17 (Partial Clone) *Given an ontology element oe_1, partial clone creates a new element oe_2, which is the copy of oe_1, and assign a new name to it. The oe_1 ontology element is used as a prototype for a deep clone based on a selected set of axioms from oe_2. The other axioms are not copied in oe_2. All the ontology elements involved in the recursive process are copied by means of partial clone.*

For example, consider a partial clone of the class `DUL:SocialAgent`. The axiomatization of `DUL:SocialAgent` is shown in Figure 3.5. We decide to select and keep two restrictions from the prototype. Such restrictions are identified in the figure by the orange arrows. Figure 3.6 depicts the ontology that results from the partial clone of `DUL:SocialAgent`. The ontology is composed of the following ontology elements:

- `SocialAgentClone`, which is the partial clone of `SocialAgent`.
- `PhysicalAgentClone`, which is the partial clone of `PhysicalAgent`.
- `isActedByClone`, which is the partial clone of `isActedByClone`.

These three ontology elements are exactly those involved in the axioms we selected for the partial clone of `SocialAgent`. It can be noticed that the definition of the class `SocialAgentClone` consists of the two previously selected restrictions (updated in order to involve only local ontology elements). In order to define the other ontology elements, the same rules are followed. Section 3.4 shows how partial clone is used in the process of extracting a pattern from a reference ontology.

Figure 3.5: The definition of the `DUL:SocialAgent` class.

3.2.3 Composition

The *composition* operation relates two CPs and results into a new ontology. The resulting ontology is composed of the union of the ontology elements and axioms from the two CPs, plus the ontology elements and axioms[5] that are added in order to link the CPs.
We define an *identity element* for the composition operation i.e., the empty CP.

Definition 18 (Empty CP, CP_0) *The empty CP is identified by $CP_0 \equiv \emptyset$*

Definition 19 (Composition, $cmp(CP_1, CP_2)$) *The composition of CP_1 and CP_2 consists of creating a semantic association between CP_1 and CP_2 by adding at least one new axiom, which involves ontology elements from both CP_1 and CP_2. The composition operation has the following semantics:*
Let $OE_{CP1} \equiv \{oecp1_1, \ldots, oecp1_n\}$ be the set of ontology elements from CP_1, $AX_{CP1} \equiv \{axcp1_1, \ldots, axcp1_m\}$ be the set of axioms of CP_1, $OE_{CP2} \equiv \{oecp2_1, \ldots, oecp2_k\}$ be the set of ontology elements from CP_2, $AX_{CP2} \equiv \{axcp2_1, \ldots, axcp2_q\}$ be the set of axioms of CP_2, $OE \equiv \{oe_1, \ldots, oe_r\}$, and $AX \equiv \{ax_1, \ldots, ax_s\}$ be the sets of ontology elements and axioms (respectively) used for composing CP_1 with CP_2. Now we can introduce:

$$
\begin{aligned}
O_{CP1CP2} &= cmp(CP_1, CP_2) \\
OE_{OCP1CP2} &\equiv (OE_{CP1} \cup OE_{CP2} \cup OE) \\
AX_{OCP1CP2} &\equiv (AX_{CP1} \cup AX_{CP2} \cup AX)
\end{aligned}
\tag{3.4}
$$

Furthermore, if $Req_3 \equiv Req_1 \cup Req_2$, then

$$cov(CP_1, Req_1), cov(CP_2, Req_2) \Rightarrow cov(O_{CP1CP2}, Req_3) \tag{3.5}$$

And, when composing with the empty CP

$$cmp(CP, CP_0) = CP \tag{3.6}$$

[5] typically at least an OWL axiom

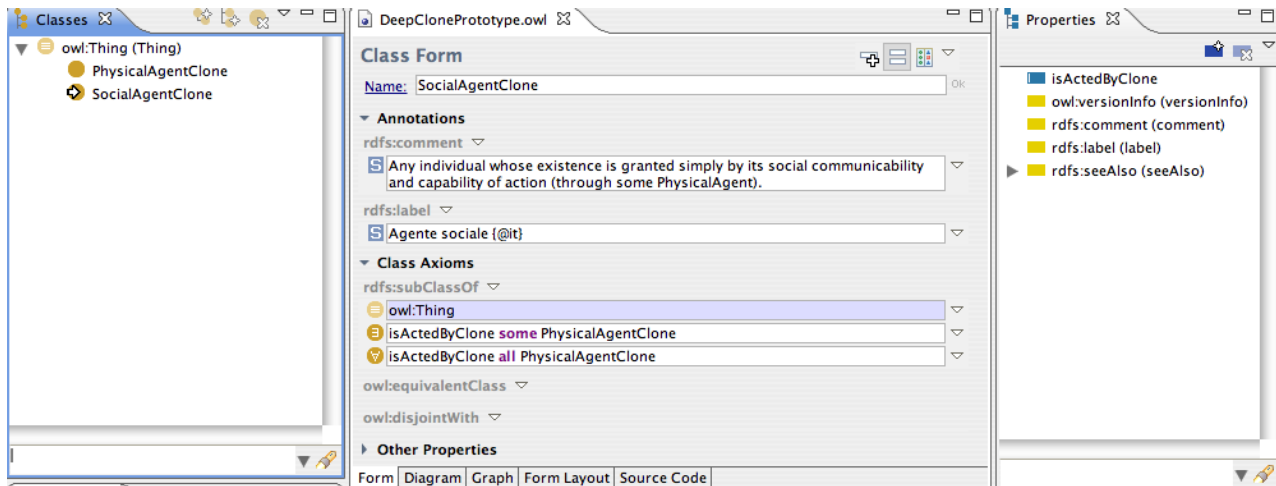


Figure 3.6: The result of a partial clone of the `DUL:SocialAgent` class.

With the previous axioms, we state that the composition between two CPs involves at least one new axiom. This is due to the fact that each CP has its own namespace. For example, consider the two CPs: CP_1 and CP_2. CP_1 includes an axiom for the class `cp1:Agent` to be a subclass of the class `cp1:Object`. CP_2 includes an axiom for the class `cp2:Object` to be a subclass of the class `cp2:Entity`.
Assuming that the intension of `cp1:Object` is the same as the intension of `cp2:Object`, $cmp(CP_1, CP_2)$ must include an OWL axiom that states the equivalence between the two classes. This means that at least one additional OWL axiom is needed.
An objection could be: why does not CP_1 directly include `cp2:Object`? In this case the composition would not need any additional OWL axiom. The answer is that that case is not a composition, but a *specialization*, which is defined in the next section.

As an example of composition, let us consider the following competency questions.

- cq_1: Who has this role?
- cq_2: Who has a suitable role in order to perform that task?
- cq_3: Which role is assigned this task to?

The CP catalogue contains the **agent role** CP such that $cov($**agent role**$, cq_1)$ and the **task role** CPs such that $cov($**task role**$, cq_3)$. Sections 4.6.2, and 4.6.3 describe **agent role** CP and **task role**, respectively. For the sake of readability, we include their UML diagrams in Figures 3.7, and 3.8, respectively.

Diagram: The first CP (agent role) allows to represents agents, the roles they play, and the relations between them. The second CP (role task) allows to represent roles, the task they are assigned to, and the relations between them. In order to cover cq_2, we need an ontology that contains both CPs. Figure 3.9 shows the diagram of the ontology that results from the composition of **agent role** and **task role** CPs. The prefixes `or:` and `agentrole:` are to be associated with the **agent role** CP[6], while the prefix `taskrole:` identifies the namespace of the **task role** CP. The composed ontology consists of all the ontology elements and axioms of both CPs plus an additional OWL axiom, which states the equivalence between the class `or:Role`, and the class `taskrole:Role`.

[6]The prefix `agentrole:` identifies the namespace of the **agentrole** CP, while the prefix `or:` identifies the namespace of the **object role** CP that is imported by **agent role**.

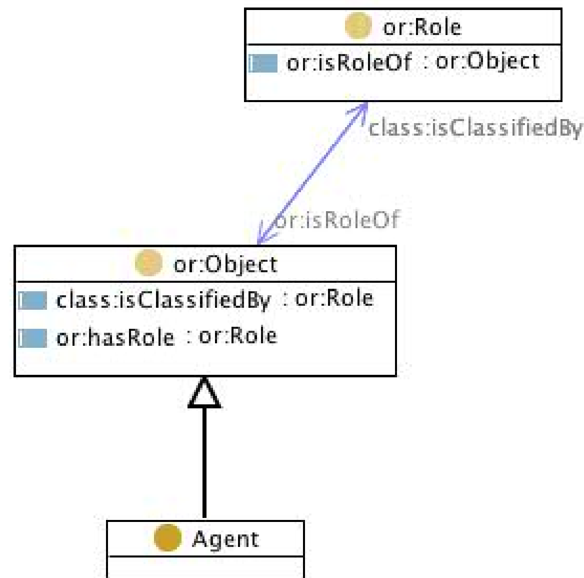


Figure 3.7: The agent role CP

Figure 3.8: The task role CP

3.2.4 Specialization and Generalization

Specialization and generalization introduce a partial order between CPs, which is defined in terms of their taxonomical order. The subsumption relation between ontology elements of two CPs determines which of the two CPs is more or less general than the other one. Specialization and generalization only rely on `rdf:subClassOf`, and `owl:subPropertyOf` OWL axioms, and always have a CP as result.

Definition 20 (Specialization, $spc(CP_1)$) *A content pattern CP_2 specializes CP_1 if at least one ontology element of CP_2 is a taxonomical subelement, i.e., it is either a `rdfs:subClassOf` or an `rdfs:subPropertyOf`, of an ontology element of CP_1. The specialization operation has the following semantics.* Let $OE_{CP1} \equiv (oecp1_1, \ldots, oecp1_n)$ *be the set of ontology elements of CP_1, and $OE_{CP2} \equiv (oecp2_1, \ldots, oecp2_k)$ be the set of ontology elements of CP_2.* Then

$$CP_2 = spc(CP_1) \Leftrightarrow \exists oecp1_i \in CP_1, oecp2_j \in CP_2 \mid oecp2_j \sqsubseteq oecp1_i \quad 1 \leq i \leq n1, 1 \leq j \leq n2 \tag{3.7}$$

Definition 21 (Generalization, $gen(CP_1)$) *A content pattern CP_2 generalizes CP_1 if at least one ontology element of CP_1 is a taxonomical subelement, i.e., it is either a `rdfs:subClassOf` or an*

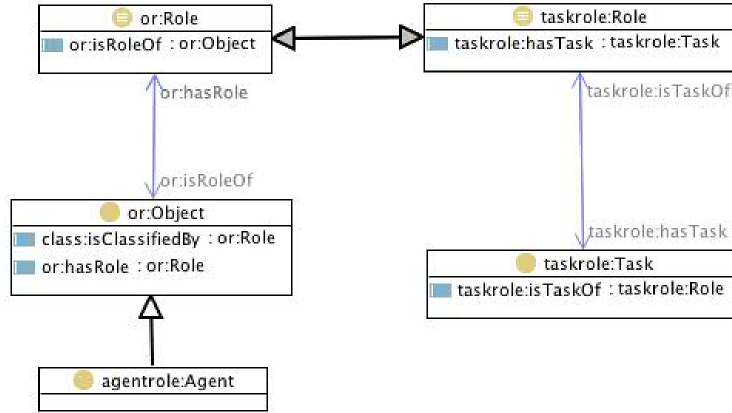


Figure 3.9: The composition of **agent role** and **task role** CPs

`rdfs:subPropertyOf`, *of an ontology element of* CP_2. *The generalization operation has the following semantics.* Let $OE_{CP1} \equiv (oecp1_1, \ldots, oecp1_n)$ *be the set of ontology elements of* CP_1, *and* $OE_{CP2} \equiv (oecp2_1, \ldots, oecp2_k)$ *be the set of ontology elements of* CP_2. Then

$$CP_2 = gen(CP_1) \Leftrightarrow \exists oecp2_i \in CP_2, oecp1_j \in CP_1 \mid oecp1_j \sqsubseteq oecp2_i \\ 1 \leq i \leq n2, 1 \leq j \leq n1 \quad (3.8)$$

3.2.5 Expansion

The *expansion* operation relates a CP to a set of ontology elements, and a set of axioms. It consists of adding new ontology elements and axioms to a CP. The resulting ontology is composed of the ontology elements and axioms of the CP, plus the added ontology elements and axioms.
We define two *identity elements* for the expansion operation i.e., the empty sets OE_0 and AX_0.

Definition 22 (Empty OE, OE_0) *The empty set of ontology elements is identified by* $OE_0 \equiv \emptyset$

Definition 23 (Empty AX, AX_0) *The empty set of OWL axioms is identified by* $AX_0 \equiv \emptyset$

Definition 24 (Expansion, $expn(CP_1, OE, AX)$) *Given a CP* CP_1 *such that* $cov(CP_1, Req_1)$, *a set of ontology elements* OE, *and a set of OWL axioms* AX, *the expansion of* CP_1 *by means of* OE *and* AX *consists of creating a new ontology* O *which contains all the ontology elements and axioms of* CP_1, OE, *and* AX, *and such that* $cov(O, Req_1)$. *Let* $OE_{CP1} \equiv \{oecp1_1, \ldots, oecp1_n\}$ *be the set of ontology elements of* CP_1 *such that* $OE_{CP1} \neq OE$, $AX_{CP1} \equiv \{axcp1_1, \ldots, axcp1_m\}$ *be the set of OWL axioms from* CP_1 *such that* $AX_{CP1} \neq AX$. *Let finally* OE_O *be the set of ontology elements from* O, *and* AX_O *be the set of OWL axioms from* O. *Then we define:*

$$O = expn(CP_1, OE, AX) \Leftrightarrow \\ \exists oe \in OE, \exists ax \in AX \mid oe \notin OE_{CP1} \wedge ax \notin AX_{CP1} \wedge \\ \nexists CP2 \neq CP1 \mid oe \in OE_{CP2} \wedge ax \in AX_{CP2} \\ OE_O \equiv OE_{CP1} \cup OE \\ AX_O \equiv AX_{CP1} \cup AX$$

Where $OE_{CP2} \equiv \{oecp2_1, \ldots, oecp2_k\}$ *and* $AX_{CP2} \equiv \{axcp2_1, \ldots, axcp2_q\}$ *are the set of ontology elements and the set of axioms from the CP* CP_2*, respectively.*
And, when expanding with the empty set of ontology elements or OWL axioms

$$expn(CP, OE_0, AX_0) = CP \tag{3.9}$$

In practice, CP expansion requires that additional elements and axioms that are added to a CP do not come from other CPs, otherwise those cases are actually compositions.

3.2.6 Import

The basic mechanism for CP reuse is *import*. It is also the only one directly supported in the OWL vocabulary with `owl:import`. Import is applicable to ontologies, hence also to CPs. If an ontology O_2 imports an ontology O_1, all the ontology elements and OWL axioms from O_1 are included in O_2. However, the imported ontology elements and axioms cannot be modified, i.e., the ontology elements and axioms are read-only entities for O_2. By importing a CP, an ontology ensures the set of inferences allowed by the CP in its corresponding knowledge base.

Definition 25 (Import, $import(O_1, CP_1)$**)** *Let* OE_{O1}*, and* AX_{O1} *be the set of ontology elements and the set of axioms of the ontology* O_1*, respectively. Let* OE_{CP1}*, and* AX_{CP1} *be the set of ontology elements and the set of axioms of the CP* CP_1*, respectively. The* import *operation has the following semantics:*

$$\begin{aligned} O_2 &= import(O_1, CP_1) \\ OE_{O2} &\equiv OE_{O1} \cup OE_{CP1} \\ AX_{O2} &\equiv AX_{O1} \cup AX_{CP1} \end{aligned} \tag{3.10}$$

Where OE_{O2} *and* AX_{O2} *are the set of ontology elements and the set of axioms of* O_2*, respectively.*

3.3 Definition of Content Ontology Design Pattern (CP)

A CP is a *networked ontology* that can play the role of a module for another ontology, according to the following definition[7]:

Definition 26 (Networked Ontology) *An ontology that is a member of a network of ontologies. A Network of Ontologies is a collection of ontologies related together via a variety of different relationships such as mapping, modularization, version, and dependency relationships.*

In practice, an ontology is networked when it has some relation to other ontologies.

Definition 27 (Content Ontology Design Pattern, CP**)** *CPs are distinguished networked ontologies and have their own namespace. They cover a specific set* Req *of competency questions (requirements), which represent the problem they provide a solution for. A CP emerges from existing conceptual models and can be extracted from a reference ontology (based on the* clone *operation), can be* reengineered *from other conceptual models (e.g. data models), can be created by* composition *of other CPs, by* expansion *of a CP, and either by* specialization *or* generalization *of another CP. A CP is associated with two sets, which are both unique: the set of its ontology elements, and the set of its OWL axioms. CPs* exemplify *logical design patterns, or some composition of them. Furthermore, CPs show the following characteristics:*

[7] Compliant with C-ODO and [HBP+07]

2006–2008 © Copyright lies with the respective authors and their institutions.

Characteristics of CPs

- *Requirements covering components*. A CP is defined in terms of the requirements it satisfies, which are expressed in terms of competency questions. Formally, given a CP CP_1, $cov(CP_1, Req_1) \rightarrow Req_1 \neq \emptyset$. This aspects is also important for stating the relevance of the CP with reference to a certain domain.

- *Computational components*. CPs have to be represented and encoded in a computational logic i.e., OWL DL in this context, in order to be processed by parsers and automatic reasoners, and to be (re)used as building blocks in ontology design.

- *Small, autonomous components*. Regardless of the particular way a CP has been created (section 3.4 describes ways to create a CP), it is a *small, autonomous* ontology. Smallness 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 either specialized or generalized. A hierarchy of CPs can be built by specializing or generalizing some of the elements (either classes or relations).

- *Cognitively relevant components*. CP visualization must be intuitive and compact, and should catch relevant, “core” notions of a domain. An interesting result from cognitive learning [Gar83] is that the development of expert skills typically “selects” patterns of concepts that are richly interconnected, and in normal cases, these patterns are applied without an explicit reference to the underlying detailed knowledge acquired during the training period.
  This result matches the need to quickly reason or to automatize certain tasks, and the experimental data on short-term memory capacity. For this reason, 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.

- *Reasoning relevant components*. A CP has to allow some form of inference.

- *Linguistically relevant components*. Many CPs nicely match linguistic patterns called *frames*. A frame can be described as a lexically founded ontology design pattern; frames typically encode argument structures for verbs, e.g. the frame **Desiring** associates elements (or “semantic roles”) such as **Experiencer**, **Event**, **FocalParticipant**, **LocationOfEvent**, etc. The richest repository of frames is FrameNet [BFL98]. Frames can be used for validating CPs with respect to lexical coverage, for lexicalizing them, and can be reengineered in order to populate the CP catalogue (cf. [ont]).

- *Best practice components*. A CP should be used to describe a “good practice” of modelling. Good practices are intended here as *local*, thus derived from experts. 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 CP comes from. However, evidence from reusability across different projects, large-scale applications, and open rating systems will provide a good base for CP evaluation.

Section 3.3.1 presents a non-exhaustive list of examples of ontological resources that are not CPs. We also give a definition of Content Ontology Design Anti-Pattern (CAP).

3.3.1 Ontological resources that are not CPs

All ontologies that do not comply to definition 27, hence do not show the CPs characteristics, are not CPs. The most important characteristics that distinguish a CP from an ontology that is not a CP are that they typically neither respond to any competency question, nor allow relevant inferences. In other words, small ontologies that lack such a relevance for a certain domain are not CPs. Examples of small ontologies that cannot be considered CPs are the following:

- Either a single isolated class, or a list of unrelated classes.
- Either a single property with neither range nor domain explicitly defined, or a list of properties like that.
- Either a class with a single subclass, or a non-branching taxonomy.

Pointing out what ontologies are not CPs is useful in order to distinguish what is a Content Ontology Design Anti Pattern (AntiCP) from ontologies that are not CPs.

3.3.2 Content Ontology Design Anti-pattern (AntiCP)

Content Ontology Anti Patterns (AntiCPs) are ontologies that implement bad modeling practices. In other words, they are based on wrong assumptions or rationales. AntiCPs produce the side-effect of inferring wrong or undesired knowledge, or of preventing the capability to infer the desired knowledge.

Definition 28 (Content Antipattern, $AntiCP$) *A content antipattern complies to all CP characteristics except one. It can be distinguished from a CP because its relevance with respect of its specific domain is that it encodes the wrong way to solve the modeling problem. Hence, it is a bad practice component (against the* best practice component *characteristics of CPs).*

It is worth to highlight that AntiCPs should not be confused with CPs that are not suitable in a certain context. In those cases, the matching between the competency questions and the actual use case is wrong. In other words, an AntiCP is always an AntiCP.

A typical example of AntiCP is characterized by using subsumption in order to model part relations. For example, consider the case of an ontology that describes geo-political areas and that addresses a scenario featuring three kinds of areas: nations e.g., Italy, regions e.g., Lazio, and cities e.g., Rome. Nations have control on regions, which in turn have control on cities. The AntiCP is to model the part relation that holds between nations, regions, and cities as a subsumption instead of a partonomy.
Figure 3.10 shows a diagram of an AntiCP for this example. All classes are subclasses of a parent class `Area`. The class that represents nations is modeled as the parent of the class that represents regions, which in turn is the parent of the class that represents cities. This modeling choice is wrong because it produces the side-effect of inferring that e.g., everything that is a city is also a region and a state. This is obviously wrong: nations, states, and cities are disjoint sets, and should be modeled as siblings with a common parent class, e.g. `Area`. The part relation can be modeled as a transitive object property that encodes the part relation. E.g., the **part of** CP is a solution to this modeling issue, and is described in detail in section 4.2.1. Other examples of AntiCPs have been detected during the evaluation of preliminary versions of the FAO ontologies. Such evaluation is described in section 3.6.2. In the literature, [RDH+04] also contains a useful list of AntiCPs.

2006–2008 © Copyright lies with the respective authors and their institutions.

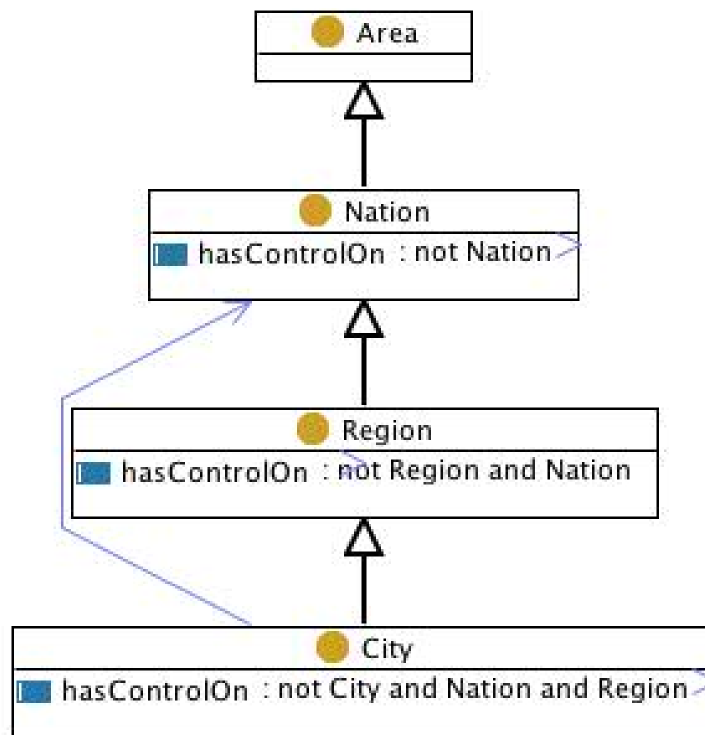


Figure 3.10: Containment vs Subsumption: an example of AntiCP

3.4 Where do CPs come from?

Content ontology design patterns (CPs) come from the experience of ontology engineers in modeling *foundational* [MGG+05], *core* [GB04], or *domain* ontologies. Informally, the distinction between these three kinds of ontologies relates to the degree by which an ontology covers the domain of interest:

- *Foundational* ontologies [MGG+05], like DOLCE [Dol] and SUMO [PNL02], axiomatize general concepts and relations, and are reusable across any domains.
- *Core* ontologies [GB04], like the Core Ontology of Fishery [GFK+04], and the Core Legal Ontology,[8] focus on a domain application without being restricted to specific applications or specific subareas. They can be built in agreement with foundational ontologies or based on general principles and well-founded methodologies.
- *Domain* ontologies, like the Gene ontology,[9] and the Unified Medical Language System (UMLS) [oM03], deal extensively with a specific domain of interest, deepen the coverage on a certain area of a domain, or address a specific use case within a domain.

Assuming those distinctions, there are four 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 conceptual modeling languages, primitives, and styles. [GP07] describes a reengineering approach for creating CPs starting from UML diagrams [Obj04], workflow patterns [VDATHKB03], and data model patterns [Hay96].

[8] http://www.loa-cnr.it/ontologies/CLO/CoreLegal.owl
[9] http://www.geneontology.org/

Other knowledge resources that can be reengineered to produce candidate CPs are database schemas, knowledge organization systems (e.g. thesauri), and lexica (cf. [SAd+07] for reengineering techniques on these resources).

Specialization/Composition of other CPs A CP can be created either by composition of other CPs or by specialization of another CP, (both composition and specialization can be combined with expansion). For example, **nary-participation** CP composes **situation** and **participation** CPs.

Extraction from reference ontologies A CP can be *extracted from* an existing ontology, which acts as the "source" ontology. Extraction of a CP is a process consisting of (partial) cloning the ontology elements of interest from 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. Since it depends on its background, a CP inherits the axiomatization (and the related reasoning service) that is already in place. E.g., the **co-participation** pattern depends on a set of axioms from the DOLCE ontology [Dol], 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 modeler specializes the co-participation pattern 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. Furthermore, a CP can be obtained by partial cloning elements from more than one source ontology. Figure 3.11 shows the typical process that is performed by an ontology engineer for creating a CP by extraction from a reference ontology, possibly including specialization and expansion. Circles with a dashed line refer to specialization and expansion, which could be skipped. The creation of a CP starts with the creation of a new ontology. A suitable namespace is assigned to the new ontology. Each pattern has its own namespace that does not depend on that of the source ontology. The source ontology(ies) is(are) imported. Elements of the source ontology must not be modified. Some tools allow designers to modify imported ontologies, when they are locally stored and writable. In such a case, it is a good practice to lock the imported ontologies in order to set the access permissions as read-only.
The creation proceeds with the partial cloning of the ontology elements of interest. Then, restrictions of the new elements are updated with local references, and if needed, some ontology elements are specialized. At this point, if there are still references to external ontology elements (even to those belonging to source ontologies), they are removed, and possible expansions are performed. Class disjointness is checked. The reasoner is launched to check consistency, and to infer implicit knowledge. Some inferences might be kept and transformed into asserted knowledge. Finally, the imports are removed, and the CP (and its elements) are annotated.
CPs are published on a web portal (ODP) and are annotated with the annotation properties defined in the *CP annotation schema*[10]. The CP annotation schema is presented in section 3.5.

3.4.1 Examples of CP creation

In this section, we show the process of creation of two CPs, which are part of the CP catalogue (chapter 4). First, we describe the creation of the **information realization** CP (see 4.3.2) that is performed by extraction

[10] http://www.ontologydesignpatterns.org/patternannotationschema.owl, currently available at http://wiki.loa-cnr.it/index.php/LoaWiki:patternannotationschema. See section 4.12 for further details about ODP.

2006–2008 © Copyright lies with the respective authors and their institutions.

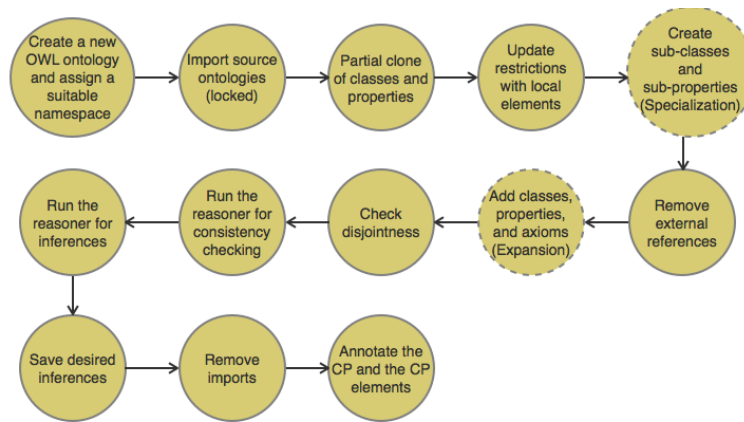


Figure 3.11: The CP extraction process.

from the Dolce Ultra Lite ontology[11]. This CP represents the relations between information objects like poems, songs, formulas, etc., and their physical realizations like printed books, recorded tracks, physical files, etc. Figure 3.12 depicts some screenshots[12] of the ontology editor while we extract the **information realization** CP.
We create a new OWL ontology for the CP named `cps:informationrealization.owl`[13]. We import the source ontology i.e., Dolce Ultra Lite.
In order to highlight the ontology element that we are going to clone, we have drawn some orange arrows over the screenshots. The class we clone from the Dolce Ultra Lite ontology is the one that represents information objects i.e., `DUL:InformationObject`[14]
The upper part of the picture (the first screenshot) depicts the axiomatization of `DUL:InformationObject`, in the bottom left part, the (shallow) clone operation is executed on `DUL:InformationObject`[15]. The result of the shallow clone operation is a new class belonging to the namespace of the information realization CP. We name this class `InformationObject` as it is shown in the third screenshot. Some of the inherited axioms are removed and references to external elements e.g., `DUL:InformationRealization`, are replaced by references to local elements. In practice, `DUL:InformationRealization` is replaced by the local element `InformationRealization`. Note that this action is performed only after we have cloned `DUL:InformationRealization` and have created a local `InformationRealization`. In order to complete the extraction, we clone the two object properties: `DUL:realizes`, and `DUL:isRealizedBy`, and create the local `realizes` and `isRealizedBy` object properties. We remove undesired axioms, and replace external references with local ones.
In the bottom right of the picture, we show the resulting axiomatization of `InformationObject`. Notice also that we have kept the comment. Finally, we have updated the axioms for the `DUL:isRealizedBy` object property, and replaced `DUL:isRealizedBy` with `isRealizedBy`.
We use the same approach for all other ontology elements we want to clone. Once cloning is completed, we remove the import and save the CP. Figure 4.26 in section the 4.3.2 shows a UML diagram of the resulting ontology i.e., the **information realization** CP.
As a second example, we describe the creation of the **time indexed person role** CP[16] that is performed by combination of extraction and specialization. This CP represents the temporal index for the relation between persons and the roles they play.

[11] http://www.loa-cnr.it/ontologies/DUL.owl
[12] The figure contains three screenshots overlapped.
[13] the prefix `cps:` is for http://www.ontologydesignpatterns.org/codeps/owl/
[14] DUL is the prefix for the Dolce Ultra Lite ontology namespace.
[15] The shallow clone operation is implemented by some ontology editors e.g., TopBraid Composer - http://www.topbraidcomposer.com/
[16] See 4.6.4)

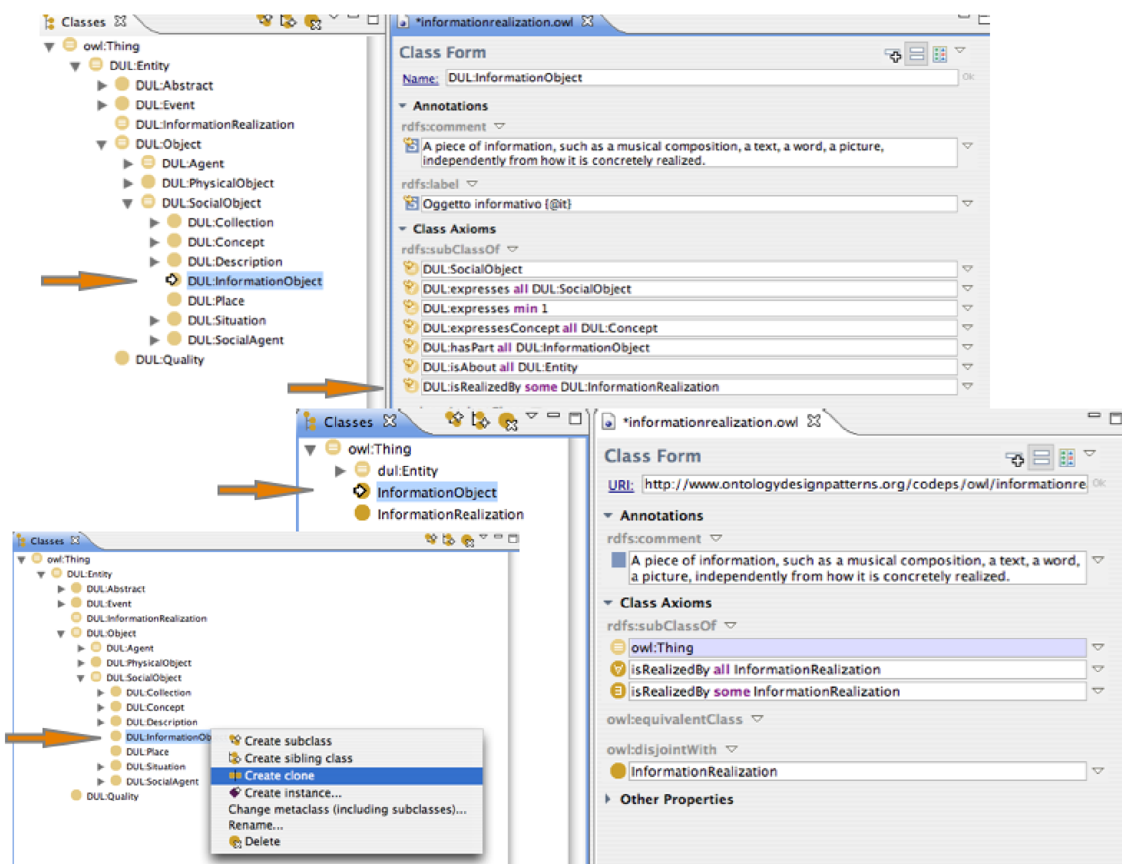


Figure 3.12: The information realization CP extraction from Dolce Ultra Lite ontology.

2006–2008 © Copyright lies with the respective authors and their institutions.

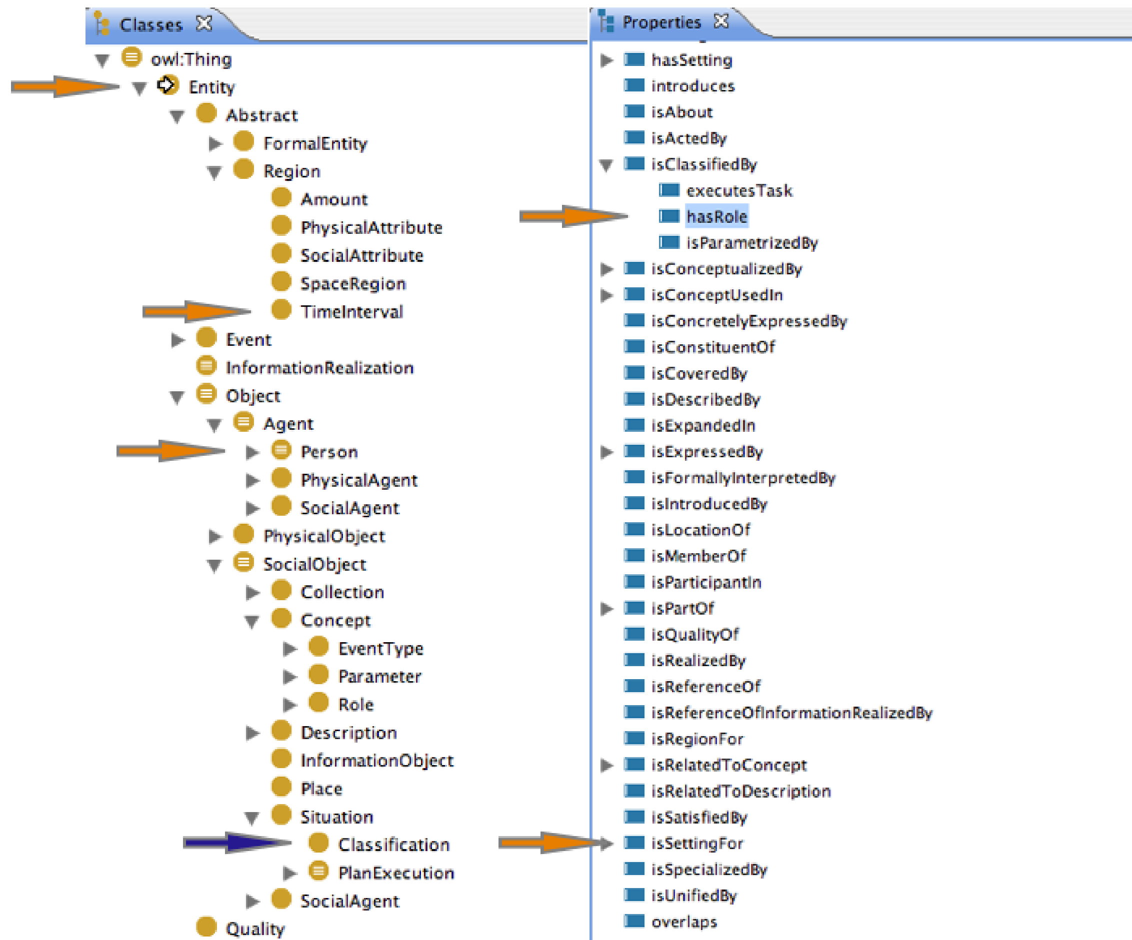


Figure 3.13: Ontology elements extracted and specialized from the Dolce Ultra Lite ontology.

We create a new OWL ontology for the CP, and assign a name to it. We import the Dolce Ultra Lite ontology, that is the source ontology for this CP. Figure 3.13 depicts two screenshots of the *Classes* and *Properties* tabs of the ontology editor. In order to highlight the Dolce Ultra Lite classes and properties that we partially clone in order to create the **time indexed person role** CP, we have drawn some orange arrows. This CP is created by combining extraction and specialization. The class `TimeIndexedPersonRole` is intended to be a specialization of the Dolce Ultra Lite class `DUL:Classification` (pointed by the blue arrow). In practice, we clone `DUL:Classification` and then create a subclass of the clone, which is `TimeIndexedPersonRole`. The other classes and properties are partially cloned with the same approach described above for the **information realization** CP. Figure 4.48 of section 4.6.4 shows a UML diagram of the **time indexed person role** CP. Both **information realization** and **time indexed person role** are described in detail in sections 4.6.4 and 4.3.2, respectively.

3.5 The CP annotation schema

The CP annotation schema has been defined in the context of this deliverable. It consists of a set of annotation properties that are used in order to annotate CPs. All OWL files implementing the CPs presented in chapter 4 are annotated by means of this schema, and any new OWL CP should follow the same specification.

`hasIntent:` This annotation property is used in order to describe the intent of the content pattern.

coversRequirements: this annotation property is used for exemplifying possible requirements the content pattern provides a solution for. Requirements are expressed as natural language competency questions.

scenarios: this annotation property is used for exemplifying possible scenarios the content pattern allows to represent. Scenarios are expressed as natural language sentences.

extractedFrom: this annotation property should be assigned with a URI, which points to the reference ontology which the annotated pattern has been extracted from (i.e. the reference ontology that the ontology elements have been deeply or partially cloned from). The range is not explicit in the definition of the annotation property because it would turn the ontology into OWL Full. E.g. The **participation** CP is extracted from the Dolce Ultra Lite ontology, hence the value for this annotation property is `http://www.loa-cnr.it/ontologies/DUL.owl`.

reengineeredFrom: this annotation property should be assigned with a URI, which points to the concept schema, page, or anything else from which the annotated pattern was reengineered. If the source does not have any URI e.g., a printed book, this property value should provide information as precise as possible in order to identify the source. This property is alternative to the extractedFrom property because it is used when the pattern comes from a concept schema that is not an owl ontology. For example CPs, which are reengineered from data model patterns, rdf schemas, etc. should be annotated with this property. E.g. The **basicpersonalfoaf** pattern is extracted from the rdf FOAF specification, hence the value for this annotation property is `http://xmlns.com/foaf/spec/20071002.rdf`

isSpecializationOf: this annotation property is useful for CPs and its elements. Its value is a URI, which refers either to a CP that is specialized by the annotated one, or to an ontology element that is specialized by the annotated one.

hasComponent: this annotation property is useful for CPs. Its value is a URI, which refers to another CP that is a component of the annotated one.

isCloneOf: this annotation property value should be assigned with a URI, which points to the class or property of the reference ontology, from which the ontology element has been deeply or partially cloned by. Domain and range are not explicit in the definition of the annotation property because this would turn the ontology into OWL Full. The domain is an ontology element i.e., a class or a property. E.g. the Object class of the **object-role** CP has been partially cloned from the dul:Object class of the Dolce Ultra Lite ontology, hence the value for this annotation property in that case is `http://www.loa-cnr.it/ontologies/DUL.owl#Object`. This annotation property is typically used when the CP only includes few clones of classes from a certain ontology. For example, if a CP is reengineered from a non-ontological resource, it is not extracted from any ontology. However, the designer might want to clone some classes from existing ontologies that are suitable for completing the CP.

hasConsequences: this annotation property is used for briefly describing the benefits and/or possible trade-offs when using the CP.

relatedCPs: this annotation property can be used to indicate other CPs (if any) that specialize, generalize, include, or are components of the CP. Furthermore, this field may indicate other CPs that are typically used in conjunction with the described one. Important similarities and differences with other patterns can be also described here.

3.6 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

2006–2008 © Copyright lies with the respective authors and their institutions.

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 typical procedures for ontology selection e.g., [SLM06] and evaluation [GCCJ06].
Informally, the *intent* of a CP must match its use case. Once a CP has been selected, it has to be applied to the design of a domain ontology. Typically, CP application is performed by means of import, specialization, composition, or expansion (see section 3.4). In realistic design projects, such operations are usually combined.

3.6.1 Matching between *intent* and use case

Several situations of matching between the intent of CPs 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 will be available online by end of January 2008. An automatic support to their selection and usage should take into account the principles informally explained in the summary below.

- *Precise matching*. The CP matches the intent, which is directly usable to describe the local use case: the CP has only to be *imported* in the domain ontology. `Expansion` is performed when needed.
- *Broader matching*. The CP matches an intent that is more general than the local use case: the **Related patterns** field of the CP's catalogue entry may contain references 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. `Expansion` is performed when needed.
- *Narrower matching*. The CP matches an intent that is more specific than the local use case: the `odpschema:specializationOf`[17] property of the ODP annotation schema may contain references i.e., URIs, to more general CPs it is the specialization of. The same information is reported in the **Related patterns** field of the CP's catalogue entry. If none of them is appropriate, a new CP should be created for the local use case and *imported*. The new CP will be more general than the one discovered, and they will have to be mutually related. `Expansion` is performed when needed.
- *Partial matching*. The CP partly matches an intent that does not cover all aspects of the local use case (it is simpler): the **Related patterns** field of the CP's catalogue entry may contain 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*. If the actual use case is not too big, it is worth to add a new entry to the catalogue of CPs for the resulting composed CP. `Expansion` is performed when needed.
- *Redundant matching*. The CP matches an intent that is more complex than the local use case: the the `odpschema:hasComponent` property of the OP annotation schema may contain references, to CPs it is composed of, the same information is reported in the **Related patterns** field of the CP's catalogue entry. One of the component CPs could be a precise matching CP for the local use case. If none of the component CPs is adequate, a new CP is extracted from the selected one and associated to it by annotation and by updating the **Related patterns** field. The CP is then *imported*. `Expansion` is performed when needed.

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

[17] odpschema is a prefix for http://www.ontologydesignpatterns.org/patternsannotationschema.owl

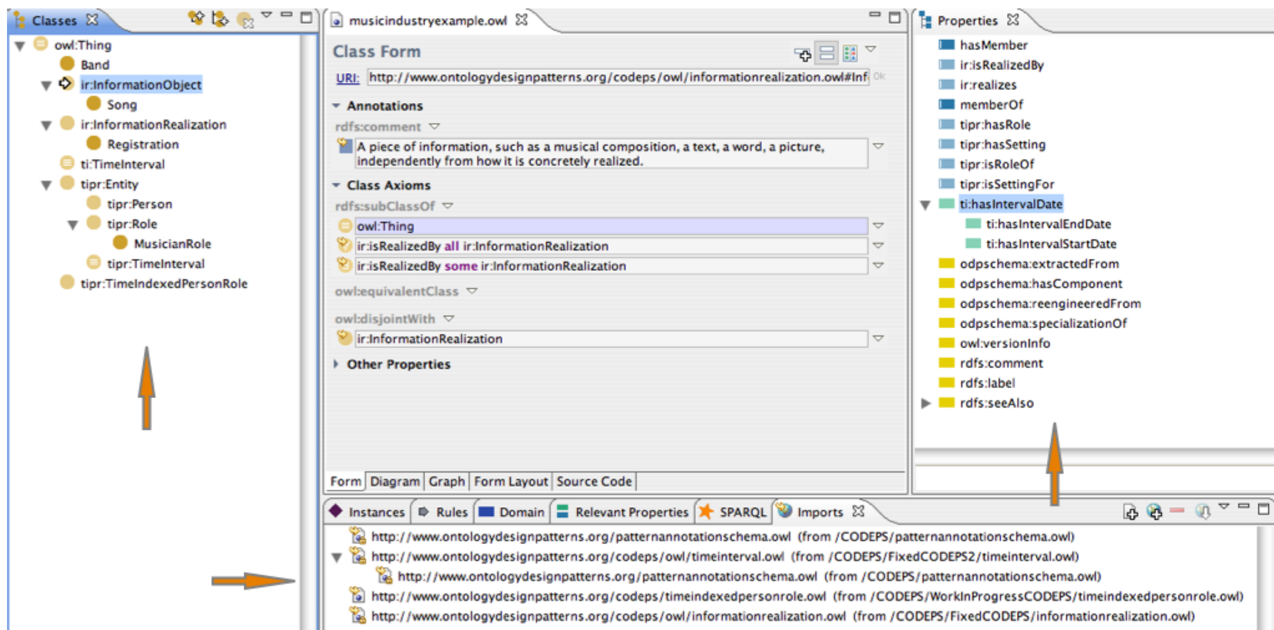


Figure 3.14: The music industry example.

- *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 registration, 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 **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. The second requirement also requires to represent a membership relation between a person and a band[18]. Let's consider a case in which we cannot find more specialized CPs for reusing. We proceed by following the above guidelines. Figure 3.14 shows a screenshot of the resulting ontology fragment. In the bottom part of the screenshot we find the import tab where the **information realization**[19] and **time indexed person role**[20] CPs are imported. Additionally, we import the **time interval** CP that allows us to assign a date to the time interval[21] 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 domain i.e., `tipr:Person`, and range i.e., `Band`.

3.6.2 Pattern-based ontology evaluation

CPs can also be used as tools for evaluating ontologies. We have started a pattern-based evaluation of preliminary versions of FAO ontologies that have been developed in the context of WP7, and we used CPs as well as other types of OPs for achieving our goal. Although the definition of a NeOn method for pattern-based ontology evaluation is still ongoing work (it will be one of the arguments of next NeOn

[18] The **collection entity** CP addresses membership.
[19] We use the prefix `ir` for this CP.
[20] We use the prefix `tipr` for this CP.
[21] The **time interval** CP also defines two additional sub-properties of the `hasIntervalDate` for expressing a start and an end date to the time interval.

2006–2008 © Copyright lies with the respective authors and their institutions.

deliverable D2.2.2a), we think such an experience deserves to be reported here.

We report the results of the evaluation analysis that has been performed on the first set of ontologies created for the FSDAS. Although some of the issues that emerged during this evaluation have already been addressed in the latest version of the ontologies (to the date of this deliverable), their description is relevant to assess typical antipatterns that happen to be followed during the development of ontologies from KOSes. According to the criteria mentioned in chapter 2, we provide evaluation about the modeling style that should be sound according to the known best practices of ontology design. Therefore, we provide a critique by performing pattern-based analysis, according to the theoretical work that has been presented in this deliverable as well as in deliverables D5.1.1 [SFBG+07], and D2.2.1 [SAd+07].

For the sake of clarity, we group the issues by highlighting the OPs that address them.

Namespace Naming: no convention has been used for giving a namespace to the different FAO ontologies. For example, one has `http://www.owl-ontologies.com/Ontology1160137180.owl`, another has `http://www.fao.org/onto.owl`, and another, which however uses a good practice, has `http://www.fao.org/commodities-ISSCFC.owl`. It is recommended to establish a unique convention for naming ontology namespaces, and stay with it, so that ontology creators do not have the responsibility of this task and are not obliged to know how to be aligned with other ontologies from the same organization. The third approach mentioned allows users to recognize the provenance of the ontology. A good practice is to compose the namespace by concatenating the organization base URI with a meaningful name that describes the topic of the ontology. Another good practice is to define a convention for encoding versioning information in the name: `http://www.fao.org/commodities-ISSCFC20070712.owl`, or in the reference directory, e.g. `http://www.fao.org/2007/07/12/commodities-ISSCFC.owl`. The latest practice is recommended by W3C[22].

Class and Property Names: some class and property names contain plurals e.g. `Areas`, `hasParts`. This is to be considered a bad practice unless it is explicitly required by the context. For example, a class the instances of which are single areas should be named `Area` and should not be named `Areas`. Moreover, some class and property names are unreadable because they are alphanumeric codes, e.g. in the Commodities-ISSCFC ontology. A good practice for naming conventions of classes and properties is to give them a readable name either directly to the class, or to its label. This allows tools to visualize hierarchies and diagrams with readable names. The FAO designers have applied a lexicalization pattern (commented in next bullet) in order to associate readable names to ontology elements. Unfortunately, current tools are not able to exploit personalized patterns for visualizing readable names.

Lexicalization: The requirement to be addressed is the following: each concept encoded by the class of the ontology has to be associated to one name for each supported language. This requirement is identified as multilingual lexicalization of ontology classes.
This issue might be addressed by means of the `rdfs:label` annotation property. Each class can be annotated by a number of labels corresponding to the number of supported languages[23]. However, this solution keeps the names out of the theory and knowledge base, with the consequence of preventing reasoning on names. To move around this issue, we need to include a class for names, and to associate each class to its corresponding name instance. The name instance is then associated with the different lexicalizations by means of appropriate datatype properties[24].
FAO designers adopted an alternative solution in order to encode multilingual lexicalization of ontology

[22] http://www.w3.org

[23] The value of the `label` includes the standard code e.g., `{en} for English`, of the specific language by which the name is expressed

[24] E.g., one for each supported language.

classes. It consists of defining a class for names i.e., `Name` and creating a dummy instance for each class, which is then related to an instance of the class `Name` (the one which names the class) through the `hasName` relation. For example:

```
Crustacean rdfs:subClassOf Commodity
Crustacean_concept rdf:type Crustacean
Crustacean_lexitem rdf:type Name
Crustacean_concept hasName Crustacean_lexitem
```

This solution creates an explicit path from classes to lexicalizations through the dummy instance `Crustacean_concept`. However, there is an issue with the dummy instance: it is ontologically wrong. Each instance of the class `Crustacean` should be (in this domain) a crustacean commodity, not a dummy object. For this reason, this solution is provides us with an example of AntiCP.

A possible good solution is provided by composing two CPs: **intension extension**[25] and **lexicalization**[26].

The **intension extension** CP represents the relation between a concept and the information object that expresses it, through the `isExpressedBy` relation. The **lexicalization** CP solves the dummy instance problem by providing the class `Concept`. The instances of this class are used instead of dummy instances. Concepts are the reified counterparts of classes, which represent their intensional semantics (where the formal semantics of classes is only extensional). Classes are related to concepts by means the relation `classifiedBy`, to which a `owl:hasValue` restriction is defined. This means that each class is associated to a specific instance of `Concept`. The class `LexicalItem` represents names. A name is associated with a concept through the `expressedBy` object property. For example:

```
Crustacean rdfs:subClassOf Commodity
Crustacean_concept rdf:type Concept
Crustacean_lexitem rdf:type LexicalItem
Crustacean rdfs:subClassOf (classifiedBy hasValue Crustacean_concept)
Crustacean_concept expressedBy Crustacean_lexitem
```

Alternative solutions for ontology lexicalization are described in the NeOn deliverable D2.4.1 [MPPdC$^+$07].

Part of: in the `FAOareas.owl` ontology we identified a typical AntiCP. Even if this ontology is not included in the prototype set of ontologies, we believe it is worth to mention this issue because it is one of the lessons learnt by FAO experts in the context this work. In the above mentioned ontology a relation `partOf` is defined in order to model partonomies. The relation is defined as functional and not transitive. There exist plenty of design patterns on `partOf` relations, either in general or for specific domains, including geographic partonomies e.g., see section 4.2.1. Most patterns agree on having the `partOf` (or `isPartOf`) relation as transitive (unless a system-oriented view is taken), and not functional, because something can be part of more than one individual. The modeling choice in this ontology (being `partOf` functional and non-transitive) is against the good practice.

Subsumption hierarchy of areas: from the past FOS project we know that there is a distinction between areas, subareas, divisions, and subdivisions. They are modeled as ontology classes. What is more, such distinction creates a partition, i.e. no instance of one distinct class can be instance of another. In `FAOareas.owl` the four classes are modeled by means of a subsumption pattern, which creates the *bad* consequence of having all finer partitions to be instances of coarser partitions i.e., it is an AntiCP. The good practice in this case is to create disjoint classes, and relate them by means of restrictions on the `partOf` relation (by following the guidelines above described). For example:

[25] Described in 4.3.1

[26] Exemplification of the **classes as values** LP described in section 2.2.1 under the *Logical Ontology Design Pattern* paragraph.

2006–2008 © Copyright lies with the respective authors and their institutions.

```
Area rdfs:subClassOf MarineArea
SubArea rdfs:subClassOf MarineArea
SubArea rdfs:subClassOf (partOf someValuesFrom Area)
```

The previous pattern is known as **granular partition**: by effect of the transitivity of `partOf`, all finer granularity marine areas will result to be necessarily `partOf` at least one (or exactly one) marine area from each of the granularity levels above it.

Political objects versus physical objects: a class `Territory` has been defined in order to model physical territories, but it also encodes political properties of these territories e.g. `isSuccessorOf`, which is used in order to state that a new political unit has been created at some time, for the same territory. A good pattern here is to have political geographic objects as distinct from physical territories, in order to keep physical and political properties distinct.

Collections: the class `Groups` represents collections having territories as members. The FAO designers put such class under the class `Areas`[27] by means of a subsumption pattern. Nevertheless, the class `Groups` should not be put there because collection are usually understood as different from (geographical) areas. The motivation is that a collection of areas has not the same properties as a single area in usual conceptualization patterns. For example, it is common to say that “Italy has a beautiful climate”, and that “the NATO accepted a new member”, while it is counterintuitive to say that “NATO has a beautiful climate” or that “Italy accepted a new member”.

Layered taxonomies: the `species.owl` ontology encodes a natural taxonomy. Currently, FAO designers approach such issue by representing the different layers as a subsumption hierarchy. This is an anti-pattern, because causes e.g. species to be also genus, families, etc. Natural taxonomies that distinguish between e.g. species, genus, family, etc. constitute a so-called **linnean taxonomy** (see section 4.10.1. A linnean taxonomy can be represented at the meta-level, or at the object level, depending on the logical level of representation chosen for species, genera, families, etc. FAO designers want to represent elements of the linnean taxonomy at the object level. In this case, the best practice is to model the layers as disjoint classes, and to relate them by means of a relation which indicates the next layer. For example, by assuming the relations from the FAO ontology as kinds of such a relation, we have the following:

```
Species rdfs:subClassOf (belongsToGenus someValuesFrom Genus)
i_Salmon_shark rdf:type Species
i_Mako_sharks rdf:type Genus
i_Salmon_shark belongsToGenus i_Mako_sharks
```

However, when we use this pattern with differentiated relations, we need to assert specifically all higher layers for lower layers. For example, if:

```
i_Mako_sharks isOfFamily i_Mackerel_sharks-porbeagles_nei
```

we would like to automatically infer that:

[27]The class `Areas` represents single areas, and according to the class and property names pattern it should be named `Area` (see above).

```
i_Salmon_shark hasHigherRank i_Mackerel_sharks-porbeagles_nei
```

This automatic inference can be obtained by declaring relations like `belongsToGenus` and `isOfFamily` as `subPropertyOf` `hasHigherRank`, and declaring `hasHigherRank` as transitive:

```
hasHigherRank rdf:type TransitiveProperty
belongsToGenus rdfs:subPropertyOf hasHigherRank
isOfFamily rdfs:subPropertyOf hasHigherRank
```

2006–2008 © Copyright lies with the respective authors and their institutions.

Chapter 4

Catalogue of Content Ontology Design Patterns

Ontology design patterns (OPs) have a strong analogy to software engineering (SE) patterns. The mainstream approach for describing SE patterns is to use a template and to collect them by means of a catalogue. A description of the most well-known SE pattern templates can be found at Martin Fowler's web site.[1]
The templates used for describing SE patterns follow quite closely that suggested by Alexander [Ale79]: 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, but the template used for the presentation has been optimized for the web and defined in an OWL annotation schema; it is the same used on the semantic web portal http://www.ontologydesignpatterns.org. This chapter contains a catalogue of CPs that are classified according to the domain they share i.e, the domain of the modeling issues they solve. Each CP is described by means of a *catalogue entry*, which is composed of the following information fields, defined in the annotation schema:

- **Name:** indicates a name for the CP.
- **Intent:** describes the goal of the CP.
- **Also Known as:** indicates alternative names (if any) for the CP.
- **Extracted from:** contains the URI (if any) of the ontology where the CP has been extracted from. Alternatively, if the CP is the result of a reengineering process, the **Reengineered from:** field is used, which indicates the source model. This field could be omitted e.g., if the CP is a specialization or composition of other CPs.
- **Requirements:** contains a list of competency questions expressed in natural language, that are covered by the CP.
- **Diagram:** depicts a UML class diagram that graphically represents the CP.
- **Example:** provides the reader with a possible scenario. This field sometimes includes a UML diagram that graphically represents individuals and relationships complying to the scenario. This field can also contain more specific requirements that can be solved by specializing the CP.
- **Elements:** describes the elements (classes and properties) included in the CP, and their role within the CP.
- **Consequences:** contains a description of the benefits and/or possible trade-offs when using the CP.

[1] http://www.martinfowler.com/articles/writingPatterns.html#CommonPatternForms

- **Known uses:** gives examples of real ontologies where the CP is used. In this presentation of the catalogue, this field is omitted, because CPs as building blocks are presented here for the first time. They are extracted from real ontologies, and on the base of experiences of reuse of those ontologies. However, the content of this field should refer to ontologies that reuse the CP as a building block.

- **Related patterns:** indicates other CPs (if any) that *specialize*, *generalize*, *include*, or *are components of* the CP. Furthermore, this field may indicate other CPs that are typically used in conjunction with the described one. Important similarities and differences with other patterns can be also described here.

- **Building block:** contains the URI of the OWL implementation of the CP, i.e. the reusable component available for download.

- **References:** optional, may contain references to resources (e.g. papers, theories, blogs) that are related to the knowledge encoded in the CP. In the catalog included in this deliverable, this field is not used, but a lot of references can be found in the web sets of the reference ontologies where the CP are extracted from.

The web addresses that are used in this chapter are abbreviated with a prefix followed by the name of the OWL file according to the following:

- `odp:` is for `http://www.ontologydesignpatterns.org/`[2]

- `loa:` is for `http://www.loa-cnr.it/ontologies/`

The CP catalogue is an evolving resource. The main aim of this chapter is to present the way the catalogue is organized, and how the CPs are described. Many patterns are extracted from the DOLCE UltraLite reference ontology, because it has proved along many years to contain useful solutions that can be easily specialized. Other patterns show different approaches for reengineering, specializing, and composing. The CPs presented here will be certified by a quality committee of experts and published on ODP[3]. It is not our aim to build a complete catalogue by ourselves: a catalogue of CPs is inherently a collaboratively built resource, since it is composed of emerging good practices.

4.1 General

4.1.1 Types of entities

Name: types of entities.

Intent: to identify and categorize the most general types of things in the domain of discourse.

Extracted from : loa:DUL.owl

Requirements: what kind of entity is that? Is this an event or an object? Is this an abstract value or a quality of an entity?

Diagram: Figure 4.1 shows the UML diagram of this CP.

[2]Temporarily CPs owl files are stored at the loa: address. Therefore, the reader should read loa: in place of odp: for the building blocks' URL.

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

2006–2008 © Copyright lies with the respective authors and their institutions.

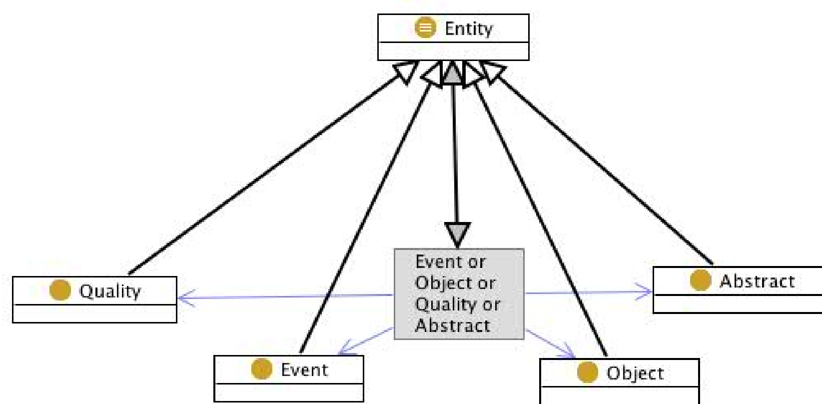


Figure 4.1: The types of entities CP

Example: that copy of *Divina Commedia* is a book (Object), the porosity (Quality) of the paper used is 5 mls/min (Abstract). The Rock Music Festival (Event) is organized by a friend of mine (Object). Figure 4.2 shows the UML diagram of this scenario.

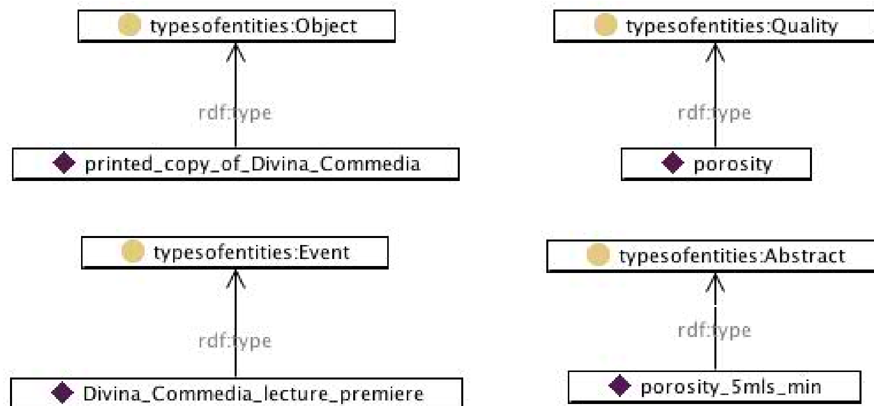


Figure 4.2: The types of entities example scenario.

Elements: the **types of entities** CP consists of the following elements.

- `Abstract`, any entity that cannot be located in space-time. E.g. mathematical entities, formal semantics elements, regions within dimensional spaces, etc.
- `Event`, any physical, social, or mental process, event, or state. More theoretically, events can be classified in different ways, possibly based on aspect (e.g. stative, continuous, achievements, etc.), or on agentivity, typical participants (e.g. human, physical, abstract, food, etc.).
- `Object`, any physical, social, or mental object, or a substance.
- `Quality`, any aspect of an entity (but not a part of it), which cannot exist without that entity. For example, the way the surface of a specific physical object looks like is a `Quality`.

The four elements are defined as disjoint with each other. The four classes are an exhaustive partition of the class Entity.

Consequences: the type of any element of the knowledge base is always known.

Building block: `odp:typesofentities.owl`

4.1.2 Description

Name: description.

Intent: to formally represent a conceptualization or a descriptive context.

Extracted from: loa:DUL.owl

Requirements: which are the assumptions, under which a certain thing is described? which are the concepts involved in the description of a certain thing? what is the interpretation of this case/event/observation?

Diagram: Figure 4.3 shows the UML diagram of this CP.

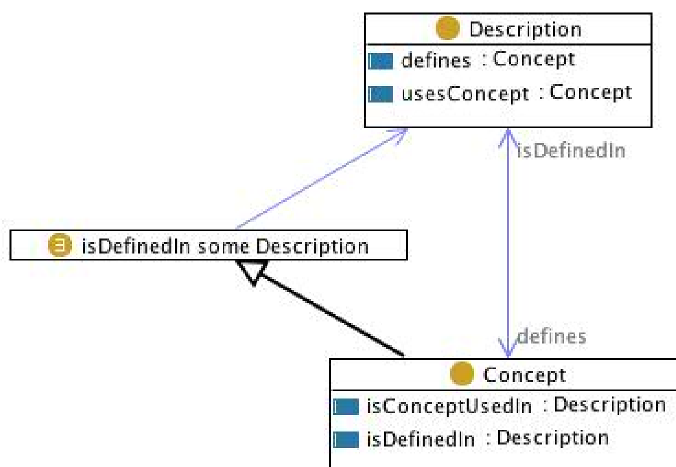


Figure 4.3: The description CP

Example: when I prepare a coffee I use a heater, a certain amount of water, and a coffee mix. Figure 4.4 shows the UML diagram of this scenario.

Elements: the **description** CP consists of the following elements.

- `Description`, a description represents a conceptualization. It can be thought also as a descriptive context that defines concepts in order to see a relational context out of a set of data or observations. For example, a Plan is a description of some actions to be executed by agents in a certain way, with certain parameters; a diagnosis is a description that provides an interpretation to a set of observed entities, etc.

2006–2008 © Copyright lies with the respective authors and their institutions.

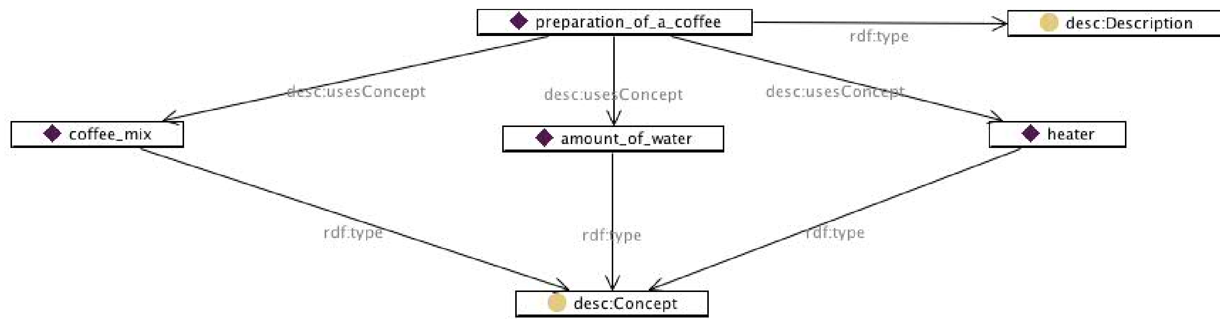


Figure 4.4: The description example scenario: preparation of a coffee.

- `Concept`, a concept can be an idea, notion, role, or even a reified class, and is defined in a description. Once defined, a concept can be used in other descriptions.
- `isDefinedIn`, a relation between a description and a concept, e.g. a workflow for a governmental organization defines the role officer, or the Italian Traffic Law defines the role Vehicle. In order to be used, a concept must be previously defined in another description. `defines` is its inverse.
- `isConceptUsedIn`, a more generic relation holding between a description and a concept. `usesConcept` is its inverse.

Consequences: this CP allows the designer to represent both a (descriptive) context and the elements that characterize and are involved in that context.

Related patterns: with respect to other patterns, descriptions are abstractions of situations (cf. the **situation** pattern), and in some complex use cases (e.g. matching executions to plan models), the two patterns can be composed (see the **descriptionandsituation** pattern). The `classifies` relation (see **classification** pattern) relates concepts to entities at some time.

Building block: `odp:description.owl`

4.1.3 Situation

Name: situation.

Intent: to represents facts, circumstances, observed contexts.

Extracted from: loa:DUL.owl

Requirements: what entities are involved in that situation (setting, state of affairs, fact, scenario, ...)?

Diagram: Figure 4.5 shows the UML diagram of this CP.

Example: I prepared a coffee with my heater, 300 ml of water, and an Arabica coffee mix. Figure 4.6 shows the UML diagram of this scenario.

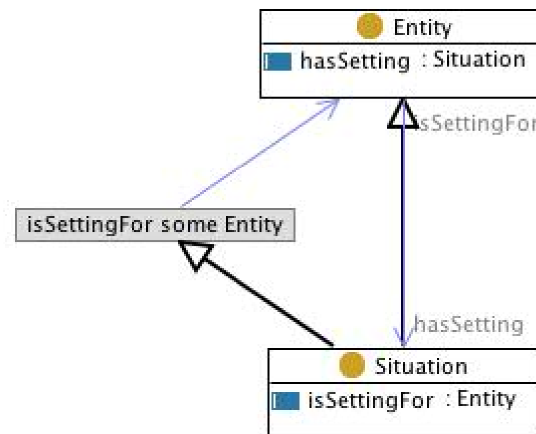


Figure 4.5: The situation CP

Figure 4.6: The situation example scenario: a particular preparation of a coffee.

Elements: the **situation** CP consists of the following elements.

- `Entity`
- `Situation`, a combination of circumstances involving a set of entities. It can be seen as a relational context, reifying a relation among the entities involved. In fact, it provides an explicit vocabulary to the **n-ary relation** Logical OP (see within section 2.2.1).
- `hasSetting`, a relation between entities and situations, e.g. this morning I've prepared my coffee with a new fantastic Arabica (i.e.: (an amount of) a new fantastic Arabica `hasSetting` the preparation of my coffee this morning). `isSettingFor` is its inverse.

Consequences: this CP allows the designer to model both a certain situation, and the entities that are involved.

Related patterns: a situation typically complies to a description, see **description** CP, section 4.1.2. If it deals with quantities and/or dimensions it can be composed with e.g., **parameter**, **region**, **price**, etc. It is specialized by e.g., **n-ary classification**, **time indexed part of**, **n-ary participation**, etc..

Building block: `odp:situation.owl`

2006–2008 © Copyright lies with the respective authors and their institutions.

4.1.4 Classification

Name: classification.

Intent: to represent the relations between concepts (roles, task, parameters) and entities (person, events, values), which concepts can be assigned to. To formalize the application (e.g. tagging) of informal knowledge organization systems such as lexica, thesauri, subject directories, folksonomies, etc., where concepts are first-order elements.

Extracted from: loa:DuI.owl

Requirements: what concept is assigned to this entity? Which category does this entity belong to?

Diagram: Figure 4.7 shows the UML diagram of this CP.

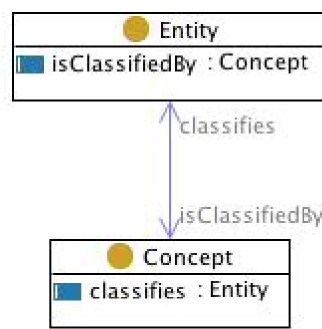


Figure 4.7: The classification CP

Example: Mac OSX 10.5 is classified as an an operating system in the Fujitsu-Siemens product catalog. Figure 4.8 shows the UML diagram of this scenario.

Figure 4.8: The classification example scenario: classification of an operating system.

Elements: the **classification** CP consists of the following elements.

- `Concept`, an idea, notion, role, constraint, a reified class, etc. In fact, it provides an explicit vocabulary to some uses of the **classes as values** Logical OP (see within section 2.2.1), e.g. for the case of 'this book is about beauty in the Middle Age'.
- `Entity`, anything: real, possible, or imaginary, which some modeler wants to talk about for some purpose.
- `classifies`, a relation between a `Concept` and an `Entity`, e.g. the role 'student' `classifies` a Person 'John'. `isClassifiedBy` is its inverse.

Consequences: it is possible to make assertions about e.g., categories, types, roles, which are typically considered at the meta-level of an ontology. Instances of Concept reify such elements, which are therefore put in the ordinary domain of an ontology. It is not possible to parametrize the classification over different dimensions e.g., time, space, etc.

Related patterns: typically combined with **descriptionandsituation**. It is specialized by **cat:objectrole**. The **n-ary classification** CP provides an alternative to this if classification has to be qualified over different dimensions.

Building block: `odp:classification.owl`.

4.1.5 N-ary Classification

Name: n-ary classification.

Intent: to represent the relations between concepts (roles, task, parameters), and entities (person, activities, values) which concepts are assigned to, as well as the time when, the place where, the agent who, or other dimensions that characterize the assignment. To formalize the application (e.g. tagging over time), of informal knowledge organization systems such as lexica, thesauri, subject directories, folksonomies, etc., where concepts are first-order elements.

Requirements: what concept is assigned to this entity at that time? Which category does this entity belong to at that time?

Diagram: Figure 4.9 shows the UML diagram of this CP.

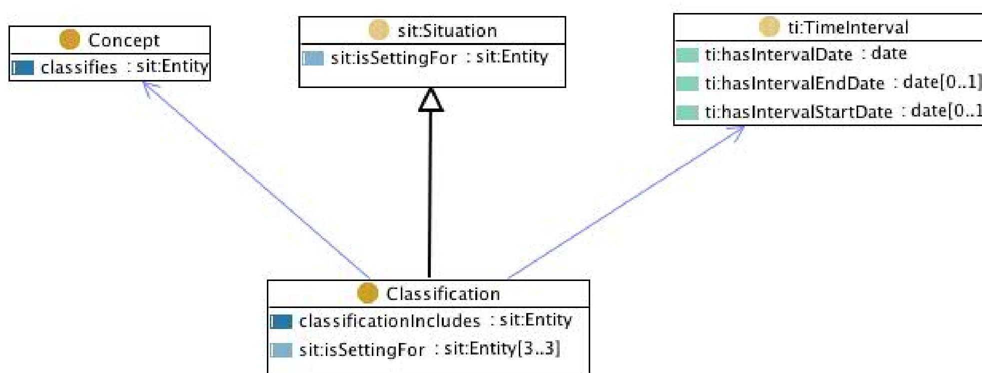


Figure 4.9: The n-ary classification CP

Example: the operating system I used in 2005 was Microsoft Windows XP, in 2006 I moved to Mac OS X 10.5. Figure 4.10 shows the UML diagram of this scenario.

2006–2008 © Copyright lies with the respective authors and their institutions.

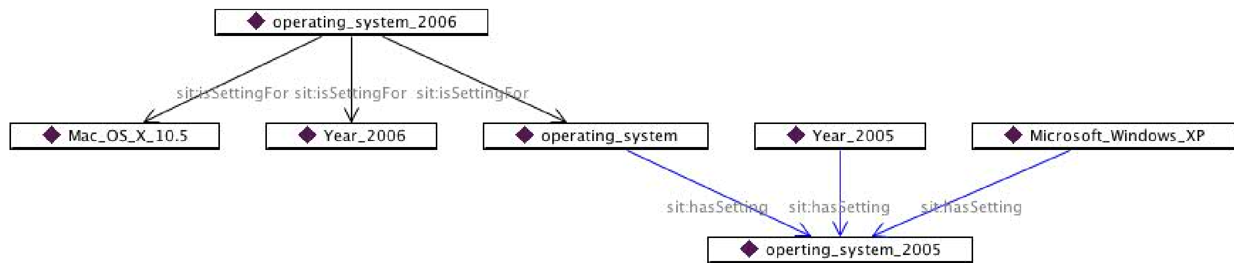


Figure 4.10: The n-ary classification example scenario: different operating systems.

Elements: the **n-ary classification** CP consists of the following elements.

- `Classification`, a special kind of Situation (see section 4.1.3) that allows to include time indexing for the classifies relation in situations. The following is a complex example that exploits the full expressivity of this pattern. If a situation s "my old cradle is used in these days as a flower pot" is a setting for the entity "my old cradle" and the TimeIntervals "8June2007" and "10June2007", and we know that s satisfies a functional description for aesthetic objects, which defines the concepts "flower pot" and "flower", then we also need to know what concept classifies "my old cradle" at what time. In order to solve this issue, we need to create a sub-situation s_1 for the classification time: "my old cradle is a flower pot in 8June2007". Such sub-situation s_1 isPartOf s.
- `Concept`, an idea, notion, role, constraint, a reified class, etc.
- `classificationIncludes`, a relation between entities and classifications. `isIncludedInClassification` is its inverse.

Consequences: it is possible to make assertions about e.g., categories, types, roles, which are typically considered at the meta-level of an ontology. Instances of Concept reify such elements, which are therefore put in the ordinary domain of an ontology. It is also possible to parametrize the classification over different dimensions e.g., time, space, agents, etc.

Related patterns: specializes the **situation** CP, and has the **time interval** CP as component. It is a more expressive alternative to the **classification** CP.

Building block: `odp:naryclassification.owl`

4.1.6 Description and Situation

Name: description and situation

Intent: to represents a conceptualization and its factual grounding i.e. a situation. To represent both a situational context and its interpretation.

Requirements: what is the description (conceptualization) of this situation (combination of circumstances)? What are the situations (combinations of circumstances) that satisfy this description (conceptualization)?

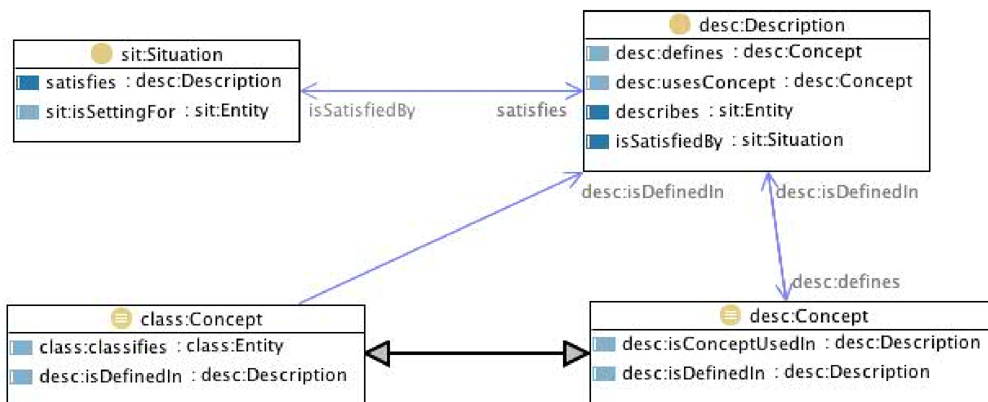


Figure 4.11: The description and situation CP

Diagram: Figure 4.11 shows the UML diagram of this CP.

Example: I make coffee by following the recipe of my mom.

Elements: the **descriptionandsituation** CP consists of the elements of its components (see 4.1.2 and 4.1.3). The following elements are defined locally.

- `describes`, the relation between a description and an entity: for example, a description gives a unity to a collection of parts (the components, or constituents), by assigning a role to each of them in the context of a whole object (the system). A same entity can be given different descriptions, for example, an old cradle can be given a unifying description based on the original aesthetic design, the functionality it was built for, or a new aesthetic functionality in which it can be used as a flower pot. `isDescribedBy` is its inverse.
- `satisfies`, a relation between a situation and a description, e.g. the execution of a plan satisfies that plan. `isSatisfiedBy` is its inverse.

Consequences: this CP allows designers to put in the same domain of discourse both a combination of circumstances and the conceptualization they can be described by.

Related patterns: is composed by other three CPs, **description** (see 4.1.2), **situation** (see 4.1.3), and **classification** (see 4.1.4).

Building block: `odp:descriptionandsituation.owl`

4.1.7 Object Role

Name: object role

Intent: to represent objects and the role they play.

2006–2008 © Copyright lies with the respective authors and their institutions.

Requirements: what role does this object play? which objects do play that role?

Diagram: Figure 4.12 shows the UML diagram of this CP.

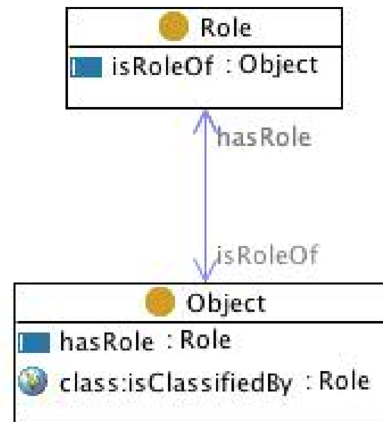


Figure 4.12: The object role CP

Example: this old glass is used as a flower pot. Figure 4.13 shows the UML diagram of this CP.

Figure 4.13: The object role example scenario: an old glass used as a flower pot.

Elements: the **object role** CP consists of the following elements.

- `Object`, any physical, social, or mental object, or a substance.
- `Role`, a concept that classifies an object.
- `hasRole`, a relation between an object and a role, e.g. the person 'John' has role 'student'.`IsRoleOf` is its inverse.

Consequences: it is possible to make assertions about roles, which are typically considered at the meta-level of an ontology. Instances of `Role` reify such elements, which are therefore put in the ordinary domain of an ontology. It is not possible to parametrize the classification over different dimensions e.g., time, space, etc.

Related patterns: it specializes the **classification** CP. It is specialized by **agent role**.

Building block: `odp:objectrole.owl`

4.2 Parts and collections

4.2.1 Part of

Name: part of.

Intent: to represents entities and their parts.

Also Known as: part-whole.

Extracted from: loa:DUL.owl

Requirements: what is this entity part of? What are the parts of this entity?

Diagram: Figure 4.14 shows the UML diagram of this CP.

Example: Brain and heart are parts of the human body, substantia nigra is part of brain. Figure 4.15 shows the UML diagram of this scenario.

Elements: the **part of** CP consists of the following elements.

- `Entity:` anything: real, possible, or imaginary, which some modeler wants to talk about for some purpose.
- `isPartOf:` a transitive relation expressing part-whole relations between any entities, e.g. brain is a part of the human body. When specializing this pattern, take care of restricting the domain and range appropriately, since it could be counterintuitive to use this relation arbitrarily, e.g. between animals and planets. `hasPart` is its inverse.

Consequences: this CP allows designers to represent entities and their parts i.e., part-whole relations, with transitivity. The temporal aspect of this relations cannot be expressed; in order to solve this issue the **time indexed part of** (4.2.2) CP can be used.

Related patterns: the **time indexed part of** is used as alternative when the temporal indexing has to be represented. Part of CP is specialized by **componency** (4.2.3). Sometimes this CP is wrongly used in order to model constituency and membership. These two modeling problems are addressed by **constituency** (4.2.4), and **collection entity** (4.2.5) CPs, respectively.

Building block: `odp:partof.owl`

4.2.2 Time Indexed Part of

Name: time indexed part of.

2006–2008 © Copyright lies with the respective authors and their institutions.

Figure 4.14: The part of CP

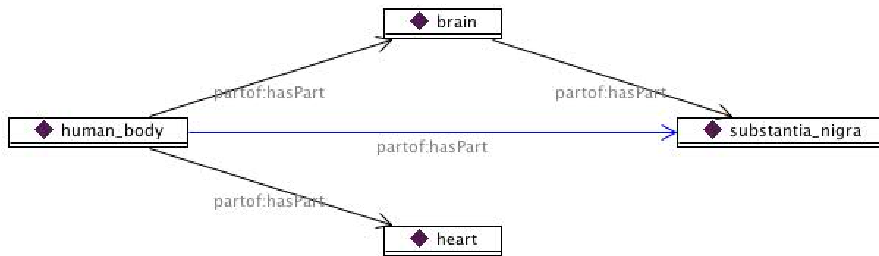


Figure 4.15: The part of example scenario: brain and heart are parts of the human body, substantia nigra is part of brain. Notice that by transitivity, also substantia nigra is inferred as part of human body (blue association denotes the inferred relation).

Intent: to represent objects that have temporary parts

Requirements: when was this object part of this other one? which object was this one part of at a certain time? what are the parts of this object at a certain time?

Diagram: Figure 4.16 shows the UML diagram of this CP.

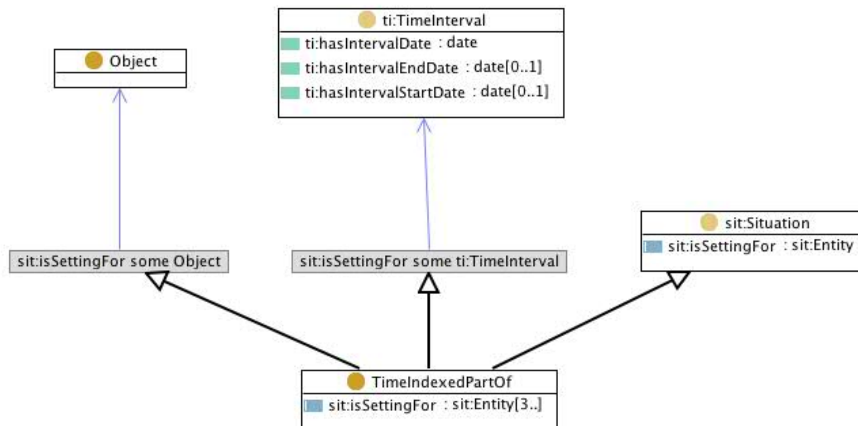


Figure 4.16: The time indexed part of CP

Example: my Toyota Yaris mounted Michelin pneumatics in 2007, but in 2008 it mounts Pirelli pneumatics. Figure 4.17 shows the UML diagram of this scenario.

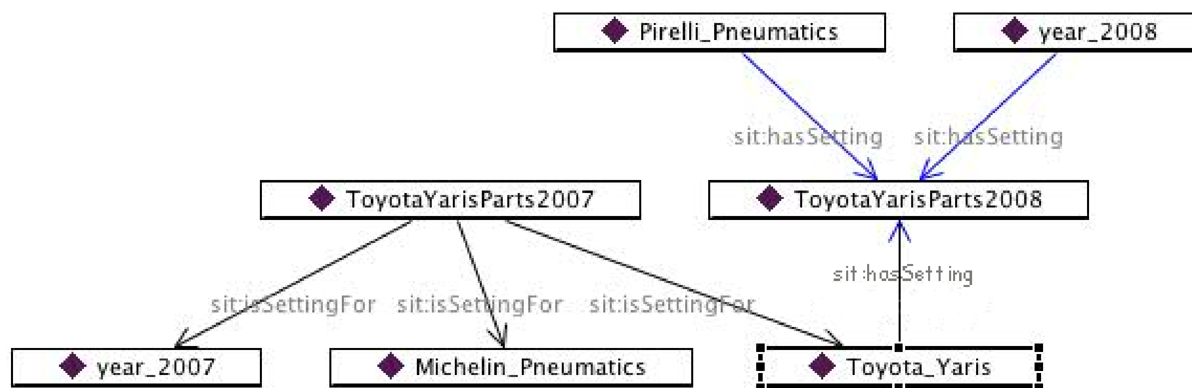


Figure 4.17: The time indexed part of example scenario: Toyota Yaris pneumatics changing over time.

Elements: the following elements are locally defined:

- `Object`: any physical, social, or mental object, or a substance
- `TimeIndexedPartOf`: a situation that includes at least two objects and one time interval, at which the part-whole relationship holds.

Consequences: this CP allows designers to represent part-whole relations with a temporal index (holding at a certain time).

Related patterns: this CP specializes **situation** (4.1.3) and is composed of **part of** (4.2.1), and **time interval** (4.8.1).

Building block: `odp:timeindexedpartof.owl`

4.2.3 Componency

Name: componency.

Intent: to represent (non-transitively) that objects either are proper parts of other objects, or have proper parts.

Also Known as: composition.

Requirements: what is this object component of? What are the components of this object?

Diagram: Figure 4.18 shows the UML diagram of this CP.

Example: a turbine is a proper part of an engine, both are parts of a car. Furthermore, the engine and a battery are proper parts of the car. Figure 4.19 shows the UML diagram of this scenario.

2006–2008 © Copyright lies with the respective authors and their institutions.

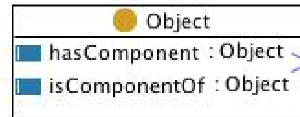


Figure 4.18: The componency CP

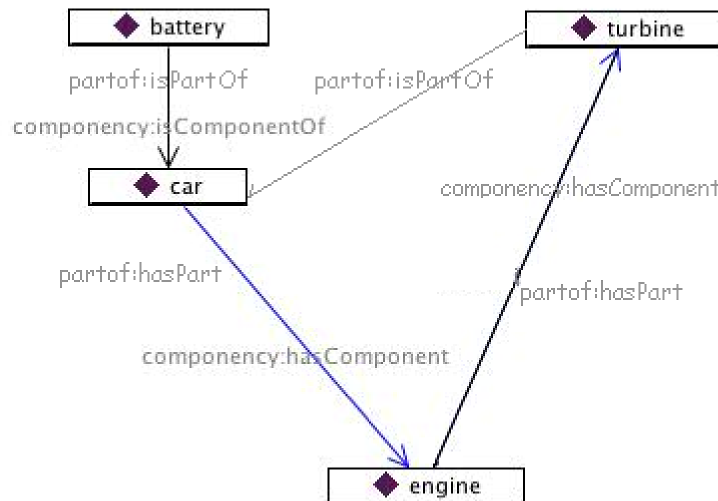


Figure 4.19: The componency example scenario

Elements: the **componency** CP consists of the elements of its components. The following elements are defined locally.

- `Object`, any physical, social, or mental object, or a substance.
- `hasComponent`, the has part relation without transitivity, holding between an object (the system) and another (the component), and assuming a design that structures the system object. For an explicit pattern encoding design, see the 'designobject' pattern. The componency pattern uses the transitive reduction logical pattern to preserve transitive on the superproperty from the partof pattern. In practice, the partof pattern acts here as the transitive reduction of the componency pattern. It is sub-property of texttthasPart of **part of** CP. `isComponentOf` is its inverse.

Consequences: this CP allows designers to represent part-whole relations. It allows to distinguish between parts and proper parts. Relation of proper part is not transitive, and implies a simple part of relation, which is transitive. Temporal indexing is not expressible.

Related patterns: specializes the **part of** (4.2.1) CP. It is an exemplification of the **transitive reduction** (2.2.1) Logical OP.

Building block: `odp:componency.owl`

4.2.4 Constituency

Name: constituency

Intent: to represent the constituents of a layered structure.

Extracted from: `loa:DUL.owl`

Requirements: which are the constituents of this entity? what does this entity is constituent of?

Diagram: Figure 4.20 shows the UML diagram of this CP.

Figure 4.20: The constituency CP

Example: Different types of wood constitute this table. Figure 4.21 shows the UML diagram of this scenario.

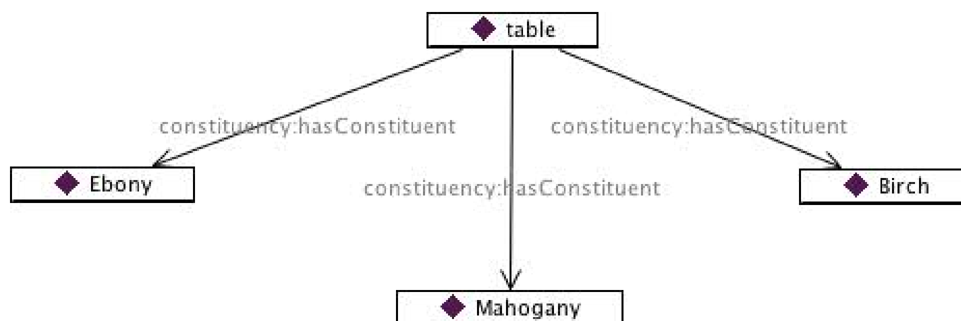


Figure 4.21: The constituency example scenario

Elements: the **constituency** CP consists of the following elements.

- `Entity`, anything real, possible, or imaginary, which some modeler wants to talk about for some purpose.
- `hasConstituent`, constituency depends on some layering of the world described by the ontology. For example, scientific granularity (e.g. body-organ-tissue-cell) or ontological 'strata' (e.g. social-mental-biological-physical) are typical layerings. Intuitively, a constituent is a part belonging to a lower layer. Since layering is actually a partition of the world described by the ontology, constituents are not properly classified as parts, although this kinship can be intuitive for common sense. Example of

constituents include the wood pieces constituting a table, the persons constituting a social system, the molecules constituting a person, the atoms constituting a river, etc. In all these examples, we notice a typical discontinuity between the constituted and the constituent object: e.g. a table is conceptualized at a functional layer, while wood pieces are conceptualized at a material layer, a social system is conceptualized at a different layer from the persons that constitute it, a person is conceptualized at a different layer from the molecules that constitute them, and a river is conceptualized at a different layer from the atoms that constitute it. `isConstituentOf` is its inverse.

Consequences: a desirable advantage of this CP is that we are able to talk e.g. of physical constituents of non-physical objects (e.g. systems), while this is typically impossible in terms of parts.

Related patterns: it has to be distinguished from **part of** (4.2.1), **collection entity** (4.2.5), and **componency** (4.2.3).

Building block: `odp:constituency.owl`

4.2.5 Collection Entity

Name: collection entity.

Intent: to represent collections and groups of entities by means of a membership relation.

Also Known as: membership.

Extracted from: `loa:DUL.owl`

Requirements: which groups does this object belong to? which are the members of this collection? who are the members of this group or collective?

Diagram: Figure 4.22 shows the UML diagram of this CP.

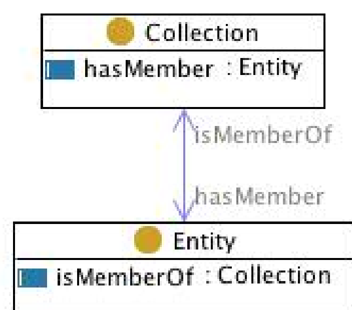


Figure 4.22: The collection entity CP

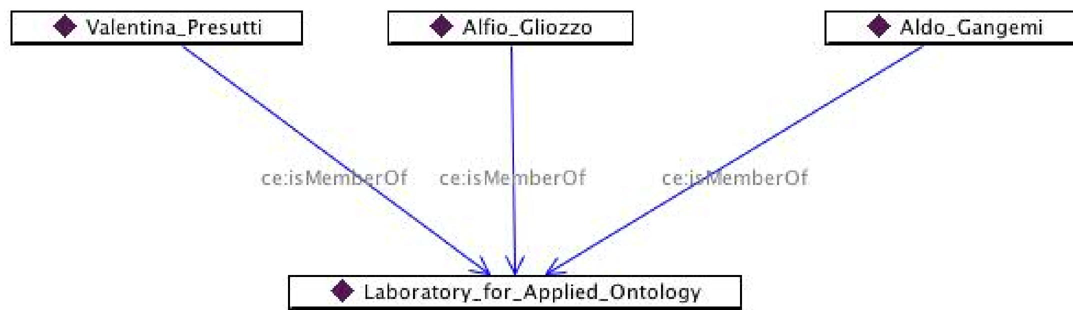


Figure 4.23: The collection entity example scenario: members of the Laboratory for Applied Ontology.

Example: Aldo Gangemi, Alfio Gliozzo and Valentina Presutti are members of the Laboratory for Applied Ontology. Figure 4.23 shows the UML diagram of this scenario.

Elements: the **collection entity** CP consists of the following elements.

- `Collection`, any container for entities that share one or more common properties. E.g. 'stone objects', 'the nurses', 'the Louvre Egyptian collection'. A collection is not a logical class: a collection is a first-order entity, while a class is a second-order one.
- `Entity`, anything: real, possible, or imaginary, which some modeler wants to talk about for some purpose.
- `hasMember`, a relation between collections and entities, e.g. 'my collection of saxophones includes an old Adolphe Sax original alto' (i.e. my collection has member an Adolphe Sax alto). `isMemberOf` is its inverse.

Consequences: it allows to represents groups, collectives, reified sets and classes, and their members. In fact, it provides an explicit vocabulary to some uses of the **classes as values** Logical OP (see within section 2.2.1), e.g. for the case of 'this book is about lions'.

Related patterns: it has to be distinguished from part of (4.2.1), **componency** (4.2.3), and **constituency** (4.2.4).

Building block: `odp:collectionentity.owl`

4.3 Semiotics

4.3.1 Intension Extension

Name: intension extension.

Intent: to represent the intensional expression (meaning) and extensional reference of information objects.

Extracted from: `loa:DUL.owl`

2006–2008 © Copyright lies with the respective authors and their institutions.

Requirements: what is this information about? what does this information object mean? which information object does express this meaning? which information object this entity is a reference of?

Diagram: Figure 4.24 shows the UML diagram of this CP.

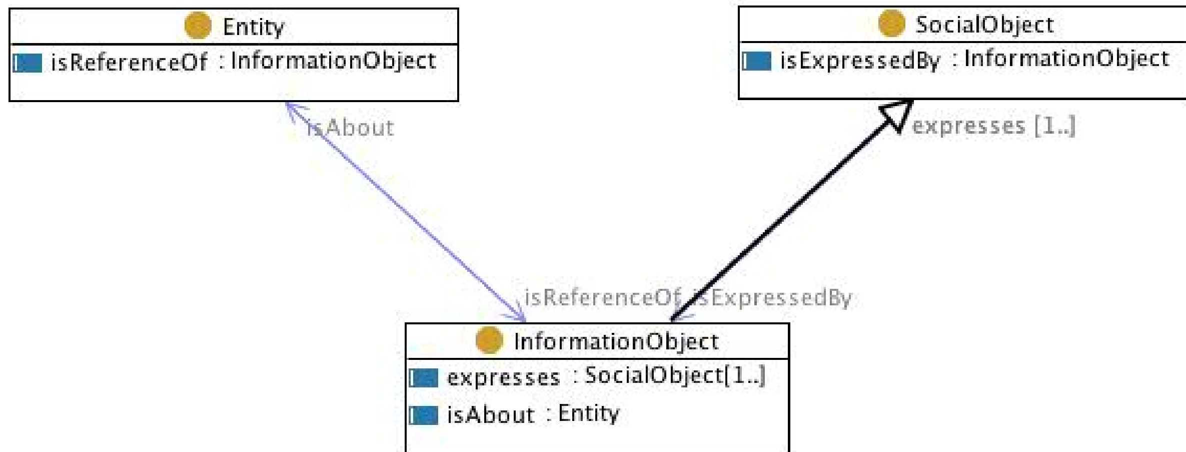


Figure 4.24: The intension extension CP.

Example: the term "sicilian tuna fisherman" expresses the concept of sicilian tuna fisherman and is about the sicilian tuna fishermen collective. Figure 4.25 shows the UML diagram of this scenario.

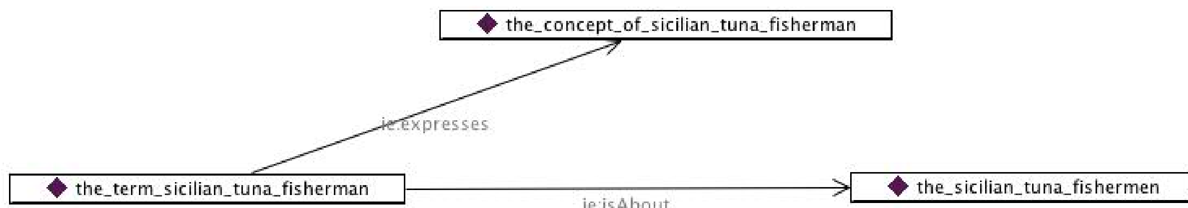


Figure 4.25: The intension extension example scenario.

Elements: This CP defines the following elements:

- `Entity`, anything: real, possible, or imaginary, which some modeler wants to talk about for some purpose.
- `SocialObject`, any Object that exists within some communication event, in which at least one physical object participates in. In other words, all objects that have been created in the process of social communication: for the sake of communication (information objects), or for incorporating new individuals (social agents, places), or for contextualizing existing entities (situations), or for collecting existing entities (collections), or for describing existing entities (descriptions, concepts).
- `InformationObject`, a piece of information, such as a musical composition, a text, a word, a picture, independently from how it is concretely realized.

- expresses, the relation between an information object and a 'meaning'. Meaning is here represented by the class SocialObject. The following clarifies this choice. What is a meaning is dependent on the background approach/theory that one assumes. For example, lexicographers that write dictionaries, glossaries, etc. assume that the meaning of a term is a paraphrase (or 'gloss', or 'definition'). Another approach is provided by concept schemes like thesauri and lexicons, which assume that the meaning of a term is a 'concept', possibly encoded as a 'lemma', 'synset', or 'descriptor'. Still another approach is that of psychologists and cognitive scientists, which often assume that the meaning of an information object is a concept encoded in the mind or cognitive system of an agent. The logical approach to meaning is completely different, since it assumes that the meaning of a term is equivalent to the set of individuals that the term can be applied to; for example, the meaning of 'Ali' is e.g. an individual person called Ali, the meaning of 'Airplane' is e.g. the set of airplanes, etc. Finally, an approach taken by structuralist linguistics and frame semantics is that a meaning is the relational context in which an information object can be applied; for example, a meaning of 'Airplane' is situated e.g. in the context ('frame') of passenger airline flights. All these different takes on what is meaning can be represented by some subclass of SocialObject. isExpressedBy is its inverse.
- isAbout, a relation between information objects and any entity (including information objects). It can be used to talk about entities are references of proper nouns: e.g. the proper noun 'Leonardo da Vinci' is about the person Leonardo da Vinci; as well as to talk about sets of entities that can be described by a common noun: e.g. the common noun 'person' is about the set of all persons in a domain of discourse. Sets of entities can be represented as collections. isReferenceOf is its inverse.

Consequences: this CP allows designers to model information objects, their meanings, the entities of the world they are about, and the relations between them.

Related patterns: can be composed with the **information realization** (section 4.3.2) CP.

Building block: odp:intensionextension.owl

4.3.2 Information Realization

Name: information realization

Intent: to represent information objects and physical entities that realize them.

Extracted from: loa:DUL.owl

Requirements: what physical entities do realize this information object? what information objects are realized by this physical entity?

Diagram: Figure 4.26 shows the UML diagram of this CP.

Example: John Lennon's biography *I Me Mine* has been realized in a limited 2000-copy edition. Figure 4.27 shows the UML diagram of this scenario.

2006–2008 © Copyright lies with the respective authors and their institutions.

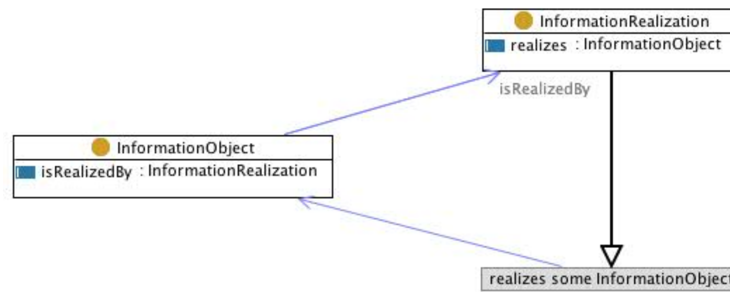


Figure 4.26: The information realization CP

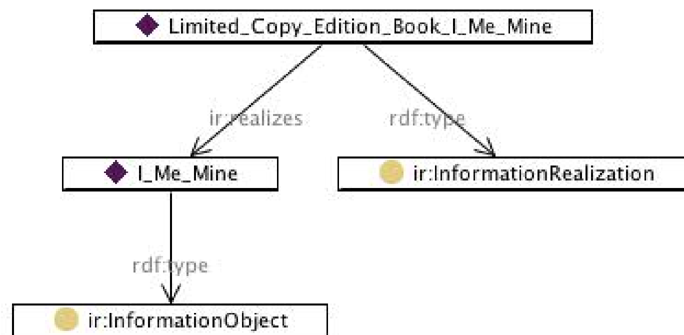


Figure 4.27: The information realization scenario: the *I Me Mine* John Lennon's biography.

Elements: the **information realization** CP consists of the following 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 information object, 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` is its inverse.

Consequences: this CPs allows designers to model information objects and their realizations. This allows the to reasons about physical objects and the information they realize by keeping them distinguished.

Related patterns: it is specialized by **multimedia data segment decomposition** (section 4.11.1).

Building block: `odp:informationrealization.owl`

4.4 Quantities and Dimensions

4.4.1 Region

Name: region.

Intent: to represent segments of a dimensional space (e.g. representing time, space, physical quantities, etc.) that can be used as values for a quality of an entity, or for talking about attribute values of an entity.

Extracted from: `loa:DUL.owl`

Requirements: what are the attribute values of this entity? which entities have this attribute value?

Diagram: Figure 4.28 shows the UML diagram of this CP.

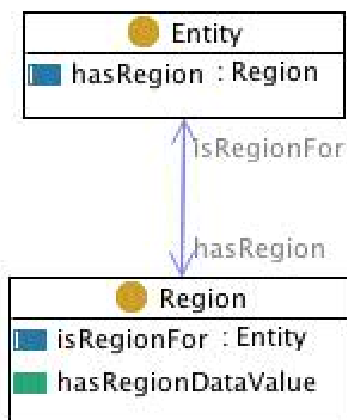


Figure 4.28: The region CP

Example: the number of wheels of my truck is 12. Figure 4.29 shows the UML diagram of this scenario.

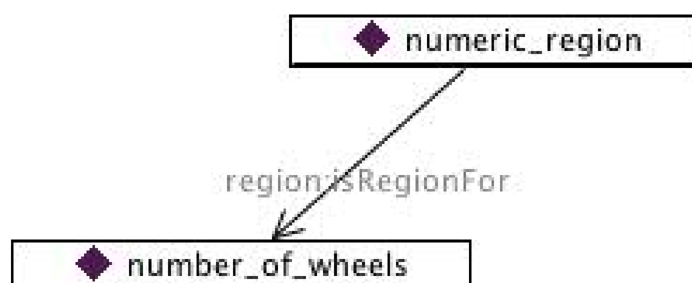


Figure 4.29: The region example scenario: number of wheels.

Elements: the **region** CP consists of the following elements.

- `Entity`, anything real, possible, or imaginary, which some modeler wants to talk about for some purpose.

2006–2008 © Copyright lies with the respective authors and their institutions.

- `Region`, any region in a dimensional space (a dimensional space is a maximal region), which can be used as a value for a quality of an entity, or directly as an attribute value of an entity. For example, time intervals, space regions, physical attributes, amounts, social attributes are all regions. Regions are not data values in the ordinary knowledge representation sense, because data values are computed as symbols, while regions are not computed as symbols, but as constants referring to non-computational entities.
- `hasRegion`, a relation between entities and regions. `isRegionFor` is its inverse.
- `hasRegionDataValue`, a datatype property that encodes values for a region, e.g. a float for the Region Height.

Consequences: this CP allows to represent and reason on dimensions.

Related patterns: typically used with **parameter** (section 4.4.3)and **basic plan** (section 4.6.7). It is a component of **parameter region** (section 4.4.4).

Building block: `odp:region.owl`

4.4.2 Region Overlap

Name: region overlap.

Intent: to represent overlapping regions.

Requirements: which regions (values) do overlap with this one? Does these two regions (values) overlap?

Diagram: Figure 4.30 shows the UML diagram of this CP.

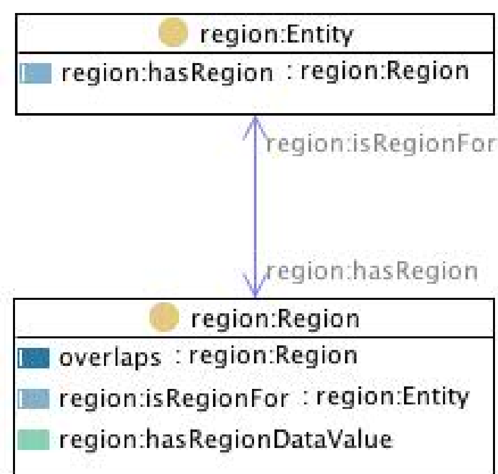


Figure 4.30: The region CP

Example: the time of my leave overlaps with the time of your arrival. Figure 4.31 shows the UML diagram of this scenario.

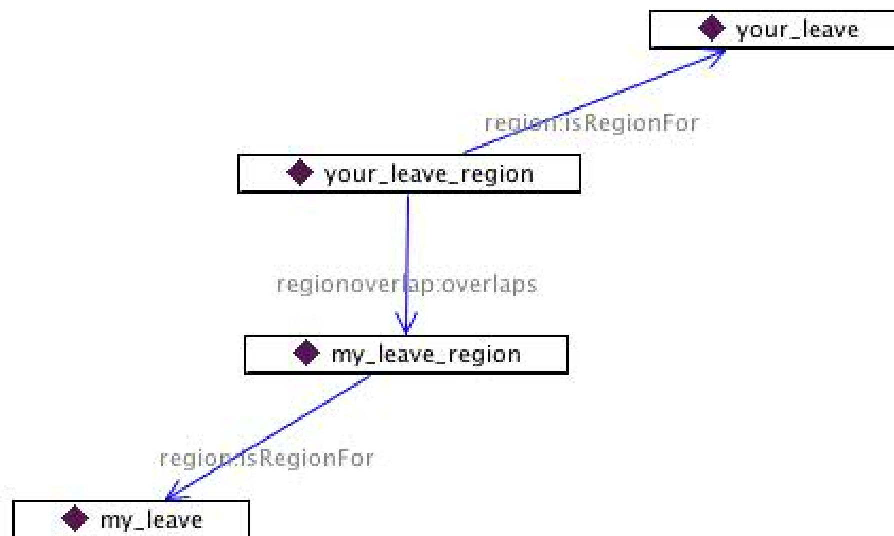


Figure 4.31: The region example scenario

Elements: the **region overlap** CP consists of the element of its component i.e., the **region** CP (section 4.4.1). The following elements are locally defined.

- `overlaps`, a schematic relation between any entities, e.g. 'the chest region overlaps with the abdomen region', 'my spoken words overlap with hers', 'the time of my leave overlaps with the time of your arrival', 'fibromyalgia overlaps with other conditions'. Subproperties and restrictions can be used to specialize overlaps for objects, events, time intervals, etc. It is a symmetric property.

Related patterns: it expands the **region** CP (section 4.4.1). It is typically used with **parameter** (section 4.4.3) and **basic plans** (section 4.6.7) CPs .

Building block: `odp:regionoverlap.owl`

4.4.3 Parameter

Name: parameter.

Intent: to represent parameters, which are constraints or selections on observable values.

Extracted from: `loa:DUL.owl`

Requirements: what is the parameter for this concept? What are the possible values of this parameter? What concept is this parameter for?

2006–2008 © Copyright lies with the respective authors and their institutions.

Diagram: Figure 4.32 shows the UML diagram of this CP.

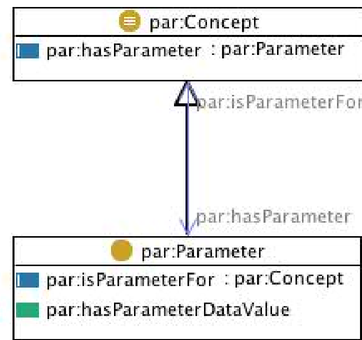


Figure 4.32: The parameter CP

Example: books can be either draft or complete. Figure 4.33 shows the UML diagram of this CP.

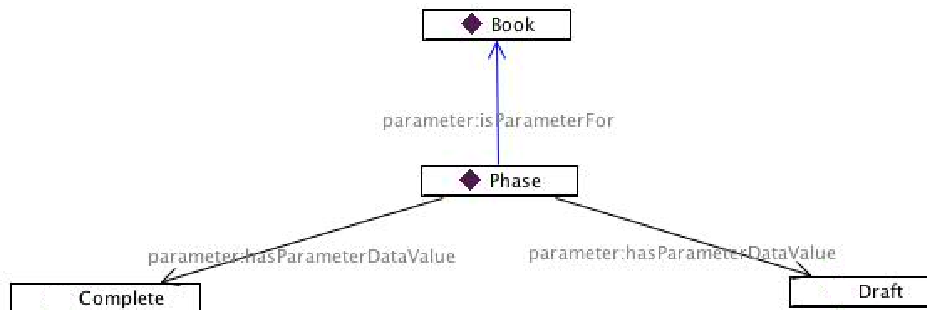


Figure 4.33: The parameter example scenario: phases of a book.

Elements: the **parameter** CP consists of the following elements.

- `Concept`, a concept is a social object, and is defined in a description .
- `Parameter`, a concept that classifies a region; the difference between a region and a parameter is that regions represent sets of observable values, e.g. the height of a given building, while parameters represent constraints or selections on observable values, e.g. 'VeryHigh'. Therefore, parameters can also be used to constrain regions, e.g. VeryHigh on a subset of values of the region Height applied to buildings, or to add an external selection criterion , such as measurement units. For example, a Meter parameter can be used on a subset of values from the Region Length.
- `isParameterFor`, a concept can have a parameter that constrains the attributes that a classified entity can have in a certain situation, e.g. a 4WheelDriver Role definedIn the ItalianTrafficLaw has a MinimumAge parameter on the Amount 16.. `hasParameter` is its inverse.
- `hasParameterDataValue`, parametrizes values from a datatype. For example, a Parameter `AgeForDriving hasParameterDataValue` 18 on datatype `xsd:int`, in the Italian traffic code. In this example, `AgeForDriving isDefinedIn` the Norm `ItalianTrafficCodeAgeDriving`.

Consequences: more complex parametrization requires workarounds. E.g. `AgeRangeForDrugUsage` could parametrize data value: 14 to 50 on the `datatype:xsd:int`. Since complex datatypes are not allowed in OWL1.0, a solution to this can only work by creating two 'sub-parameters': `MinimumAgeRangeForDrugUsage` (that hasParameterDataValue 14) and `MaximumAgeRangeForDrugUsage` (that `hasParameterDataValue` 50), which are components of the main parameter `AgeRangeForDrugUsage`. Ordering on subparameters can be created by using the **precedence** CP.

Related patterns: typically used with **region** (section 4.4.1) and **basic plan** (section 4.6.7).

Building block: `odp:parameter.owl`

4.4.4 Parameter Region

Name: parameter region.

Intent: to represent relations between parameters and regions.

Requirements: which parameter does parametrize this region? Which regions are parametrized by this parameter?

Diagram: Figure 4.34 shows the UML diagram of this CP.

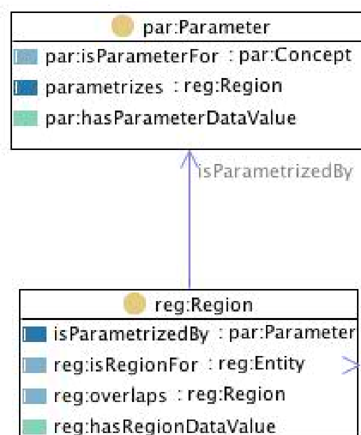


Figure 4.34: The parameter region CP

Example: the preparation of a book goes through phases such as "draft" and "complete". Figure 4.35 shows the UML diagram of this CP.

Elements: the **parameter region** CP consists of the elements of its components i.e., **region** (section 4.4.1) and **parameter** (section 4.4.3) CPs. The following elements are locally defined:

- `parametrizes`, the relation between a parameter, e.g. 'MajorAgeLimit', and a region, e.g. '18_year'. For a more data-oriented relation, see hasDataValue. `isParametrizedBy` is its inverse.

2006–2008 © Copyright lies with the respective authors and their institutions.

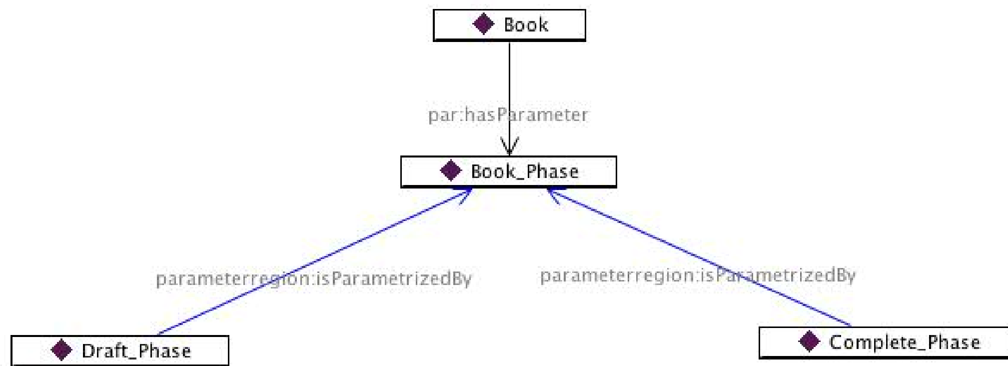


Figure 4.35: The parameter region example scenario

Consequences: this CP allows designers to model dimensions as individuals that can be put in the domain of an ontology.

Related patterns: is the composition of **parameter** (section 4.4.3) and **region** (section 4.4.1) CPs.

Building block: `odp:parameterregion.owl`

4.5 Participation

4.5.1 Participation

Name: participation.

Intent: to represent participation of an object in an event.

Extracted from: `loa:DUL.owl`

Requirements: which objects do participate in this event? Which events do this object participate in?

Diagram: Figure 4.36 shows the UML diagram of this CP.

Example: Aldo Gangemi participates in the premiere of the *La Dolce Vita*. Figure 4.37 shows the UML diagram of this CP.

Elements: the **participation** CP consists of the following elements.

- `Event`, any physical, social, or mental process, event, or state.
- `Object`, any physical, social, or mental object, or substance
- `hasParticipant`, a relation between events and objects. `isParticipantIn` is its inverse.

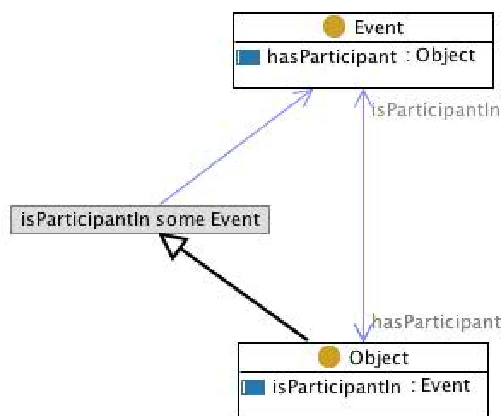


Figure 4.36: The participation CP.

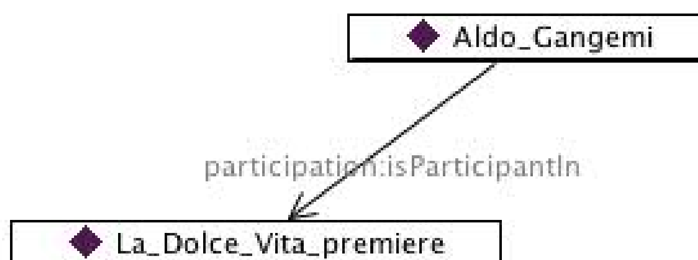


Figure 4.37: The participation example scenario

Consequences: it is possible to model whatever relation between objects and events. Using cardinality restrictions appropriately allows to limit the number of participants, e.g. 'life of' is a specialization of this pattern that requires a functional object property (cardinality 1. . .1).

Related patterns: **co-participation** (section 4.5.2) allows to model the relation between two objects that participate in a same event (implicitly); **n-ary participation** (section 4.5.3) allows to model participation over different dimensions e.g., temporal, spatial, etc., as well as explicitly relate more than one object to the same event; **object role** (section 4.1.7), **n-ary classification** (section 4.1.5) are examples of CPs that can be composed with this CP.

Building block: `odp:participation.owl`

4.5.2 Co-participation

Name: co-participation.

Intent: to represent the participation of two objects in a same event.

Requirements: what objects do participate in a same event?

Diagram: Figure 4.38 shows the UML diagram of this CP.

2006–2008 © Copyright lies with the respective authors and their institutions.

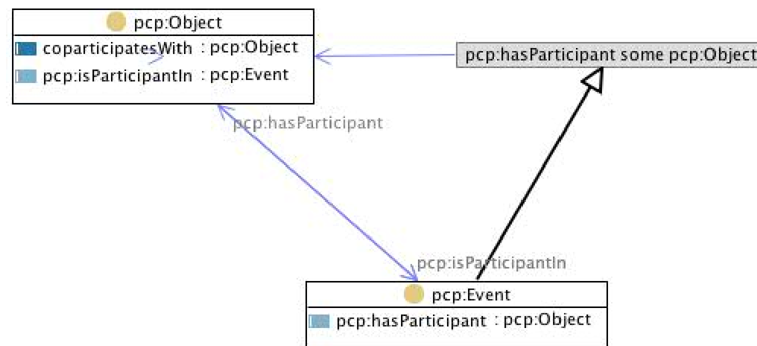


Figure 4.38: The co-participation CP

Example: Valentina Presutti and Aldo Gangemi participate in the ISWC 2007 conference. Figure 4.38 shows the UML diagram of this scenario.

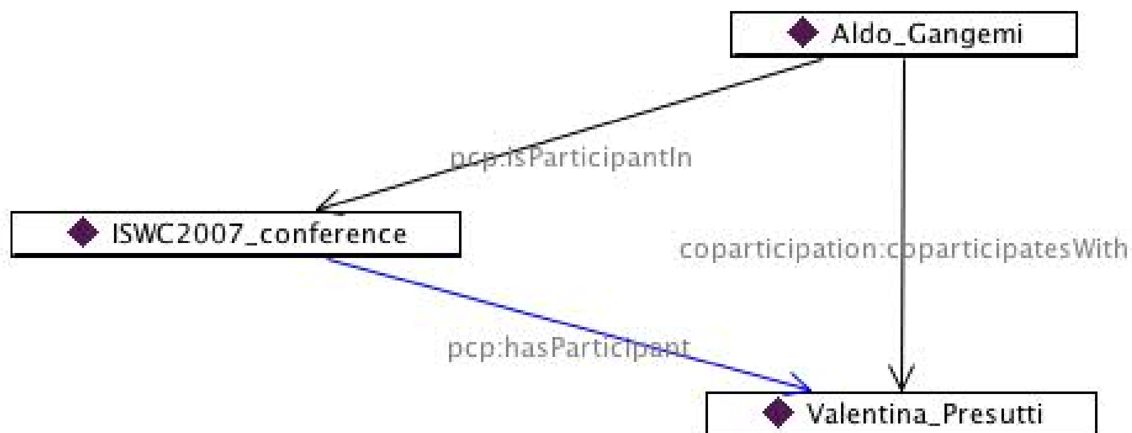


Figure 4.39: The co-participation example scenario: co-participation in the ISWC 2007 conference.

Elements: the **co-participation** CP consists of the elements of its component i.e., the **participation** CP (section 4.5.1). The following elements are defined locally.

- `coparticipatesWith`, a relations between two objects participating in a same event. This property is symmetric.

Consequences: it allows to assert that two objects (e.g. persons) participate in a same event. The coparticipatesWith property can also be inferred by means of a rule or a CONSTRUCT SparQL query, e.g.: $(coparticipatesWith\,?x\,?y) \leftarrow (hasParticipant\,?z\,?x) \wedge (hasParticipant\,?z\,?y)$. This CP does not allow either to represent participation over different dimensions e.g., time, space, etc., or to represent coparticipation of more than two objects in the same event (unless applied repeatedly). In order to address these issues a better choice is the **n-ary participation** (section 4.5.3) CP.

Related patterns: expands the **participation** CP (section 4.5.1).

Building block: `odp:coparticipation.owl`

4.5.3 N-ary Participation

Name: n-ary participation.

Intent: to represent a participation situation that involves objects, events, time, etc.

Requirements: which objects do participate in this event? When do the event in which these objects co-participate take place? Where did the events in which those objects participated take place?

Diagram: Figure 4.40 shows the UML diagram of this CP.

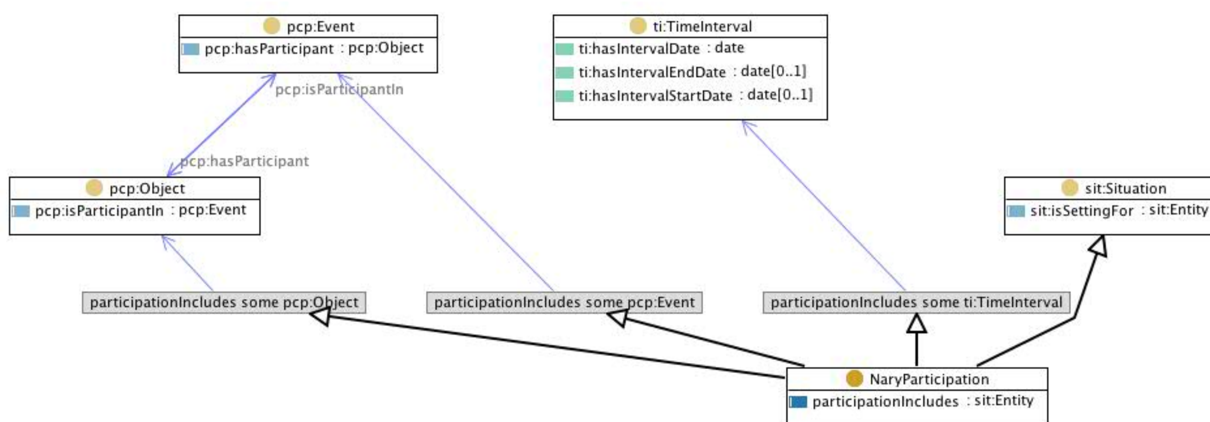


Figure 4.40: The n-ary participation CP

Example: Aldo Gangemi, Valentina Presutti and Alfio Gliozzo participated in the 2007 edition of ISCW with a paper. Figure 4.41 shows the UML diagram of this scenario.

Elements: the **n-ary participation** CP consists of the elements of its components i.e., **participation** CP (section 4.5.1) and **situation** CP (section 4.1.3). The following elements are defined locally.

- `NaryParticipation`, a participation situation with n elements involved.
- `isIncludedInParticipation`, a relation between objects, events, time intervals and situations of participation. `participationIncludes` is its inverse.

Consequences: it is possible to represent participation and co-participation of objects to events over different dimensions e.g., time, space, etc.

Related patterns: specializes and expands the composition of the **participation** (section 4.5.1) **time interval** (section 4.8.1) and **situation** (section 4.1.3) CPs. It is also an exemplification of the **n-ary relation** Logical CP (section 2.2.1).

Building block: `odp:naryparticipation.owl`

2006–2008 © Copyright lies with the respective authors and their institutions.

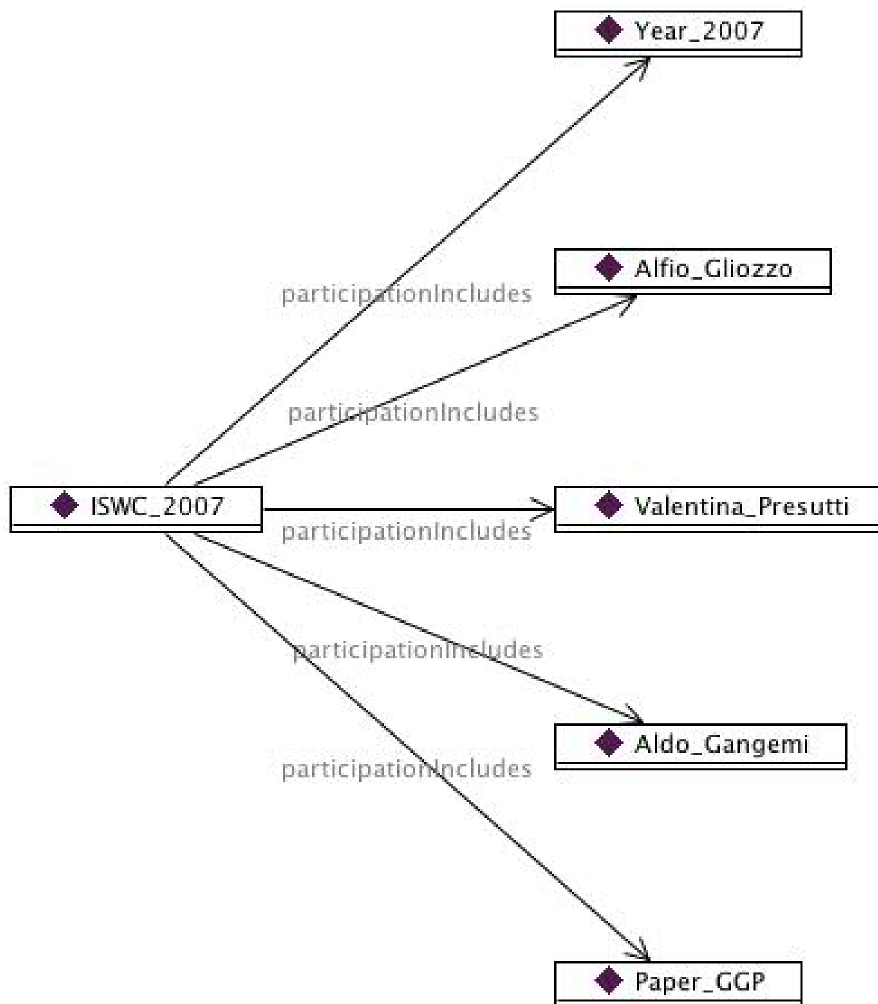


Figure 4.41: The n-ary participation example scenario: co-participation of objects to a certain event at a certain time, place, etc.

4.6 Organization, Management, and Scheduling

4.6.1 Precedence

Name: precedence.

Intent: to represent sequences of entities by means of ordering relations between each other.

Also Known as: sequence.

Extracted from: `loa:DUL.owl`

Requirements: which entities(y) do(es) precede this one? which entities(y) do(es) follow this one? Is this entity before this one? Is this entity after this one?

Diagram: Figure 4.42 shows the UML diagram of this CP.

Figure 4.42: The precedence CP.

Example: implementation follows design, and design follows analysis. Figure 4.43 shows the UML diagram of this CP.

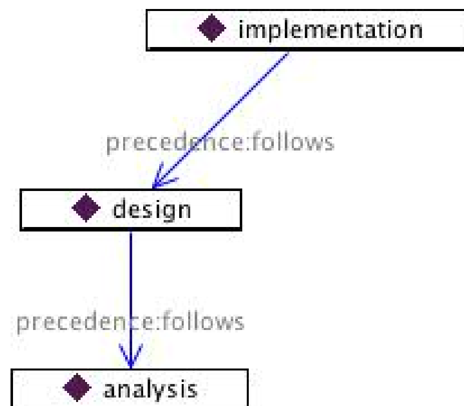


Figure 4.43: The precedence CP

Elements: the **precedence** CP consists of the following elements.

- `Entity`, anything real, possible, or imaginary, which some modeler wants to talk about for some purpose.
- `follows`, a relation between entities, expressing a 'sequence' schema. E.g. 'year 2000 follows 1999', 'preparing coffee' follows 'deciding what coffee to use', 'II World War follows I World War', 'in the Milan to Rome autoroute, Florence follows Bologna', etc. `precedes` is its inverse.

Consequences: This CP allows designers to model sequences involving any kind of entity. The order between two entities can be inferred thanks to the transitivity of the properties involved. It is not possible to express direct (intransitive) precedence, which can be modelled by specializing **precedence** and exemplifying the **transitive reduction** Logical OP (section 2.2.1, paragraph on **transitive reduction**).

Related patterns: it can be used between tasks, processes, time intervals, spatially locates objects, situations, etc. In general with *Organization, Management, and Scheduling* CPs (section 4.6).

Building block: `odp:precedence.owl`

4.6.2 Agent Role

Name: agent role.

2006–2008 © Copyright lies with the respective authors and their institutions.

Intent: to represent agents, the roles they play, and the relations between them.

Requirements: what agent does play this role? What is the role played by that agent?

Diagram: Figure 4.44 shows the UML diagram of this CP.

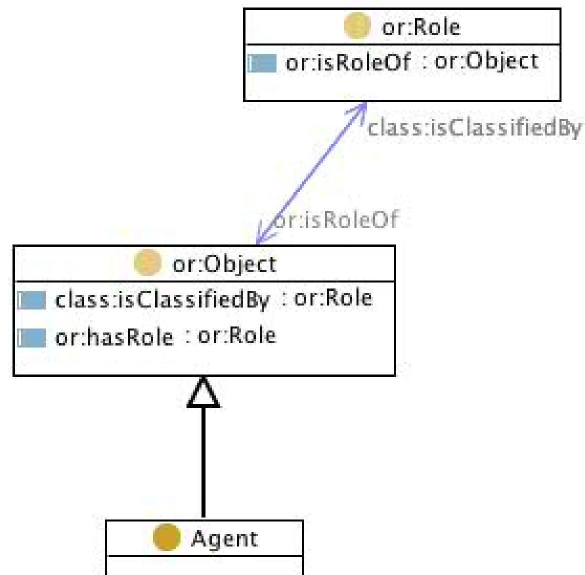


Figure 4.44: The agent role CP

Example: Aldo Gangemi plays the roles of father, saxophonist, and senior researcher. Figure 4.45 shows the UML diagram of this scenario.

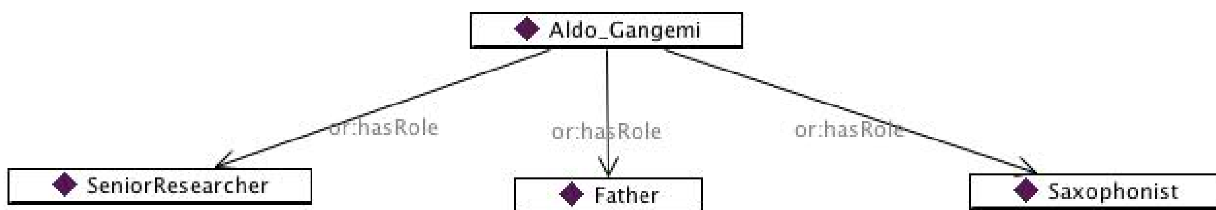


Figure 4.45: The agent role example scenario

Elements: the **agent role** CP consists of the elements of the CPs it reuses i.e., **object role** CP (section 4.1.7). The following elements are locally defined.

- `Agent`, any agentive object , either physical, or social. Disjoint with `Role`. It is the clone of `loa:DUL.owl#Agent`

Consequences: it is possible to make assertions on roles played by agents without involving the agents that play those roles, and vice versa. Temporal, spatial, or other dimensional aspects of roles are not expressible. In order to address these issues, the **n-ary classification** (section 4.1.5) can be specialized.

Related patterns: specializes the **object role** CP. The **time-indexed person role** CP (section 4.6.4) specializes this CP and allows to represent temporariness of roles played by persons. It can be generalized for including whatever objects or, alternatively the **n-ary classification** CP (section 4.1.5) can be specialized in order to obtain the same expressivity.

Building block: `odp:agentrole.owl`

4.6.3 Task Role

Name: task role.

Intent: to represent tasks, roles, and the assignment of tasks to roles.

Extracted from: `loa:DUL.owl`

Requirements: which roles is this task assigned to? which are the tasks of this role?

Diagram: Figure 4.46 shows the UML diagram of this CP.

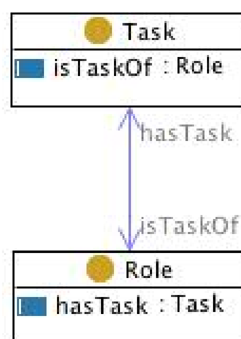


Figure 4.46: The task role CP

Example: the administrator is in charge of making the configuration of the system, while the authors will write the content to be published. Figure 4.47 shows the UML diagram of this scenario.

Elements: the **task role** CP consists of the following elements.

- `Task`, an event type (concept) that classifies an action to be executed.
- `Role`, a concept that classifies an object.
- `hasTask`, a relation between roles and tasks, e.g. 'students have the duty of giving exams' (i.e. the Role 'student' hasTask the Task 'giving exams'). `isTaskOf` is its inverse.

Consequences: tasks and roles are in the domain of your ontology, and you can talk about them, e.g. in order to plan or configure projects, systems, or organizations.

2006–2008 © Copyright lies with the respective authors and their institutions.

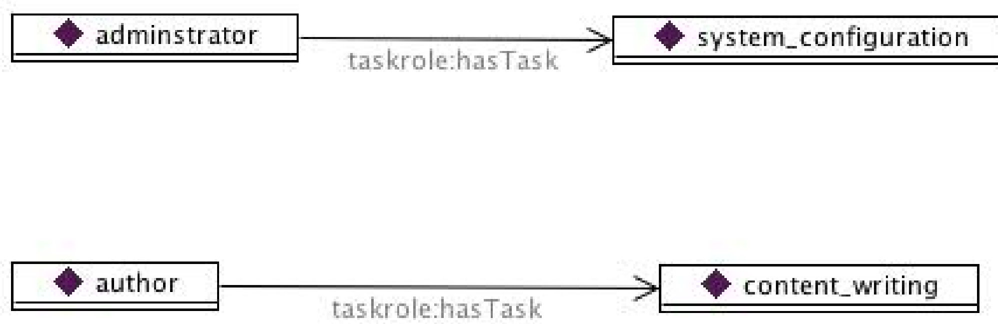


Figure 4.47: The task role example scenario: system configuration (administrator) and content writing (author).

Related patterns: usually it is composed with **object role** (section 4.1.7) or **agent role** (section 4.6.2) CP. It is a component of **basic plan description** CP (section 4.6.5).

Building block: `odp:taskrole.owl`

4.6.4 Time-Indexed Person Role

Name: time-indexed person role.

Intent: to represent a person that plays a certain role at a certain time interval (or at a certain place, in a certain way, etc.).

Examples: who was playing a certain role at a certain time interval? When did a certain person play this role?

Diagram: Figure 4.48 shows the UML diagram of this CP.

Example: Valentina Presutti was a Ph.D student in 2005 and a junior researcher in 2006. Figure 4.49 shows the UML diagram of this scenario.

Elements: the **time-indexed person role** CP consists of the elements of the CPs it reuses i.e., **n-ary classification** CP (section 4.1.5) and **agent role** CP (section 4.6.2). The following elements are defined locally.

- `Entity`: anything: real, possible, or imaginary, which some modeler wants to talk about for some purpose.
- `Person`: persons in commonsense intuition, i.e. either as physical agents (humans) or social persons.
- `Role`: a concept that classifies a person.
- `TimeInterval`: any region in a dimensional space that represents time.

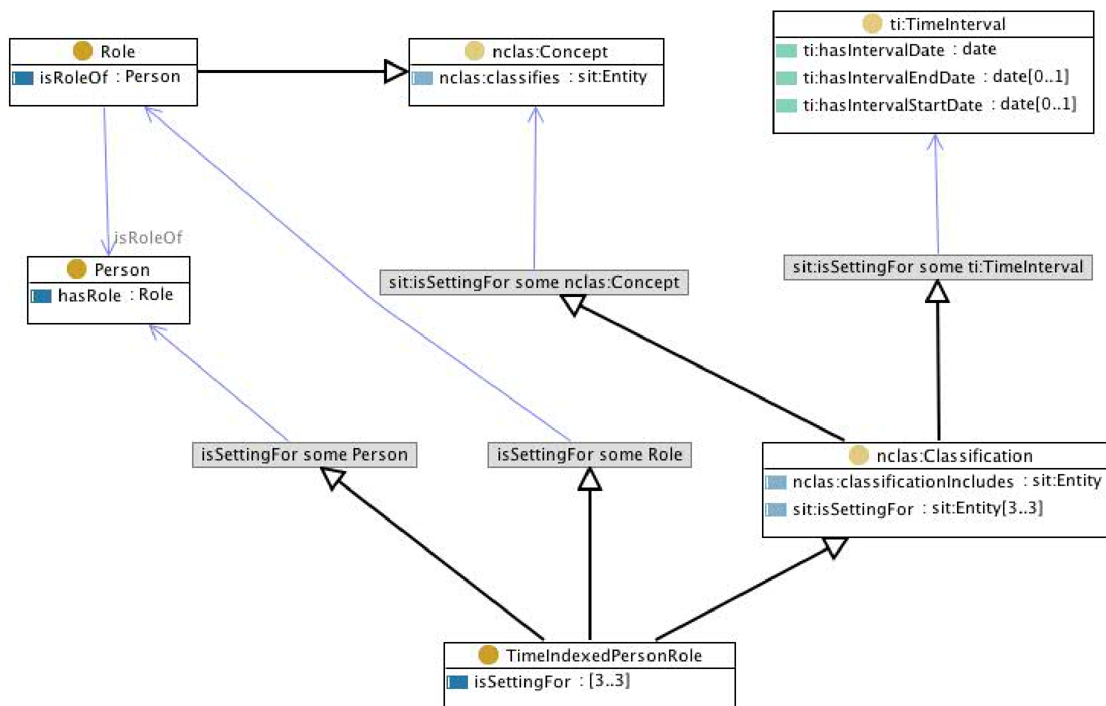


Figure 4.48: The time indexed person role CP

- `TimeIndexedPersonRole`: a situation that expresses time indexing for the relation between persons and the 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` is its inverse
- `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` is its inverse.

Consequences: the CP allows to assign a time interval to roles played by persons.

Related patterns: it is the specialization of **n-ary classification** (section 4.1.5) and **agent role** (section 4.6.2) CPs.

Building block: `odp:timeindexedpersonrole.owl`

4.6.5 Basic Plan Description

Name: basic plan description.

Intent: to represent the conceptualization of a plan, the concepts that are included in or defined by such a conceptualization, and the relations between them.

2006–2008 © Copyright lies with the respective authors and their institutions.

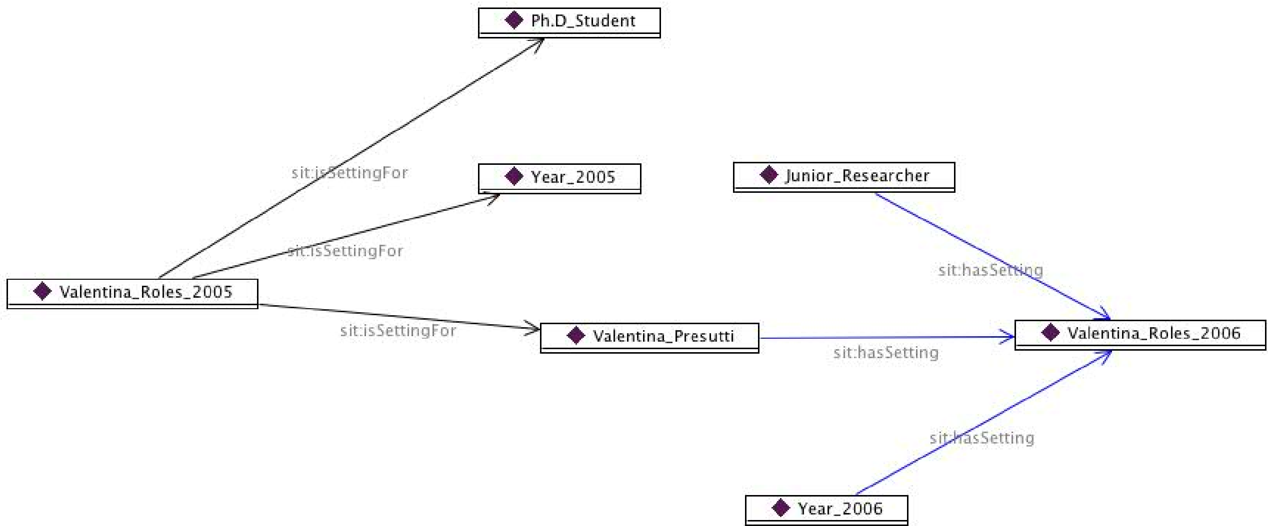


Figure 4.49: The time indexed person role example scenario.

Extracted from: this CP is the composition of other CPs, however it reflects a fragment of the `loa:PlansLite.owl` ontology.

Requirements: what are the concepts involved in this plan? What is the main goal/objective of this plan? What is(are) the goal(s) of this plan? What are the roles, tasks, and parameter that are defined by this plan?

Diagram: Figure 4.50 shows the UML diagram of this CP.

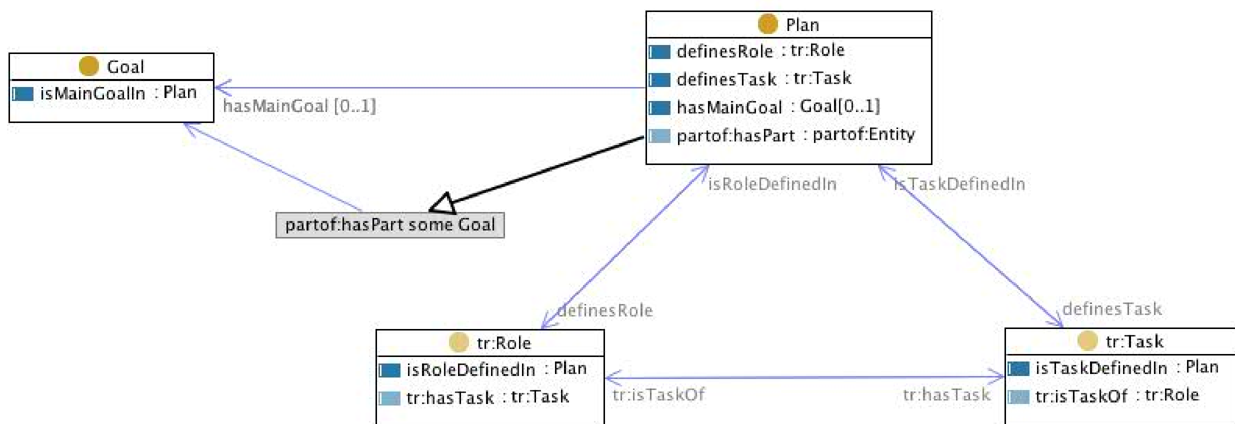


Figure 4.50: The basic plans description CP

Example: in order to write a paper, the editor has to perform a grammar check, and put all sections together. The authors have to write the content. Content writing has to be finished by the internal deadline, while the other tasks start after the internal deadline. The grammar check is the last thing to do. Figure 4.51 shows the UML diagram of this scenario.

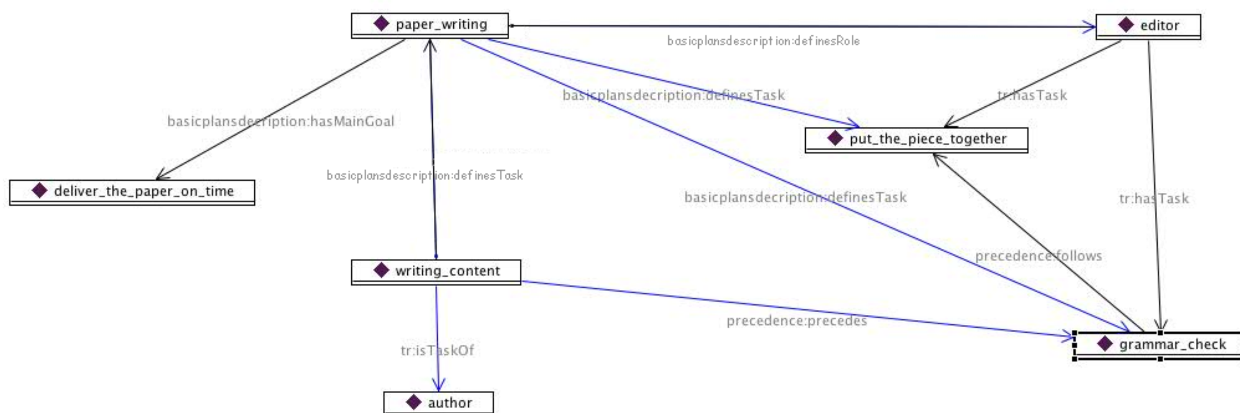


Figure 4.51: The basic plan description example scenario.

Elements: the **basic plan description** CP consists of the elements of its components i.e., **description** (section 4.1.2), **parameter** (section 4.4.3), **part of** (section 4.2.1), and **task role** (section 4.6.3) CPs. The following elements are defined locally.

- `Goal`, the description of a situation that is desired by an agent (physical, social, computational), and usually associated with a plan that describes how to actually achieve it.
- `Plan`, a description having an explicit goal, to be achieved by executing the plan.
- `isRoleDefinedIn`, a relation between a description and a role, e.g. the role 'Ingredient' is defined in the recipe for a cake. `definesRole` is its inverse.
- `isTaskDefinedIn`, a relation between a description and a task, e.g. the task 'bake' is defined in a recipe for a cake. `definesTask` is its inverse.
- `isMainGoalIn`, a main goal characterizes the overall plan, and can be defined as a goal that is part of a plan but not of one of its subplans. `hasMainGoal` is its inverse.

Consequences: plan descriptions and the concepts involved are put in the domain of your ontology, and can be modelled as first-order entities. This is the same domain of discourse as the resources, time, space, etc. that are needed for the execution of that plan (section 4.6.6).

Related patterns: it expands the composition of other CPs: **description** (section 4.1.2), **parameter** (section 4.4.3), **part of** (section 4.2.1), and **task role** (section 4.6.3).

Building block: `odp:basicplandescription.owl`

4.6.6 Basic Plan Execution

Name: basic plan execution.

Intent: to represent the execution of a plan and the entities that are involved in that execution.

2006–2008 © Copyright lies with the respective authors and their institutions.

Extracted from: this CP is the composition of other CPs as indicated below. However, it reflects a fragment of the loa:PlansLite.owl and loa:DUL.owl ontologies.

Requirements: which actions, agents, objects, and regions are in the setting of this plan execution? Which agents in the setting of this plan execution do perform this action? Which actions in the settings of this plan execution are performed by this agent? Which is the value of the parameter that is in the setting of this plan execution?

Diagram: Figure 4.52 shows the UML diagram of this CP.

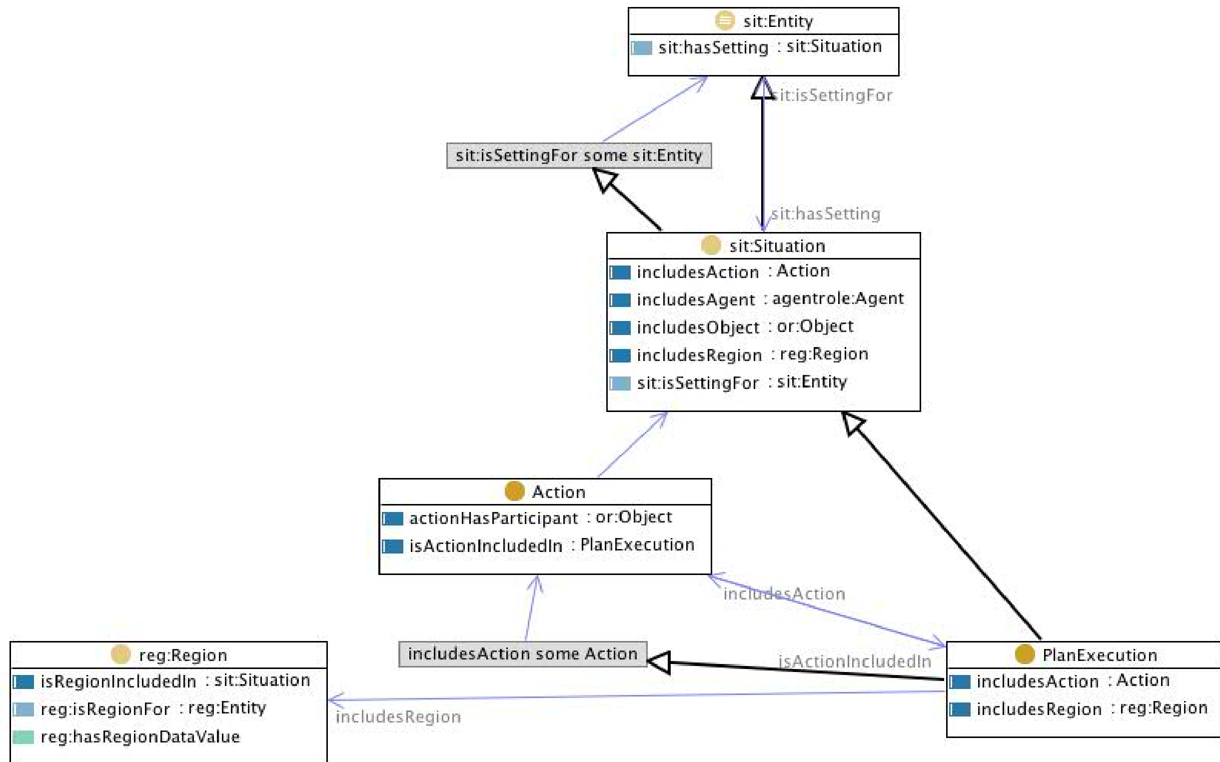


Figure 4.52: The basic plan execution CP

Example: Aldo performed the grammar check for the ISWC paper by the end of June. Figure 4.53 shows the UML diagram of this scenario.

Elements: the **basic plan execution** CP consists of the the elements of its components i.e., **situation** (section 4.1.3), **participation** (section 4.5.1), **agent role** (section 4.6.2), and **region** (section 4.4.1). The following elements are defined locally.

- Action, an event with at least one agent that is participant in, and that executes a task that is defined in a plan.
- PlanExecution, situations that proactively satisfy a plan.
- isActionIncludedIn, a relation between actions and plan executions. includesAction is its inverse.

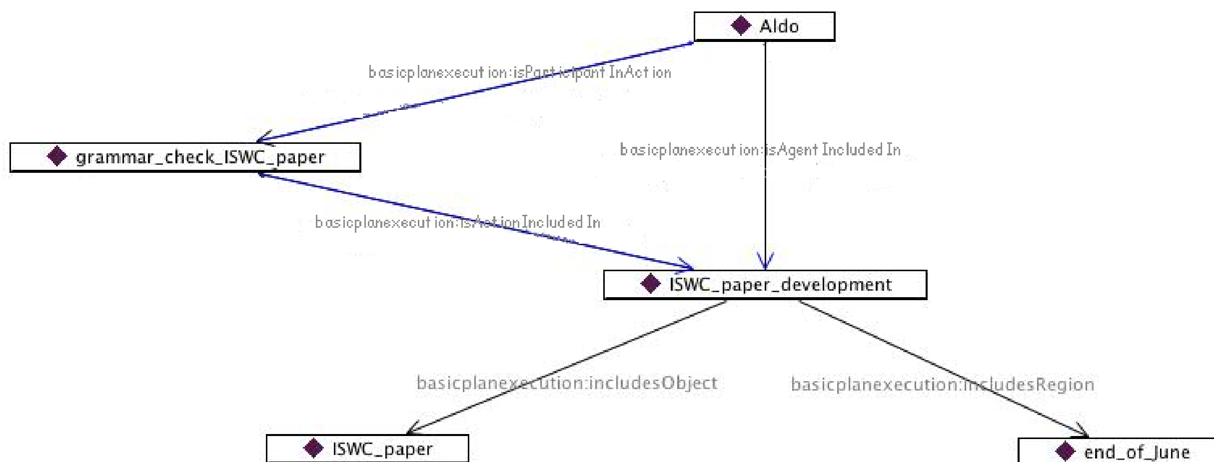


Figure 4.53: The basic plans execution example scenario: the ISWC paper writing.

- `isAgentIncludedIn`, a relation between agents and plan executions. `includesAgent` is its inverse.
- `isObjectIncludedIn`, a relation between objects and plan executions. `includesObject` is its inverse.
- `isRegionIncludedIn`, a relation between regions and plan executions. `includesRegion` is its inverse.
- `isParticipantInAction`, a relation between an object and a action, e.g. 'a mass of snow is a participant in an avalanche', 'an agent, some sugar, flour, etc. are participants in the cooking of a cake'. `ActionHasParticipant` is its inverse.

Consequences: plan executions and elements involved in their settings are put in the same domain of discourse.

Related patterns: this CP is the composition of **situation** (section 4.1.3), **participation** (section 4.5.1), **agent role** (section 4.6.2) and **region** (section 4.4.1).

Building block: `odp:basicplansexecution.owl`

4.6.7 Basic Plan

Name: basic plan.

Intent: to represent description of plans, their executions, the relations between them and their related concepts and entities.

Also Known as: description and execution of plans.

2006–2008 © Copyright lies with the respective authors and their institutions.

Requirements: which plan description does this plan execution satisfy? What executions do satisfy this plan description? Which objects are classified by these roles from this plan? Which actions are classified by this task from this plan? which region is parametrized by this parameter from this plan?

Diagram: Figure 4.54 shows the UML diagram of this CP.

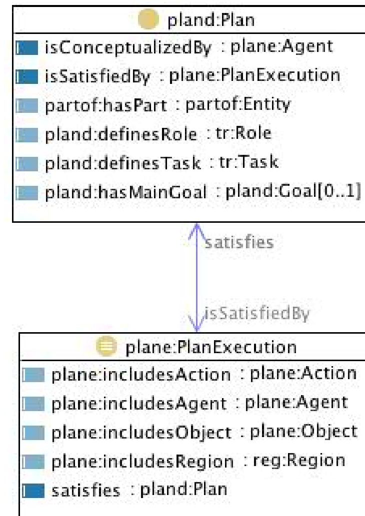


Figure 4.54: The basic plans CP

Example: we wrote our ISWC paper by following our plan for writing, including role and task distribution and parameters for length, deadlines, etc.

Elements: the **basic plan** CP consists of the elements of its components i.e., **basic plan description** (section 4.6.5) and **basic plan execution** (section 4.6.6). The following elements are defined locally.

- `GoalSituation`, a goal situation is a situation that satisfies a goal. A goal situation is not part of a plan execution. This helps to account for the following cases: a) Execution of plans containing abort or suspension conditions, since the plan would be satisfied even if the goal has not been reached, b) Incidental satisfaction, as when a situation satisfies a goal without being intentionally planned (but anyway desired).
- `conceptualizes`, a relation stating that an agent is internally representing a social object . E.g., 'John believes in the conspiracy theory'; 'Niels Bohr created the solar-system metaphor for the atomic theory'; 'Jacques assumes all swans are white'; 'the task force members share the attack plan'. `isConceptualizedBy` is its inverse.
- `classifies`, a relation between a Concept and an Entity, e.g. the Role 'student' classifies a Person 'John'. `isClassifiedBy` is its inverse.
- `isExecutedIn`, a relation between an action and a task, e.g. 'putting some water in a pot and putting the pot on a fire until the water starts bubbling' executes the task 'boiling'. It is sub-property of `classifies`. `executesTask` is its inverse.
- `parametrizes`, the relation between a parameter, e.g. 'MajorAgeLimit', and a region, e.g. '18_year'. It is sub-property of `classifies`. `isParametrizedBy` is its inverse. For a more data-oriented relation, see hasDataValue

- `isSatisfiedBy`, a relation between a situation and a description, e.g. the execution of a plan satisfies that plan. `satisfies` is its inverse.

Consequences: this CP allows designers to put plan descriptions/concepts and their executions/resources in the same domain of discourse.

Related patterns: it is defined by combining and expanding other CPs i.e., **basic plan description** (section 4.6.5) and **basic plan execution** (section 4.6.6).

Building block: `odp:basicplans.owl`

4.7 Business

4.7.1 Price

Name: price.

Intent: to represent the price of an entity (e.g. product, service) in a certain currency.

Requirements: what is the price of this entity expressed in a certain currency? Which is the currency of the price of this entity? what is the price of this entity?

Diagram: Figure 4.55 shows the UML diagram of this CP.

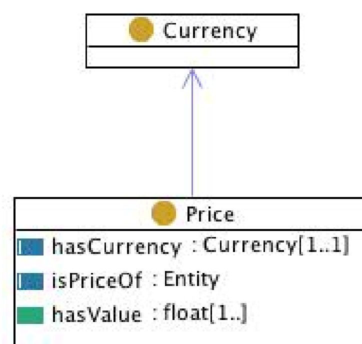


Figure 4.55: The price CP

Example: the european price of the macbook pro 2.2 GHz is $1899,00$ Euro, while the US price is $1999,00$ US Dollars. Figure 4.56 shows the UML diagram of this scenario.

2006–2008 © Copyright lies with the respective authors and their institutions.

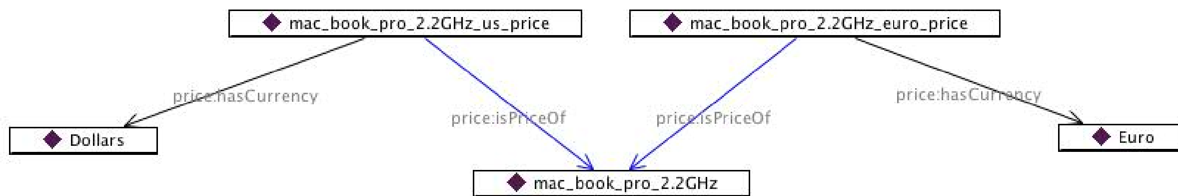


Figure 4.56: The price example scenario: the price of macbook pro 2.2. GHz expressed in Euro and Dollars.

Elements: the **price** CP consists of the following elements.

- `Currency`, a system of money used in a particular country.
- `Entity`, anything that can have a price.
- `Price`, the amount of money expected, required, or given in payment for something.
- `hasCurrency`, a relation between prices and the currencies they are expressed by.
- `hasPrice`, a relation between entities and their prices. `isPriceOf` is its inverse.
- `hasValue`, the numerical value of a price.

Consequences: allows designers to represent the price of an entity in different currencies.

Related patterns: used in **sales and purchase order contracts** (section 4.7.2).

Building block: `odp:price.owl`

4.7.2 Sales and Purchase Order Contracts

Name: sales and purchase order contracts.

Intent: to describe the purchase orders and sales orders, by which an organization buys and sells goods and services, and to extend these to encompass the more general concept of contract or agreement.

Reengineered from: 'Data Model Patterns'. David C. Hay, 1996. ISBN: 00-932633-29-3.

Requirements: who are the parties in this sale (purchase) order contract? What is the line item of this sale (purchase) order contract? What catalogue items are sold (purchased) according to this contract? Which organizations (persons) are involved in this sale (purchase) order?

Diagram: Figure 4.57 shows the UML diagram of this CP.

Example: the "Messagerie Musicali" store buys 1000 pieces of the Queen Greatest hits album from the "EMI music records" company. Figure 4.58 shows the UML diagram of this scenario.

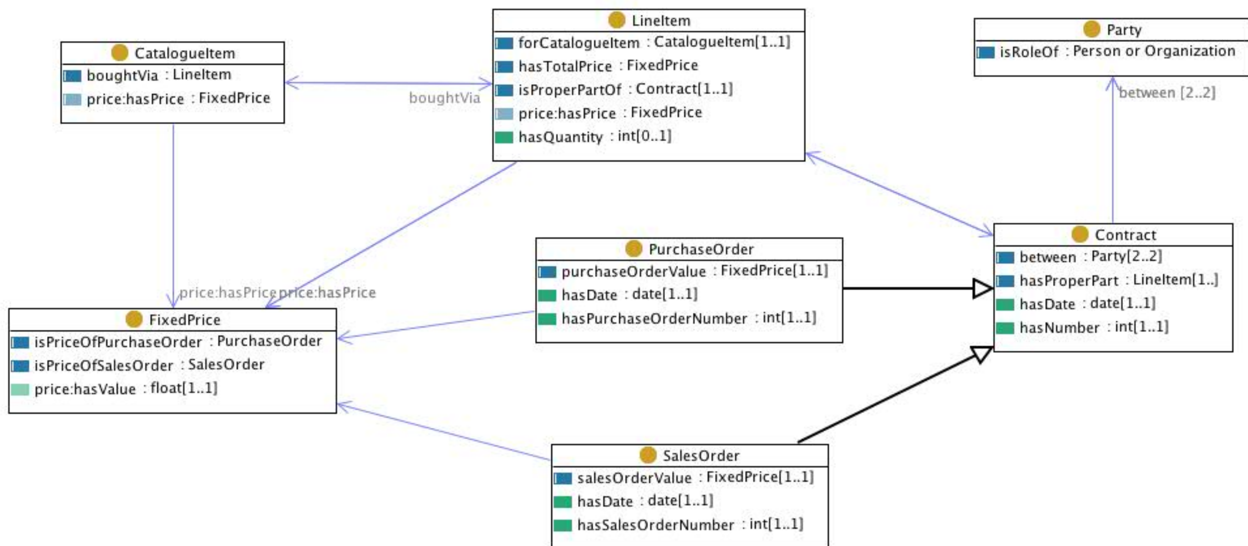


Figure 4.57: The sales and purchase order contracts CP

Elements: the **sales and purchase order contracts** CP consists of the elements of the component it reuses i.e., **price** CP (section 4.7.1); in addition, the following elements are defined locally.

- `Contract`, an agreement between two parties that has at least one line item as a proper part.
- `PurchaseOrder`, an agreement by which an organization buys goods and services.
- `SalesOrder`, an agreement by which an organization sells goods and services.
- `Organization`, an organized body of people with a particular purpose, especially a business, society, association, etc. Examples of organizations are: company, firm, corporation, institution, group, consortium, conglomerate, agency, association, society, informal outfit.
- `Party`, a role played in a contract by either a person or an organization e.g., vendor, seller, etc.
- `Person`, a social person.
- `CatalogueItem`, an individual article or unit, that is part of the set of items that can be sold or bought.
- `LineItem`, a part of a purchase or sales order. A line item refers only to a catalogue item, with which associates a price and a quantity.
- `FixedPrice`, a fixed price.
- `between`, a relation between contracts and its parties.
- `boughtVia`, a relation between a catalogue item and a line item. `forCatalogueItem` is its inverse.
- `hasProperPart`, a relation between contracts and line items. `isProperPartOf` is its inverse.
- `hasRole`, a relation between either persons or organization, and roles. `isRoleOf` is its inverse.
- `hasTotalPrice`, a relation between line items and their total price. `isTotalPrice` is its inverse.
- `purchaseOrderValue`, a relation between purchase orders and their fixed price. `isPriceOfPurchaseOrder` is its inverse.

2006–2008 © Copyright lies with the respective authors and their institutions.

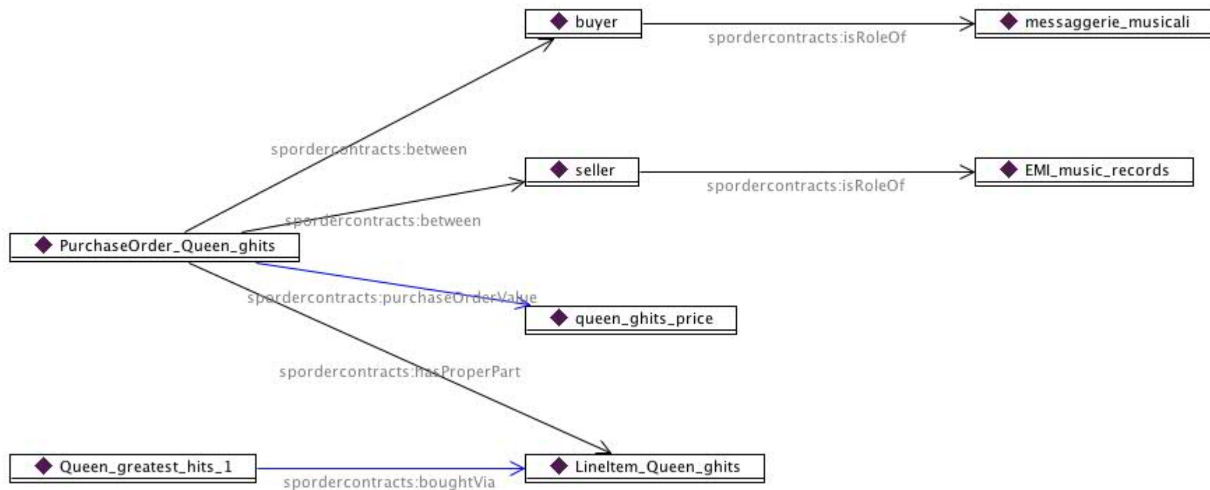


Figure 4.58: The sales and purchase order contracts example scenario: a purchase of Queen Greatest hits album.

- `salesOrderValue`, a relation between sales orders and their fixed price. `isPriceOfSalesOrder` is its inverse.
- `hasDate`, the date of a contract.
- `hasNumber`, the number of a contract.
- `hasQuantity`, the quantity of a line item.

Consequences: this CP allows designers to put sales and purchase order in the same domain of discourse of items that are involved in the sales or purchase.

Related patterns: reuses the **price** CP (section 4.7.1).

Building block: `odp:salespurchaseordercontracts.owl`

4.8 Time

4.8.1 Time Interval

Name: time interval.

Intent: to represent time intervals.

Requirements: what is the end time of this interval? What is the starting time of this interval? What is the date of this time interval?

Diagram: Figure 4.59 shows the UML diagram of this CP.

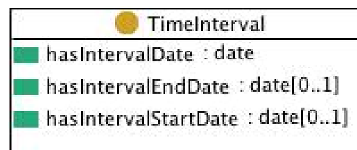


Figure 4.59: The time interval CP

Example: the time interval "January 2008" starts at $2008 - 01 - 0$ and ends at $2008 - 01 - 31$. Figure 4.60 shows the UML diagram of this scenario.

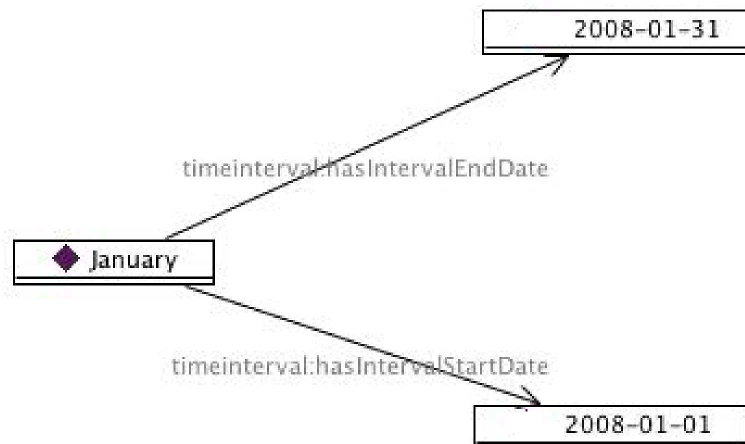


Figure 4.60: The time interval example scenario

Elements: the **time interval** CP consists of the following elements.

- `TimeInterval`, any region in a dimensional space that represents time.
- `hasIntervalDate`, a datatype property that encodes values from `xsd:date` for a time interval; a same time interval can have more than one `xsd:date` value: begin date, end date, date at which the interval holds, as well as dates expressed in different formats: `xsd:gYear`, `xsd:dateTime`, etc.
- `hasIntervalEndDate`, the end date of a time interval.
- `hasIntervalStartDate`, the start date of a time interval.

Consequences: the dates of the time interval are not part of the domain of discourse, they are datatype values. If there is the need of reasoning about dates this CP should be used in composition with the **region** CP (section 4.4.1).

Related patterns: it is a component of **time-indexed person role** (section 4.6.4), **time indexed part of** (section 4.2.2), and can be composed with other CPs when temporal aspects need to be represented.

Building block: `odp:timeinterval.owl`

2006–2008 © Copyright lies with the respective authors and their institutions.

4.9 Space

4.9.1 Move

Name: move.

Intent: to represent the action of moving a physical object from a place to another.

Extracted and Reengineered from: `http://cidoc.ics.forth.gr/cidoc_v4.2.owl`

Requirements: which is the destination of this object? where does this object come from? which is the itinerary of this object?

Diagram: Figure 4.61 shows the UML diagram of this CP.

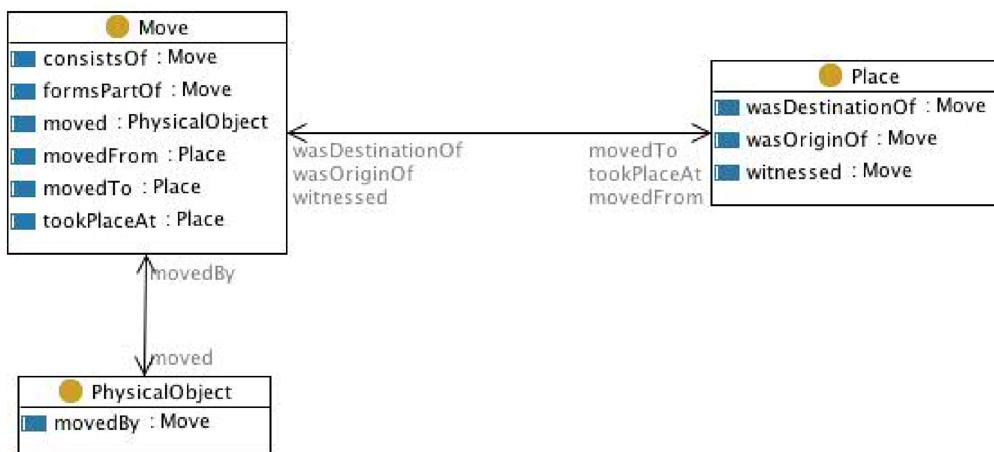


Figure 4.61: The move CP

Example: Monet's painting "Impression sunrise" was moved for the first Impressionist exhibition in 1874. Figure 4.62 shows the UML diagram of this scenario.

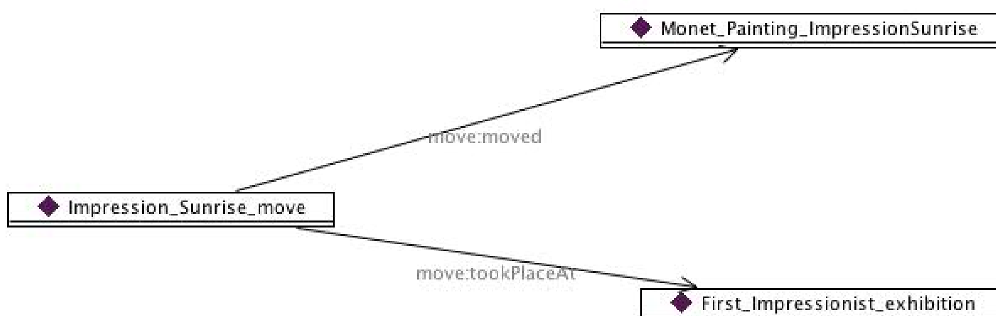


Figure 4.62: The move example scenario

Elements: the **move** CP consists of the following elements.

- `Move`, comprises changes of the physical location of a physical object. Note, that the class move inherits the property took place at a place. This property should be used to describe the trajectory or a larger area within which a move takes place, whereas the properties moved to (was destination of), moved from (was origin of) describe the start and end points only. Moves may also be documented to consist of other moves (via consists of (forms part of)), in order to describe intermediate stages on a trajectory. In that case, start and end points of the partial moves should match appropriately between each other and with the overall event.

- `PhysicalObject`, comprises items of a material nature that are units for documentation and have physical boundaries that separate them completely in an objective way from other objects. The class also includes all aggregates of objects made for functional purposes of whatever kind, independent of physical coherence, such as a set of chessmen. Typically, instances of physical object can be moved (if not too heavy). The decision as to what is documented as a complete item, rather than by its parts or components, may be a purely administrative decision or may be a result of the order in which the item was acquired.

- `Place`, comprises extents in space, in particular on the surface of the earth, in the pure sense of physics: independent from temporal phenomena and matter. The instances of place are usually determined by reference to the position of "immobile" objects such as buildings, cities, mountains, rivers, or dedicated geodetic marks. A place can be determined by combining a frame of reference and a location with respect to this frame. It may be identified by one or more instances of place appellation. It is sometimes argued that instances of place are best identified by global coordinates or absolute reference systems. However, relative references are often more relevant in the context of cultural documentation and tend to be more precise. In particular, we are often interested in position in relation to large, mobile objects, such as ships. For example, the place at which Nelson died is known with reference to a large mobile object, H.M.S Victory. A resolution of this place in terms of absolute coordinates would require knowledge of the movements of the vessel and the precise time of death, either of which may be revised, and the result would lack historical and cultural relevance. Any object can serve as a frame of reference for place determination. The model foresees the notion of a "section" of an physical object as a valid place determination.

- `consistsOf`, describes the decomposition of an instance of move into discrete, subsidiary move. The sub-moves into which the move is decomposed form a logical whole, although the entire picture may not be completely known, and the sub-moves are constitutive of the general move. `formsPartOf` is its inverse.

- `moved`, identifies the physical object that is moved during a move event. The property implies the object's passive participation. In reality, a move must concern at least one object. `movedBy` is its inverse.

- `tookPlaceAt`, describes the spatial location of a move. The related place should be seen as an approximation of the geographical area within which the phenomena that characterize the move in question occurred. took place at (witnessed) does not convey any meaning other than spatial positioning (generally on the surface of the earth). `witnessed` is its inverse.

- `movedFrom`, identifies the starting place of a move. A move will be linked to an origin, such as the move of an artifact from storage to display. A move may be linked to many origins. In this case the move describes the picking up of a set of objects. The area of the move includes the origin, route and destination. `wasOriginOf` is its inverse.

- `movedTo`, identifies the destination of a move. A move will be linked to a destination, such as the move of an artifact from storage to display. A move may be linked to many terminal instances of places. In

2006–2008 © Copyright lies with the respective authors and their institutions.

this case the move describes a distribution of a set of objects. The area of the move includes the origin, route and destination. `wasDestinationOf` is its inverse.

Consequences: this CP allows designers to represent moves of physical objects as paths that can be divided into many parts. The temporal aspect is not modeled. In order to address temporal indexing it might be composed with the **time interval** CP (section 4.8.1).

Building block: `odp:move.owl`

4.10 Life Science

4.10.1 Linnean Taxonomy

Name: Linnean taxonomy.

Intent: to represent the layered classification invented by Linnaeus in the XVIII century. The schema underlying this pattern can be applied to any kind of layered classification.

Reengineered from: `http://en.wikipedia.org/wiki/Linnaean_taxonomy` `http://en.wikipedia.org/wiki/Taxon`, and `http://en.wikipedia.org/wiki/Taxonomic_rank`.

Requirements: what is the taxon of this (group of) organism(s)?

Diagram: Figure 4.63 shows the UML diagram of this CP.

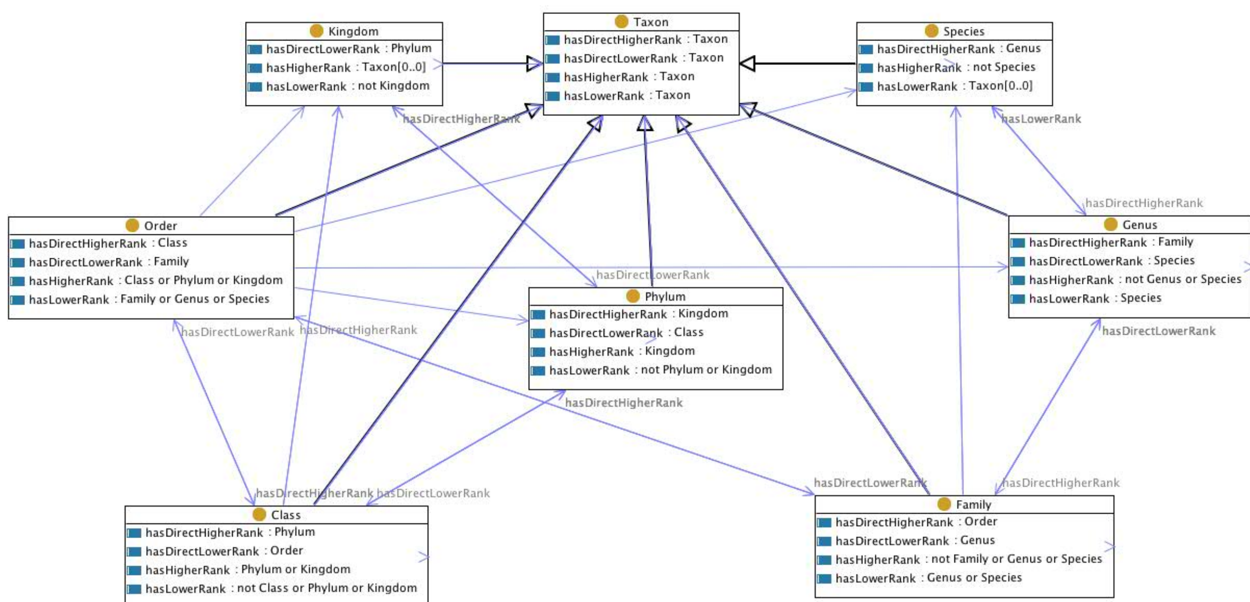


Figure 4.63: The Linnean taxonomy CP

Example: Homo is a Genus of Family Hominidae of Order Primates. Figure 4.64 shows the UML diagram of this scenario.

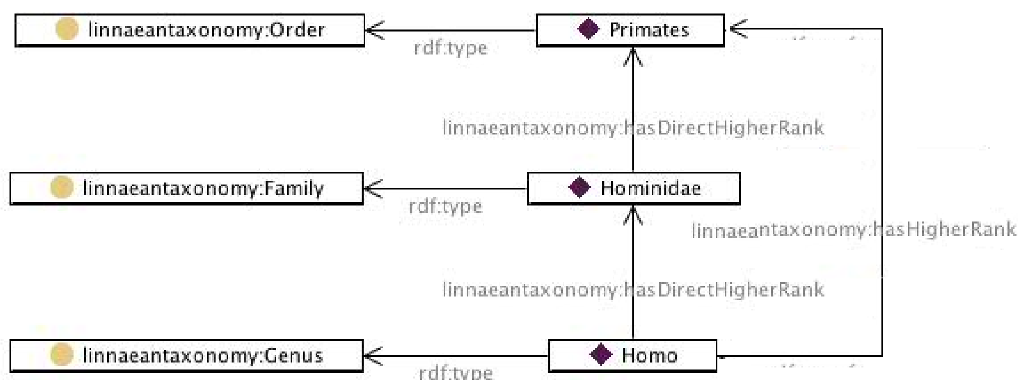


Figure 4.64: The Linnean taxonomy example

Elements: the **linnean taxonomy** CP consists of the following elements.

- `Taxon`, a concept denoting a type of organism or of a group of organisms.
- `Kingdom`, the highest traditional taxon. E.g., in the case of humans the kingdom is ‘Animalia’.
- `Phylum`, the second highest traditional taxon. E.g., in the case of humans the phylum is ‘Chordata’.
- `Class`, the third highest traditional taxon. E.g., in the case of humans the class is ‘Mammalia’.
- `Order`, the fourth highest traditional taxon. E.g., in the case of humans the order is ‘Primates’.
- `Family`, the fifth highest traditional taxon. E.g., in the case of humans the family is ‘Hominidae’.
- `Genus`, the sixth highest traditional taxon. E.g., in the case of humans the genus is ‘Homo’.
- `Species`, the lowest traditional taxon. E.g., in the case of humans the species is ‘Homo sapiens’.
- `hasHigherRank`, relates two taxa, where the first is more specific than the second..`hasLowerRank` is its inverse.
- `hasDirectHigherRank`, sub-property of `hasHigherRank`. It relates two taxa, where the first is more specific than the second and there is no other taxon that is both more general than the first and more specific than the second. `hasDirectLowerRank` is its inverse.

Building block: `odp:linnean taxonomy.owl`

4.11 Multimedia

4.11.1 Multimedia Data Segment Decomposition

Name: multimedia data segment decomposition.

2006–2008 © Copyright lies with the respective authors and their institutions.

Intent: to represent decompositions of multimedia content into segments (inspired by MPEG-7). A segment is the most general abstract concept in MPEG-7 and can refer to a region of an image, a piece of text, a temporal scene of a video or even to a moving object tracked during a period of time.

Extracted from: `http://multimedia.semanticweb.org/COMM/multimedia-ontology.owl`

Requirements: which are the roles involved in this segmentation algorithm? which media does realizes this multimedia data? to which segment algorithm does this segment decomposition comply?

Diagram: Figure 4.65 shows the UML diagram of this CP.

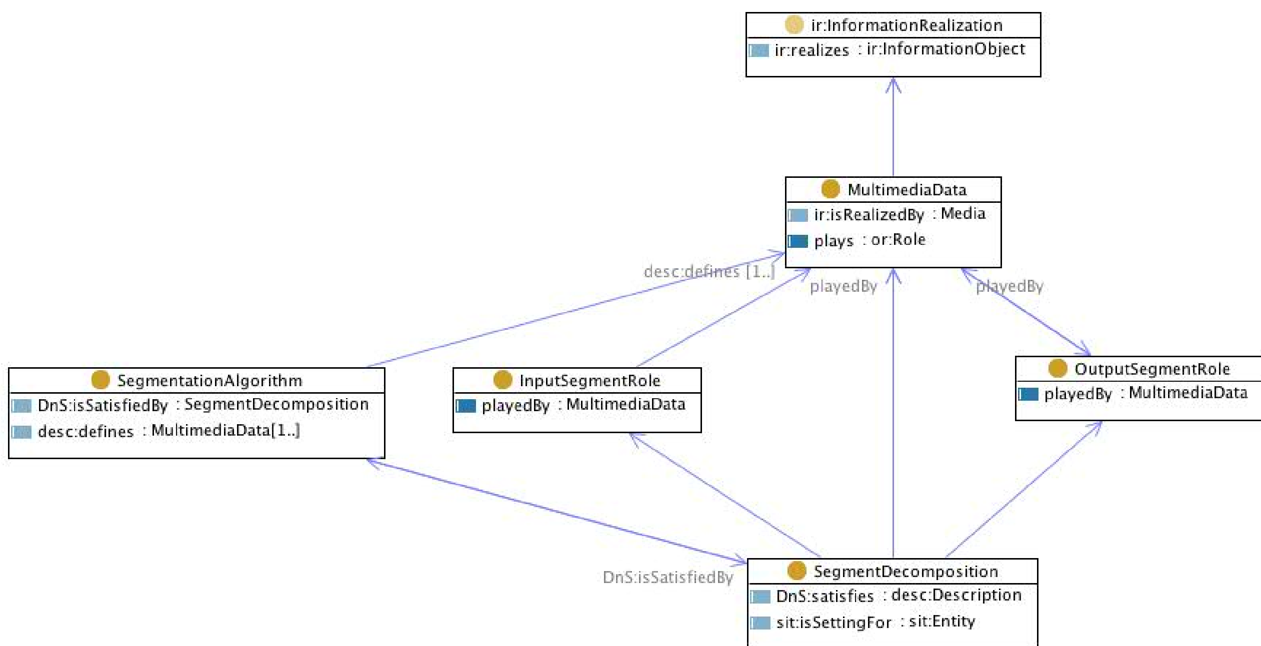


Figure 4.65: The multimedia data segment decomposition CP

Example: The Figueiredo's segmentation algorithm is used for image segmentation.

Elements: the **multimediadatasegmentdecomposition** CP consists of the elements of its components. The following elements are defined locally.

- `MultimediaData`, the main idea of MPEG-7 is to describe digital multimedia content by the means of some (XML-) metadata. Thus, we will consider multimedia data as a subconcept of digital data that is described by some other digital data. Multimedia data is an abstract base concept that has to be specialized for specific media types, e.g. image data carries the visual content of an image. Multimedia data is realized by some physical media.
- `Media`, multimedia data is realized by some physical media. An image is an example for a media. In the case of a JPEG image we would consider JPEG as the corresponding information encoding system that orders the physical image. For the multimedia ontology it is important to include the physical realization of multimedia data as some MPEG-7 description tools have to be attached to the

physical media of multimedia content. This holds for the media information description tools that are specified in MPEG-7 part 5, clause 8.

- `SegmentationAlgorithm`, a segmentation algorithm decomposes a multimedia data entity in one or more other multimedia data entities. It always defines exactly one input role for the multimedia entity that is decomposed.
- `SegmentDecomposition`, the decomposition of a multimedia data entity is considered as a situation which satisfies a description that is provided by a segmentation algorithm or a method which has been applied to perform the segmentation. The application of methods corresponds to manual or semi-automatic segmentations. An example would be the handcrafted selection of segments using a magic wand tool. The segment decomposition corresponds to the MPEG-7 'SegmentDecompositionType'.
- `InputSegmentRole`, this role has to be played by within a decomposition situation. The role identifies the multimedia data entity that is being decomposed into segments.
- `OutputSegmentRole`, a multimedia data entity that plays this role represents a segment in a segment decomposition. Segment roles are only defined by methods or algorithms. The segment role corresponds to the name of the MPEG-7 'SegmentType'. The MPEG-7 description tools that are listed inside the 'SegmentType' can be attached to the multimedia data entities that play a segment role.
- `plays`, This is the immediate relation between roles and object. A role classifies the position (function, use, relevance, ...) of an object within a context (description). Roles can be ordered, interdependent, at different layers, etc. `playedBy` is its inverse.

Consequences: The CP allows designers to describe segmentation algorithm for multimedia objects and their executions. Furthermore, it allows to distinguish between multimedia data and their realizations.

Related patterns: it specializes the composition of the **descriptions and situations**, **information realization**, and **object role**.

Building block: `odp:multimediadatasegmentdecomposition.owl`

4.12 Towards a Web Portal of CPs

For reusability in ontology design, it is key to have a repository of CPs with appropriate services, e.g. where CPs can be added and retrieved, and to guarantee that published CPs have a high level of quality.
With the above principles in mind, we have set up the Ontology Design Patterns Web portal[4] (ODP), where CPs are collected, classified, described with a specific template (the one used in this chapter for the catalogue entries), and available for download (as building blocks). They comply to the definition of CP given in section 2.2.6. The Web portal is open to contribution from all Web users. They are only required to register in order to have authoring rights. A lightweight workflow is implemented in order to guarantee both quality and openness. Below we give a brief description of the ODP and its quality workflow. The ODP will be available for testing to a restrict group of people from the NeOn community starting from end of February. Our aim is to collect feedback, fix possible problems, and add features based on this testing period. The ODP will be presented at the next European Semantic Web Conference in the context of a NeOn event, and also supported during a workshop at the same conference on Reuse and Reengineering over the Semantic Web. As a temporary resource, a flat list of CPs is available for download (in the form of OWL building block files) at http://wiki.loa-cnr.it/index.php/LoaWiki:CPRepository .

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

2006–2008 © Copyright lies with the respective authors and their institutions.

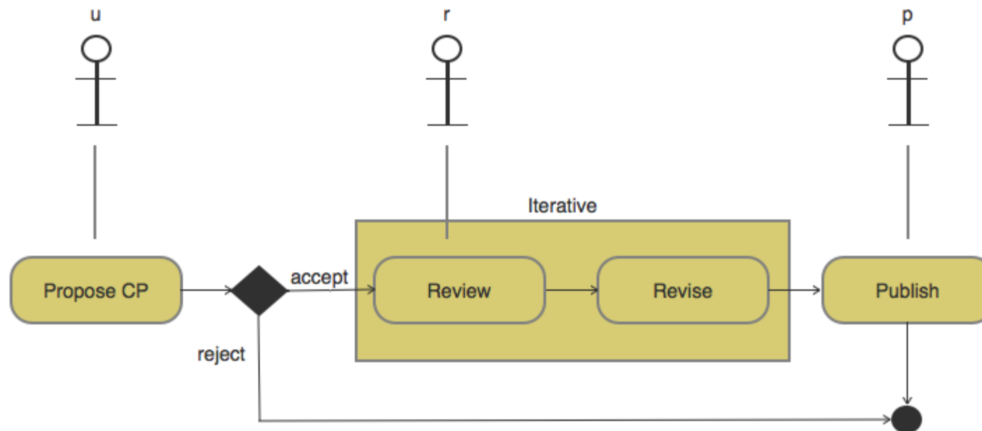


Figure 4.66: The ODP publishing and quality workflow.

ODP organization and quality workflow

ODP users can play four different roles, each associated with an authoring rights policy:

- *Registered user* u: any user who is registered to the ODP;
- *Reviewer* r: a member of the quality committee;
- *Publisher* p*)*: a member of the CP publishers committee.

The ODP content is organized in three areas: *Catalogue*, *Reviews*, and *Talk*. All areas are readable by any Web user (including users who are not registered). Authoring rights are policed as follows:

- *Proposing and Discussing (PD).* All u users can raise and participate in discussions about e.g., design problems, experiences with CPs, etc. Furthermore, specific templates are available to submit a new CP to the reviewers committee with the aim of publishing it, as well as to propose modeling issues and discuss practical experiences.
- *Reviewing (R).* The content of this area relates to CPs that are under review. Each CP is assigned to a representative of r users who are in charge of describe the rationales that motivates revisions of proposed CP i.e., peer reviewing. CPs have dedicated pages where all discussions, decisions, and rationales about their review process are kept. This area is accessible for authoring to r users.
- *Catalogue (C).* This area contains the catalogue entries, which are described in terms of the entry template used for the catalogue and presented earlier in this chapter; p users can author pages in this area.

Figure 4.66 sketches the ODP quality and publishing workflow, and associated roles.

Appendix A

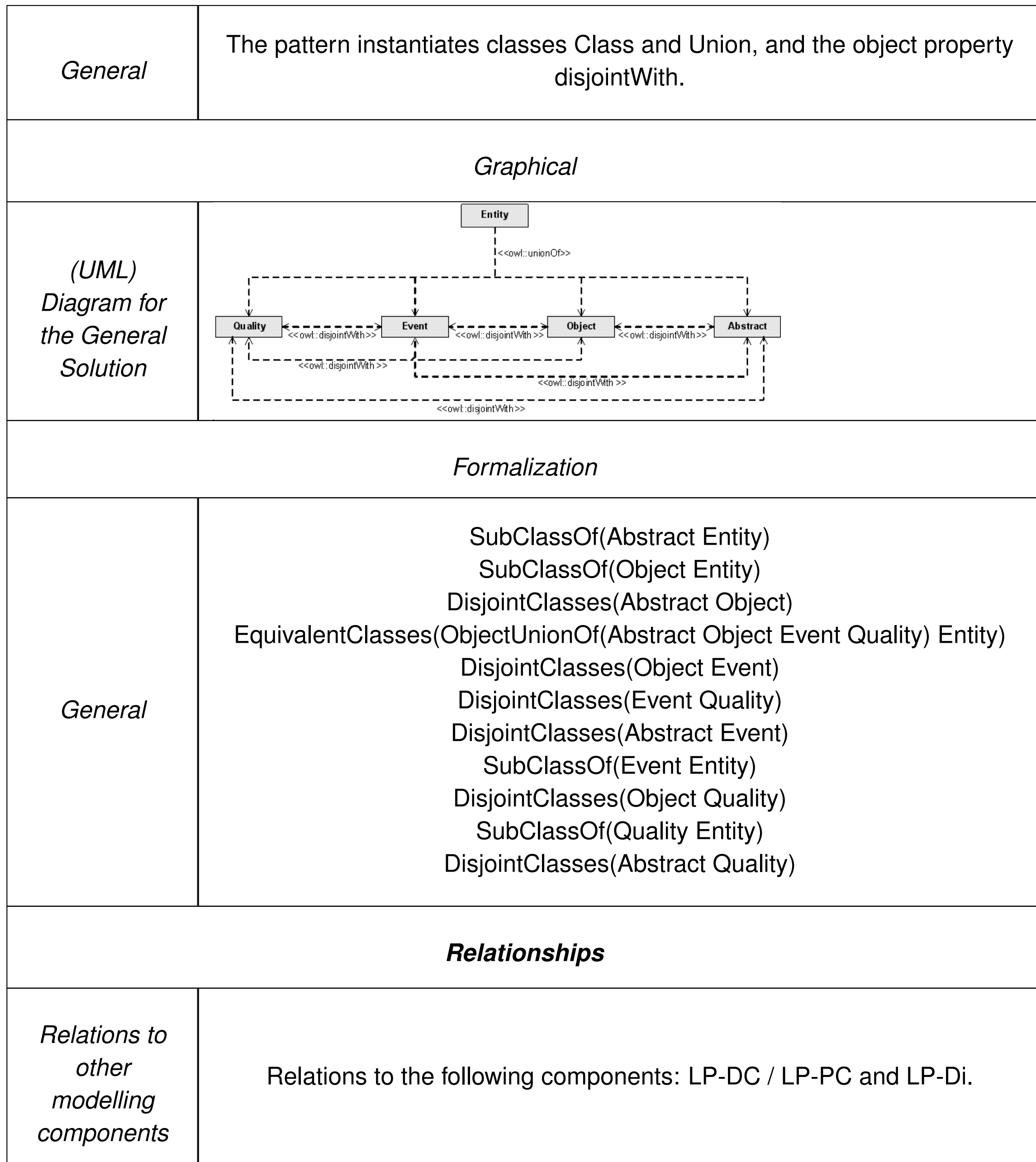
Software Engineering templates

In this appendix, the CPs presented in the catalogue of chapter 4 are put in an alternative presentation format, which is compliant to that defined in deliverable D5.1.1 [SFBG+07], and that is tailored to software engineers.

A.1 General

Slot	Value
General Information	
Name	Types Of Entities
Identifier	CP-TOE-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that a entity can be only an object, an abstract, an event or a quality.
Examples	Suppose that someone wants to express the formal semantics elements which are mathematical entities.
Ontology Design Pattern	
Informal	

2006–2008 © Copyright lies with the respective authors and their institutions.

General	The pattern instantiates classes Class and Union, and the object property disjointWith.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(Abstract Entity) SubClassOf(Object Entity) DisjointClasses(Abstract Object) EquivalentClasses(ObjectUnionOf(Abstract Object Event Quality) Entity) DisjointClasses(Object Event) DisjointClasses(Event Quality) DisjointClasses(Abstract Event) SubClassOf(Event Entity) DisjointClasses(Object Quality) SubClassOf(Quality Entity) DisjointClasses(Abstract Quality)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC and LP-Di.

Slot	**Value**
General Information	
Name	Description
Identifier	CP-DES-01
Type of Component	Content Pattern (CP)

Use Case	
General	Represents the conceptualization of an idea, notion, role or even a reified class.
Examples	Suppose that someone wants to describe the process of ’preparing a coffee’.
Ontology Design Pattern	
Informal	
General	The pattern consists in a ‘defines’ (and its inverse ‘isDefinedIn’) relation between ‘descriptions’ and ‘concepts’. The pattern should instantiate the classes Class and ObjectProperty and the object properties inverseOf and disjointWith.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	DisjointClasses(Concept Description) InverseObjectProperties(isDefinedIn defines) ObjectPropertyDomain(isDefinedIn Concept) ObjectPropertyRange(isDefinedIn Description) ObjectPropertyRange(defines Concept) InverseObjectProperties(isDefinedIn defines) ObjectPropertyDomain(defines Description)
Relationships	

2006–2008 © Copyright lies with the respective authors and their institutions.

Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP and LP-Di.
Comments	
Comments	Once defined, a Concept can be used in other descriptions.

Slot	Value
General Information	
Name	Situation
Identifier	CP-SI-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express a situation of anything (real, possible, or imaginary) which some modeller wants to talk about for some purpose.
Examples	Suppose that I want to represent that ‘this morning I’ve prepared my coffee with a new fantastic Arabica’.
Ontology Design Pattern	
Informal	
General	The pattern consists in a ‘isSettingFor’ (and its inverse ‘hasSetting’) relation between ‘situations’ and ‘entities’. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the existential restriction someValuesFrom. Also the object properties inverseOf and subClassOf are instantiated.
Graphical	

(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(Situation Entity) SubClassOf(Situation ObjectAllValuesFrom(isSettingFor Entity)) SubClassOf(Situation ObjectSomeValuesFrom(isSettingFor Entity)) InverseObjectProperties(isSettingFor hasSetting) ObjectPropertyDomain(isSettingFor Situation) ObjectPropertyRange(isSettingFor Entity) ObjectPropertyRange(hasSetting Situation) ObjectPropertyDomain(hasSetting Entity)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-UR and LP-ER . This pattern is imported in the CP-NPAR and CP-TIPO ones.

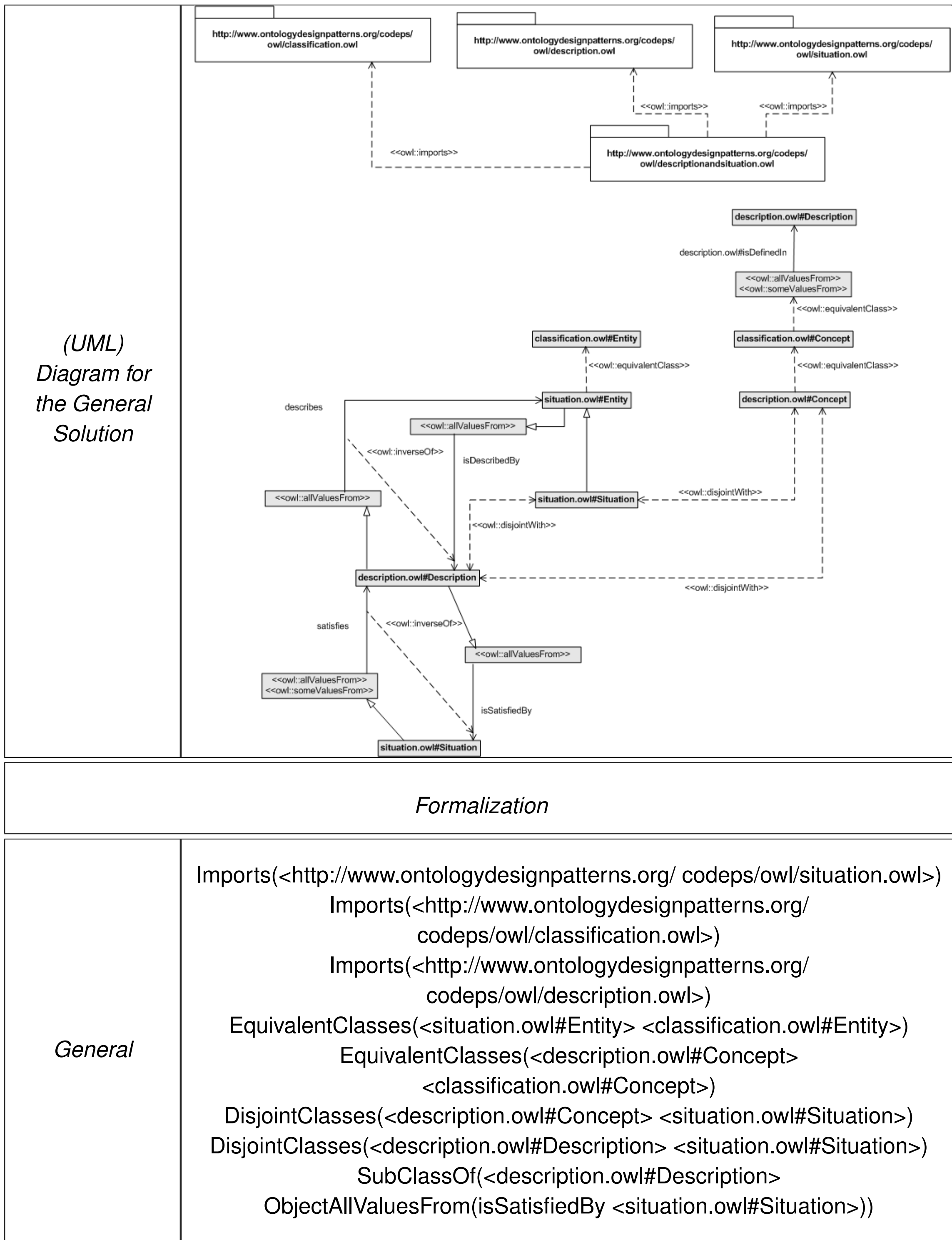
Slot	**Value**
General Information	
Name	Classification
Identifier	CP-Cl-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents the relations between concepts e.g., roles, task, and entities to which concepts are assigned to e.g., person, activities.

2006–2008 © Copyright lies with the respective authors and their institutions.

Examples	Suppose that someone wants to represent that the Role 'student' classifies a Person 'John'
Ontology Design Pattern	
Informal	
General	The pattern consist of the classes 'Entity' and 'Concept' and the relations 'classifies' and 'isClassifiedBy' between 'Entity' and 'Concept'. The pattern should instantiate the classes *Class* and *ObjectProperty* with the universal restriction *allValuesFrom*. The object property *inverseOf* is also instantiated.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(Concept ObjectAllValuesFrom(classifies Entity)) SubClassOf(Entity ObjectAllValuesFrom(isClassifiedBy Concept)) InverseObjectProperties(classifies isClassifiedBy) ObjectPropertyDomain(isClassifiedBy Entity) ObjectPropertyRange(isClassifiedBy Concept) ObjectPropertyDomain(classifies Concept) ObjectPropertyRange(classifies Entity)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC and LP-UR.

Slot	Value
General Information	
Name	Description and Situation
Identifier	CP-DAS-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents conceptualizations i.e., descriptions, and corresponding groundings i.e., situations.
Examples	Suppose that someone wants to represent that the execution of a Plan is compiant to (satisfies) a certain plan description.
Ontology Design Pattern	
Informal	
General	The pattern partially consist of the relations: 'satisfies' between 'Situation' and 'Description' and 'describes' between 'Description' and 'Entity'. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the classes *Class* and *ObjectProperty* with the universal restriction *allValuesFrom* and the existential restriction *someValuesFrom*. The object properties *inverseOf*, *subClassOf* and *equivalentClass* and the relation between ontologies *import* are also instantiated.
Graphical	

2006–2008 © Copyright lies with the respective authors and their institutions.

(UML) Diagram for the General Solution	
Formalization	
General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/situation.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/classification.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/description.owl>) EquivalentClasses(<situation.owl#Entity> <classification.owl#Entity>) EquivalentClasses(<description.owl#Concept> <classification.owl#Concept>) DisjointClasses(<description.owl#Concept> <situation.owl#Situation>) DisjointClasses(<description.owl#Description> <situation.owl#Situation>) SubClassOf(<description.owl#Description> ObjectAllValuesFrom(isSatisfiedBy <situation.owl#Situation>))

	SubClassOf(<description.owl#Description> ObjectAllValuesFrom(describes <situation.owl#Entity>)) SubClassOf(<situation.owl#Situation> ObjectSomeValuesFrom(satisfies <description.owl#Description>)) SubClassOf(<situation.owl#Situation> ObjectAllValuesFrom(satisfies <description.owl#Description>)) SubClassOf(<classification.owl#Concept> ObjectSomeValuesFrom(<description.owl#isDefinedIn> <description.owl#Description>)) SubClassOf(<classification.owl#Concept> ObjectAllValuesFrom(<description.owl#isDefinedIn> <description.owl#Description>)) SubClassOf(<classification.owl#Entity> ObjectAllValuesFrom(isDescribedBy <description.owl#Description>)) ObjectPropertyRange(isSatisfiedBy <situation.owl#Situation>) InverseObjectProperties(satisfies isSatisfiedBy) ObjectPropertyDomain(isSatisfiedBy <description.owl#Description>) ObjectPropertyDomain(describes <description.owl#Description>) InverseObjectProperties(isDescribedBy describes) ObjectPropertyRange(describes <situation.owl#Entity>) ObjectPropertyRange(isDescribedBy <description.owl#Description>) ObjectPropertyDomain(isDescribedBy <situation.owl#Entity>) ObjectPropertyRange(satisfies <description.owl#Description>) ObjectPropertyDomain(satisfies <situation.owl#Situation>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-EQ, LP-UR, LP-ER, AP-MD, CP-SI, CP-DES and CP-CL.

Slot	Value
General Information	
Name	Object - Role
Identifier	CP-OR-01
Type of Component	Content Pattern (CP)

2006–2008 © Copyright lies with the respective authors and their institutions.

Use Case	
General	This pattern represents the relation between objects and the roles they play.
Examples	Suppose that someone wants represents that the person 'John' has role 'student'.
Ontology Design Pattern	
Informal	
General	The pattern consist of the classes 'Role' and 'Object' and the following relations: 'isRoleOf', 'hasRole' and 'isClassifiedBy' between 'Role' between 'Object'. The pattern should instantiate the classes *Class* and *ObjectProperty* with the universal restriction *allValuesFrom*. The object property *inverseOf* is also instantiated.
Graphical	
(UML) Diagram for the General Solution	<<owl::ObjectProperty>> hasRole <<owl::inverseOf>> hasRole «owl::allValuesFrom» <<rdfs::range>> Role «owl::allValuesFrom» Object classification.owl#isClassifiedBy <<rdfs::range>> <<owl::ObjectProperty>> isRoleOf «owl::allValuesFrom» isRoleOf
Formalization	
General	SubClassOf(Role ObjectAllValuesFrom(isRoleOf Object)) SubClassOf(Object ObjectAllValuesFrom(hasRole Role)) SubClassOf(Object ObjectAllValuesFrom(<classification.owl#isClassifiedBy> Role)) InverseObjectProperties(hasRole isRoleOf) ObjectPropertyDomain(isRoleOf Role) ObjectPropertyRange(isRoleOf Object) ObjectPropertyDomain(hasRole Object) ObjectPropertyRange(hasRole Role)
Relationships	

Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP and LP-UR.

A.2 Parts and collections

Slot	Value
General Information	
Name	Part Of
Identifier	CP-PO-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern serves to represent entities formed by other entities.
Examples	Suppose that someone wants to represent that the human body has a brain as part.
Ontology Design Pattern	
Informal	
General	The pattern consists in a class ‘entity’ that have a transitive relation “isPartOf’ (and its inverse ‘hasPart’). The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom. Also the `http://www.w3.org/TR/2004/REC-owl-guide-20040210/` TransitiveProperty and the object property inverseOf are needed.
Graphical	

2006–2008 © Copyright lies with the respective authors and their institutions.

(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(Entity ObjectAllValuesFrom(hasPart Entity)) SubClassOf(Entity ObjectAllValuesFrom(isPartOf Entity)) TransitiveObjectProperty(hasPart) ObjectPropertyDomain(hasPart Entity) ObjectPropertyRange(hasPart Entity) InverseObjectProperties(hasPart isPartOf) TransitiveObjectProperty(isPartOf) ObjectPropertyRange(isPartOf Entity) ObjectPropertyDomain(isPartOf Entity)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP and LP-UR. This pattern is imported in CP-COM and CP-TIPO ones.

Slot	Value
General Information	
Name	Time Indexed Part Of
Identifier	CP-TIPO-01
Type of Component	Content Pattern (CP)
Use Case	
General	The patterns represents a situation that includes at least two Objects and one TimeInterval, at which the part-whole relationship holds.

Examples	My car mounted the Michelin pneumatics last year, now it mounts the Pirelli..
Ontology Design Pattern	
Informal	
General	The pattern consists in a class 'timeIndexedPartOf' that have a relation 'isSettingFor' with 'entiy'. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the existencial restriction someValuesFrom. The object properties equivalentClass, subClassOf and disjointWith is used in this case. The ontology "TimeIndexedPartOf" imports the "PartOf" and "Situation" ones.
Graphical	
(UML) Diagram for the General Solution	http://www.ontologydesignpatternsorg/codeps/owl/situationowl http://www.ontologydesignpatternsorg/codeps/owl/partof.owl <<owl::imports>> <<owl::imports>> http://www.ontologydesignpatternsorg/codeps/owl/timeindexedpartofowl situation.owl#isSettingFor 3..* situationowl#Entity <<owl::equivalentClass>> partof.owl#Entity situation.owl#Situation <<owl::disjointWith >> TimeInterval <<owl::disjointWith >> Object <<owl::disjointWith >> <<owl::allValuesFrom>> TimeIndexedPartOf situation.owl#isSettingFor <<owl::someValuesFrom>> situation.owl#isSettingFor
Formalization	

 © Copyright lies with the respective authors and their institutions.

General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/situation.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/partof.owl>) SubClassOf(TimeInterval <partof.owl#Entity>) DisjointClasses(TimeInterval <situation.owl#Situation>) DisjointClasses(TimeInterval Object) EquivalentClasses(<situation.owl#Entity> <partof.owl#Entity>) SubClassOf(Object <partof.owl#Entity>) SubClassOf(timeindexedpartof ObjectAllValuesFrom (<situation.owl#isSettingFor> <situation.owl#Entity>)) SubClassOf(timeindexedpartof ObjectMinCardinality(3 <situation.owl#isSettingFor>)) SubClassOf(timeindexedpartof ObjectSomeValuesFrom (<situation.owl#isSettingFor> TimeInterval)) SubClassOf(timeindexedpartof ObjectSomeValuesFrom (<situation.owl#isSettingFor> Object)) SubClassOf(timeindexedpartof <situation.owl#Situation>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-SC, LP-UR, LP-ER, LP-Di, LP-EQ, AP-MD, CP-SI and CP-PO.

Slot	Value
General Information	
Name	Componency
Identifier	CP-COM-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that an object can be the compendium of other objects.
Examples	An engine is composed of pieces.
Ontology Design Pattern	

Informal	
General	The pattern consists in a class ‘object’ that have a reflexive relation ‘hasComponent’ (and its inverse ‘isComponentOf’). The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the object property inverseOf. The ontology “Componency” imports the “ParOf” ones.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/partof.owl>) SubClassOf(Object ObjectAllValuesFrom(hasComponent Object)) SubClassOf(Object <partof.owl#Entity>) SubClassOf(Object ObjectAllValuesFrom(isComponentOf Object)) InverseObjectProperties(isComponentOf hasComponent) SubObjectPropertyOf(hasComponent <partof.owl#hasPart>) ObjectPropertyDomain(hasComponent Object) ObjectPropertyRange(hasComponent Object) InverseObjectProperties(isComponentOf hasComponent) ObjectPropertyDomain(isComponentOf Object) SubObjectPropertyOf(isComponentOf <partof.owl#isPartOf>) ObjectPropertyRange(isComponentOf Object)
Relationships	

Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-UR, LP-SC, LP-SP, AP-MD and CP-PO.

Slot	**Value**
General Information	
Name	Constituency
Identifier	CP-CON-01
Type of Component	Content Pattern (CP)
Use Case	
General	The pattern "constituency" is used to represent domains formed by layers.
Examples	We can represent that a social system is conceptualized at a different layer from the persons that constitute it.
Ontology Design Pattern	
Informal	
General	The pattern consists in a class ‘entity’ that have a reflexive relation ‘hasConstituent’ (and its inverse ‘isConstituentOf’). The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the object property inverseOf.
Graphical	
(UML) Diagram for the General Solution	<<owl::ObjectProperty>> hasConstituent <<owl::inverseOf>> hasConstituent «owl::allValuesFrom» <<rdfs:range>> Entity Entity <<rdfs:range>> <<owl::ObjectProperty>> isConstituentOf «owl::allValuesFrom» isConstituentOf

Formalization	
General	SubClassOf(Entity ObjectAllValuesFrom(<http://www.loa-cnr.it/ontologies/DUL.owl#hasConstituent> <http://www.loa-cnr.it/ontologies/DUL.owl#Entity>)) ObjectPropertyRange(hasConstituent Entity) ObjectPropertyDomain(hasConstituent Entity) InverseObjectProperties(isConstituentOf hasConstituent) ObjectPropertyDomain(isConstituentOf Entity) InverseObjectProperties(isConstituentOf hasConstituent) ObjectPropertyRange(isConstituentOf Entity)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP and LP-UR.

Slot	Value
General Information	
Name	Collection Entity
Identifier	CP-CE-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that a collection can be seen as a container for entities that share one or more common properties
Examples	Suppose that someone wants to represent a collection of stone objects.
Ontology Design Pattern	
Informal	

2006–2008 © Copyright lies with the respective authors and their institutions.

General	The pattern consists in a ‘hasMember’ (and its inverse ‘isMemberOf’) relation between ‘collection’ and ‘entity’. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the object property inverseOf.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(Collection ObjectAllValuesFrom(hasMember Entity)) SubClassOf(Entity ObjectAllValuesFrom(isMemberOf Collection)) InverseObjectProperties(isMemberOf hasMember) ObjectPropertyDomain(isMemberOf Entity) ObjectPropertyRange(isMemberOf Collection) InverseObjectProperties(isMemberOf hasMember) ObjectPropertyRange(hasMember Entity) ObjectPropertyDomain(hasMember Collection)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP and LP-UR.

A.3 Semiotics

Slot	Value
General Information	
Name	Intension- Extension
Identifier	CP-IE-01

Type of Component	Content Pattern (CP)
Use Case	
General	This pattern allows representing the intensional expression and extensional reference of information objects.
Examples	This pattern can be used to talk about e.g. entities are references of proper nouns: the proper noun 'Leonardo da Vinci' isAbout the Person Leonardo da Vinci.
Ontology Design Pattern	
Informal	
General	The pattern consists on the one hand in a 'expresses' (and its inverse 'isExpressedBy') relation between 'information objects' and 'social objects' and on the other hand in a 'isAbout' (and its inverse 'isReferenceOf') relation between 'information objects' and 'entities'. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the object properties subClassOf and inverseOf. Moreover the pattern indicates a minimum cardinality.
Graphical	
(UML) Diagram for the General Solution	<<owl::ObjectProperty>> isExpressedBy; <<owl::inverseOf>>; isExpressedBy; «owl::allValuesFrom»; <<rdfs::range>>; InformationObject; SocialObject; 1..*; <<rdfs::range>>; <<owl::ObjectProperty>> expresses; «owl::allValuesFrom»; expresses; <<owl::ObjectProperty>> isReferenceOf; <<owl::inverseOf>>; isReferenceOf; «owl::allValuesFrom»; <<rdfs::range>>; InformationObject; Entity; <<rdfs::range>>; <<owl::ObjectProperty>> isAbout; «owl::allValuesFrom»; isAbout
Formalization	

General	SubClassOf(SocialObject ObjectAllValuesFrom(isExpressedBy InformationObject)) EntityAnnotation(OWLClass(Entity) Label("Entità"@it)) SubClassOf(Entity ObjectAllValuesFrom(isReferenceOf InformationObject)) SubClassOf(InformationObject ObjectAllValuesFrom(isAbout Entity)) SubClassOf(InformationObject ObjectMinCardinality(1 expresses)) SubClassOf(InformationObject SocialObject) SubClassOf(InformationObject ObjectAllValuesFrom(expresses SocialObject)) InverseObjectProperties(isReferenceOf isAbout) ObjectPropertyDomain(isAbout InformationObject) ObjectPropertyRange(isAbout Entity) ObjectPropertyDomain(isReferenceOf Entity) ObjectPropertyRange(isReferenceOf InformationObject) InverseObjectProperties(isReferenceOf isAbout) InverseObjectProperties(expresses isExpressedBy) ObjectPropertyRange(isExpressedBy InformationObject) ObjectPropertyDomain(isExpressedBy SocialObject) ObjectPropertyDomain(expresses InformationObject) ObjectPropertyRange(expresses SocialObject)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP and LP-UR.

Slot	**Value**
General Information	
Name	Information - Realization
Identifier	CP-IR-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that a piece of information is represented by a concrete realization.

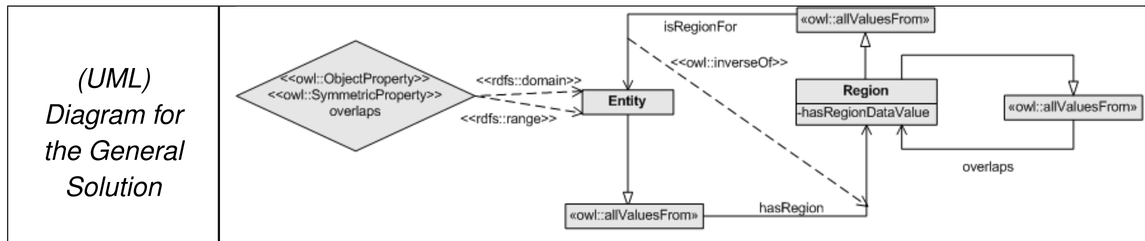
Examples	Suppose that someone wants to express that the paper copy of the Italian Constitution realizes the text of the Constitution.
Ontology Design Pattern	
Informal	
General	The pattern consists in a 'realizes' (and its inverse 'isRealizedBy') relation between 'InformationRealization' and 'InformationObjects'. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the existencial restriction someValuesFrom. Also the object properties inverseOf and disjointWith are instantiated.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(InformationRealization ObjectSomeValuesFrom (realizes InformationObject)) DisjointClasses(InformationRealization InformationObject) SubClassOf(InformationObject ObjectAllValuesFrom(isRealizedBy InformationRealization)) SubClassOf(InformationObject ObjectSomeValuesFrom(isRealizedBy InformationRealization)) ObjectPropertyDomain(realizes InformationRealization) InverseObjectProperties(realizes isRealizedBy) ObjectPropertyRange(realizes InformationObject) ObjectPropertyRange(isRealizedBy InformationRealization) ObjectPropertyDomain(isRealizedBy InformationObject) InverseObjectProperties(realizes isRealizedBy)
Relationships	

2006–2008 © Copyright lies with the respective authors and their institutions.

Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-UR, LP-ER and LP-Di.

A.4 Quantities and Dimensions

Slot	Value
General Information	
Name	Region
Identifier	CP-Re-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents regions that are portions of a dimensional space which can be used as a value for a quality of an Entity.
Examples	Suppose that someone wants to represent that the color of my car is red.
Ontology Design Pattern	
Informal	
General	The pattern consist of the classes 'Entity' and 'Region' and the relations 'hasRegion' and 'isRegionFor' between 'Entity' and 'Region' and 'overlaps' between 'Region' and 'Region'. The pattern should instantiate the classes *Class*, *DatatypeProperty*, *SymmetricObjectProperty* and *ObjectProperty* with the universal restriction *allValuesFrom*. The object property *inverseOf* is also instantiated.
Graphical	

(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(Entity ObjectAllValuesFrom(hasRegion Region)) SubClassOf(Region ObjectAllValuesFrom(overlaps Region)) SubClassOf(Region ObjectAllValuesFrom(isRegionFor Entity)) ObjectPropertyRange(overlaps Entity) SymmetricObjectProperty(overlaps) InverseObjectProperties(overlaps overlaps) ObjectPropertyDomain(overlaps Entity) ObjectPropertyDomain(isRegionFor Region) InverseObjectProperties(isRegionFor hasRegion) ObjectPropertyRange(isRegionFor Entity) ObjectPropertyDomain(hasRegion Entity) ObjectPropertyRange(hasRegion Region) DataPropertyDomain(hasRegionDataValue Region)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-DP and LP-UR.

Slot	**Value**
General Information	
Name	Parameter
Identifier	CP-Pm-01
Type of Component	Content Pattern (CP)
Use Case	

2006–2008 © Copyright lies with the respective authors and their institutions.

General	This pattern represents parameters which are constraints or selections on observable values.
Examples	Suppose that someone wants to represent that the minimum age to drive is 18.
Ontology Design Pattern	
Informal	
General	The pattern consist of the classes ’Parameter’ and ’Concept’ and the relations ’hasParameter’ and ’isParameterFor’ between ’Parameter’ and ’Concept’. The pattern should instantiate the classes *Class*, *DatatypeProperty* and *ObjectProperty* with the universal restriction *allValuesFrom*. The object property i*nverseOf* is also instantiated.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	SubClassOf(Parameter ObjectAllValuesFrom(isParameterFor Concept)) SubClassOf(Concept ObjectAllValuesFrom(hasParameter Parameter)) ObjectPropertyRange(hasParameter Parameter) ObjectPropertyDomain(hasParameter Concept) InverseObjectProperties(hasParameter isParameterFor) ObjectPropertyRange(isParameterFor Concept) ObjectPropertyDomain(isParameterFor Parameter) DataPropertyDomain(hasParameterDataValue Parameter)
Relationships	

Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-DP and LP-UR.

Slot	Value
General Information	
Name	Parameter - Region
Identifier	CP-PmR-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern is the composition of the parameter and region CPs and represents relations between parameters and regions.
Examples	Suppose that someone wants to represent that the age of majority is 18 years old, and that there is the need of reasoning on the dimension “18 years old«« .
Ontology Design Pattern	
Informal	
General	The pattern consist of the classes ’Parameter’ and ’Region’ and the relations ’parametrizes’ and ’isParametrizedBy’ between ’Parameter’ and ’Region’. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom. The object property inverseOf and the dependency import are also instantiated.
Graphical	

2006–2008 © Copyright lies with the respective authors and their institutions.

(UML) Diagram for the General Solution	
Formalization	
General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/parameter.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/region.owl>) SubClassOf(<region.owlchar35Region> ObjectAllValuesFrom(isParametrizedBy <parameter.owlchar35Parameter>)) SubClassOf(<parameter.owlchar35Parameter> ObjectAllValuesFrom(parametrizes <region.owlchar35Region>)) ObjectPropertyRange(parametrizes <region.owlchar35Region>) ObjectPropertyDomain(parametrizes <parameter.owlchar35Parameter>) InverseObjectProperties(isParametrizedBy parametrizes) ObjectPropertyRange(isParametrizedBy <parameter.owlchar35Parameter>) ObjectPropertyDomain(isParametrizedBy <region.owlchar35Region>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-UR and AP-MD.

A.5 Participation

Slot	Value

General Information	
Name	Participation
Identifier	CP-PAR-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that objects take part in events, without temporal indexing.
Examples	Suppose that someone wants to represent that a person takes part in an international conference.
Ontology Design Pattern	
Informal	
General	The pattern consists in a ‘isParticipantIn’ (and its inverse ‘hasParticipant’) relation between ‘objects’ and ‘events’. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the object properties inverseOf and disjointWith.
Graphical	
(UML) Diagram for the General Solution	<<owl::ObjectProperty>> hasParticipant <<owl::inverseOf>> <<owl::someValuesFrom>> <<owl::allValuesFrom>> hasParticipant <<rdfs::range>> Object <<owl::disjointWith>> Event <<rdfs::range>> <<owl::ObjectProperty>> isParticipantIn <<owl::someValuesFrom>> <<owl::allValuesFrom>> isParticipantIn
Formalization	

2006–2008 © Copyright lies with the respective authors and their institutions.

General	SubClassOf(Event ObjectAllValuesFrom(hasParticipant Object)) SubClassOf(Event ObjectSomeValuesFrom(hasParticipant Object)) DisjointClasses(Event Object) SubClassOf(Object ObjectSomeValuesFrom(isParticipantIn Event)) SubClassOf(Object ObjectAllValuesFrom(isParticipantIn Event)) ObjectPropertyDomain(hasParticipant Event) InverseObjectProperties(isParticipantIn hasParticipant) ObjectPropertyRange(hasParticipant Object)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-UR, LP-ER and LP-Di. This pattern is imported in the patterns: CP-NPAR and CP-COP.

Slot	**Value**
General Information	
Name	Coparticipation
Identifier	CP-COP-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that an object can co-participate in an event with other object.
Examples	Suppose that someone wants to represent that two persons take part to the same international conference.
Ontology Design Pattern	
Informal	

General	The pattern consists in a imported class ‘object’ that have a reflexive and symmetric relation ‘coparticipateWith’ (and its inverse ‘isRoleOf’). The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the object property inverseOf. The ontology “Coparticipation” imports the “Participation” ones.
Graphical	
(UML) Diagram for the General Solution	
Formalization	
General	Imports (<http://www.ontologydesignpatterns.org/codeps/owl/participation.owl>) SubClassOf (<participation.owl#Object> ObjectSomeValuesFrom(coparticipatesWith <participation.owl#Object>)) InverseObjectProperties(coparticipatesWith coparticipatesWith) ObjectPropertyRange(coparticipatesWith <participation.owl#Object>) SymmetricObjectProperty(coparticipatesWith) ObjectPropertyDomain(coparticipatesWith <participation.owl#Object>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-ER, CP-PAR and AP-MD.

Slot	**Value**

2006–2008 © Copyright lies with the respective authors and their institutions.

General Information	
Name	N-Ary Participation
Identifier	CP-NPAR-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that participation can be composed of several entities.
Examples	Suppose that someone wants to represent that a two persons take part at the different events at a certain time.
Ontology Design Pattern	
Informal	
General	The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the existencial restriccion someValuesFrom. The object properties inverseOf, disjointWith and subObjectPropertyOf are instantiated. The ontology "NaryParticipation" imports the "Situation" and "Participation" ones.
Graphical	
(UML) Diagram for the General Solution	

Formalization	
General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/situation.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/participation.owl>) SubClassOf(NaryParticipation ObjectSomeValuesFrom(participationIncludes TimeInterval)) SubClassOf(NaryParticipation ObjectSomeValuesFrom(participationIncludes <participation.owl#Event>)) SubClassOf(NaryParticipation ObjectSomeValuesFrom(participationIncludes <participation.owl#Object>)) SubClassOf(NaryParticipation <situation.owl#Situation>) SubClassOf(NaryParticipation ObjectAllValuesFrom(participationIncludes <situation.owl#Entity>)) DisjointClasses(<situation.owl#Situation> TimeInterval) ObjectPropertyDomain(isIncludedInParticipation <situation.owl#Entity>) SubObjectPropertyOf(isIncludedInParticipation <situation.owl#hasSetting>) ObjectPropertyRange(isIncludedInParticipation NaryParticipation) ObjectPropertyRange(participationIncludes <situation.owl#Entity>) ObjectPropertyDomain(participationIncludes NaryParticipation) SubObjectPropertyOf(participationIncludes <situation.owl#isSettingFor>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-UR, LP-ER , LP-Di, CP-SI, CP-PAR and AP-MD.

A.6 Organization, Management, and Scheduling

Slot	**Value**
General Information	
Name	Precedence
Identifier	CP-PRE-01
Type of Component	Content Pattern (CP)
Use Case	

General	Using this pattern we can express a 'sequence' schema.
Examples	Imagine that we want to represent that 'preparing coffee' follows 'deciding what coffee to use'.
Ontology Design Pattern	
Informal	
General	The pattern consists in a class 'entity' that have a transitive relation "follows" (and its inverse 'precedes). The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom. Also the `http://www.w3.org/TR/2004/REC-owl-guide-20040210/` TransitiveProperty and the object property inverseOf are needed.
Graphical	
(UML) Diagram for the General Solution	<<owl::ObjectProperty>> <<owl::TransitiveProperty>> precedes <<owl::inverseOf>> precedes «owl::allValuesFrom» <<rdfs::range>> Entity Entity <<rdfs::range>> <<owl::ObjectProperty>> <<owl::TransitiveProperty>> follows «owl::allValuesFrom» follows
Formalization	
General	SubClassOf(Entity ObjectAllValuesFrom(follows Entity)) SubClassOf(Entity ObjectAllValuesFrom(precedes Entity)) InverseObjectProperties(follows precedes) ObjectPropertyRange(precedes Entity) TransitiveObjectProperty(precedes) ObjectPropertyDomain(precedes Entity) ObjectPropertyRange(follows Entity) TransitiveObjectProperty(follows) ObjectPropertyDomain(follows Entity)
Relationships	

Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP and LP-UR.

Slot	Value
General Information	
Name	Agent-Role
Identifier	CP-AR-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that agents can have different roles.
Examples	Suppose that someone wants to express that 'John' is a person who has the role 'student'.
Ontology Design Pattern	
Informal	
General	The pattern consists in a 'hasRole' (and its inverse 'isRoleOf') relation between 'roles' and 'agents'. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the object property inverseOf.
Graphical	
(UML) Diagram for the General Solution	

Formalization	
General	DisjointClasses(Role Agent) SubClassOf(Role ObjectAllValuesFrom(isRoleOf Agent)) SubClassOf(Agent ObjectAllValuesFrom(hasRole Role)) ObjectPropertyDomain(hasRole Agent) InverseObjectProperties(isRoleOf hasRole) ObjectPropertyRange(hasRole Role) ObjectPropertyDomain(isRoleOf Role) ObjectPropertyRange(isRoleOf Agent) InverseObjectProperties(isRoleOf hasRole)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-UR and LP-Di.

Slot	Value
General Information	
Name	Task Role
Identifier	CP-TR-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that objects which participate in an event can play different roles, and that roles have different task. The tasks are executed in the event.
Examples	Suppose that someone wants to represent that John is a student and students have the duty of giving exams.
Ontology Design Pattern	
Informal	

General	The pattern consists in two imported classes ‘object’ and ‘even’t related to one another by de ‘hasPart’ and ‘isParticipantIn’ relations. Also the pattern relates the classes ‘objec’t with ‘role’, ‘role’ with ‘task’ and ‘task’ with ‘event’ using the relations ‘hasRole’ (‘isRoleOf’), hasTask (‘isTaskOf’) and ‘isExecutedIn’ (‘executes’) respectively. The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom and the existencial restriccion someValuesFrom. Morever the object properties inverseOf and disjointWith are instanciated. The ontology imports the “Participation” ontology.
Graphical	
(UML) Diagram for the General Solution	
Formalization	

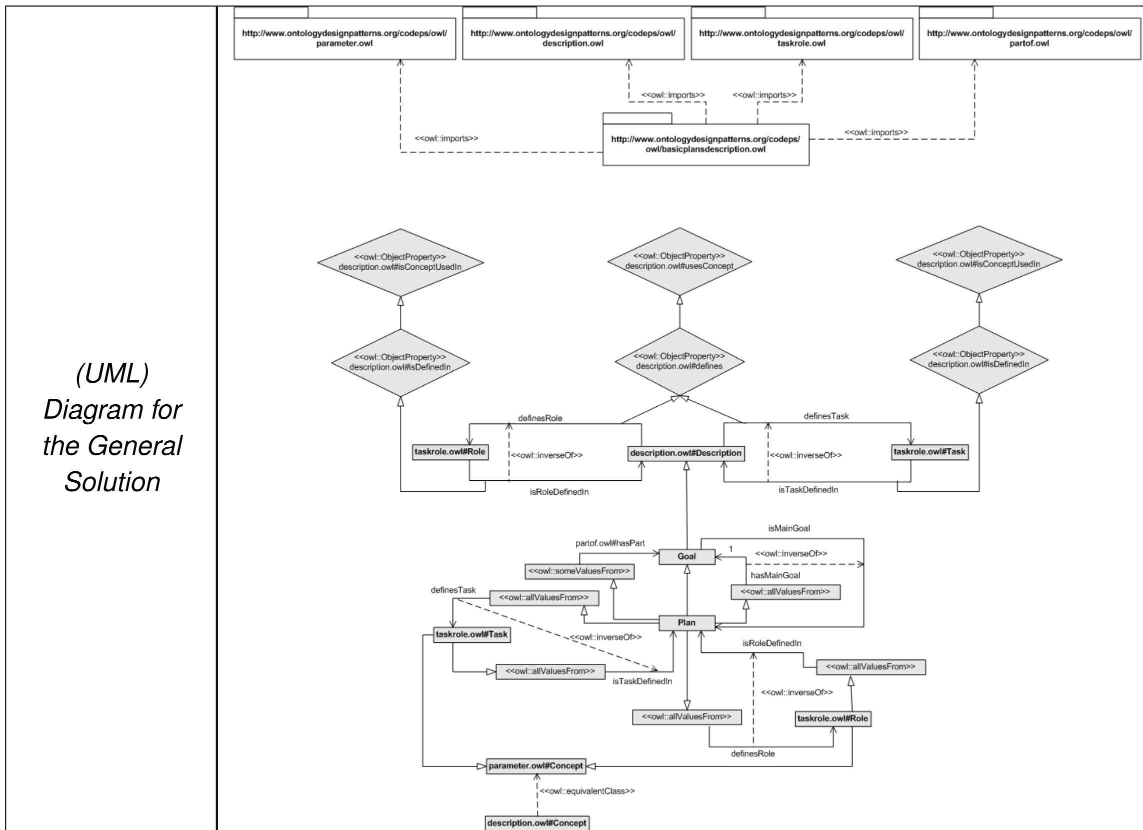
2006–2008 © Copyright lies with the respective authors and their institutions.

General	Imports(<http://www.ontologydesignpatterns.org/codeps/owl/participation.owl>) DisjointClasses(<participation.owl#Event> Role) DisjointClasses(<participation.owl#Event> Task) DisjointClasses(Role Task) SubClassOf(Role ObjectAllValuesFrom(hasTask Task)) SubClassOf(Task ObjectAllValuesFrom(isTaskOf Role)) ObjectPropertyDomain(hasRole <participation.owl#Object>) InverseObjectProperties(hasRole isRoleOf) ObjectPropertyRange(hasRole Role) ObjectPropertyRange(isRoleOf <participation.owl#Object>) ObjectPropertyDomain(isRoleOf Role) ObjectPropertyRange(isExecutedIn <participation.owl#Event>) ObjectPropertyDomain(isExecutedIn Task) InverseObjectProperties(executes isExecutedIn) ObjectPropertyDomain(executes <participation.owl#Event>) ObjectPropertyRange(executes Task) ObjectPropertyDomain(isTaskOf Task) ObjectPropertyRange(isTaskOf Role) InverseObjectProperties(hasTask isTaskOf) ObjectPropertyDomain(hasTask Role) ObjectPropertyRange(hasTask Task)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-ER, LP-UR, LP-Di, CP-PAR and AP-MD.

Slot	**Value**
General Information	
Name	Basic Plan Description
Identifier	CP-BPD-01
Type of Component	Content Pattern (CP)
Use Case	

General	This pattern represents the conceptualization of a plan and the concepts that are included in such a conceptualization.
Examples	Suppose that someone wants to represent that the recipe for a cake defines the task 'boil' and the role 'ingredient'.
Ontology Design Pattern	
Informal	
General	The pattern partially consist of the classes 'Goal' and 'Plan' and the following relations: 'definesTask' between 'Description' and 'Task'; 'hasMainGoal' between 'Plan' and 'Goal'; and 'definesRole' between 'Description' and 'Role'. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the classes *Class* and *ObjectProperty* with the universal restriction *allValuesFrom* and the existential restriction *someValuesFrom*. The object properties *inverseOf*, *subClassOf*, *subPropertyOf* and *equivalentClass* and the relation between ontologies *import* are also instantiated.
Graphical	

2006–2008 © Copyright lies with the respective authors and their institutions.

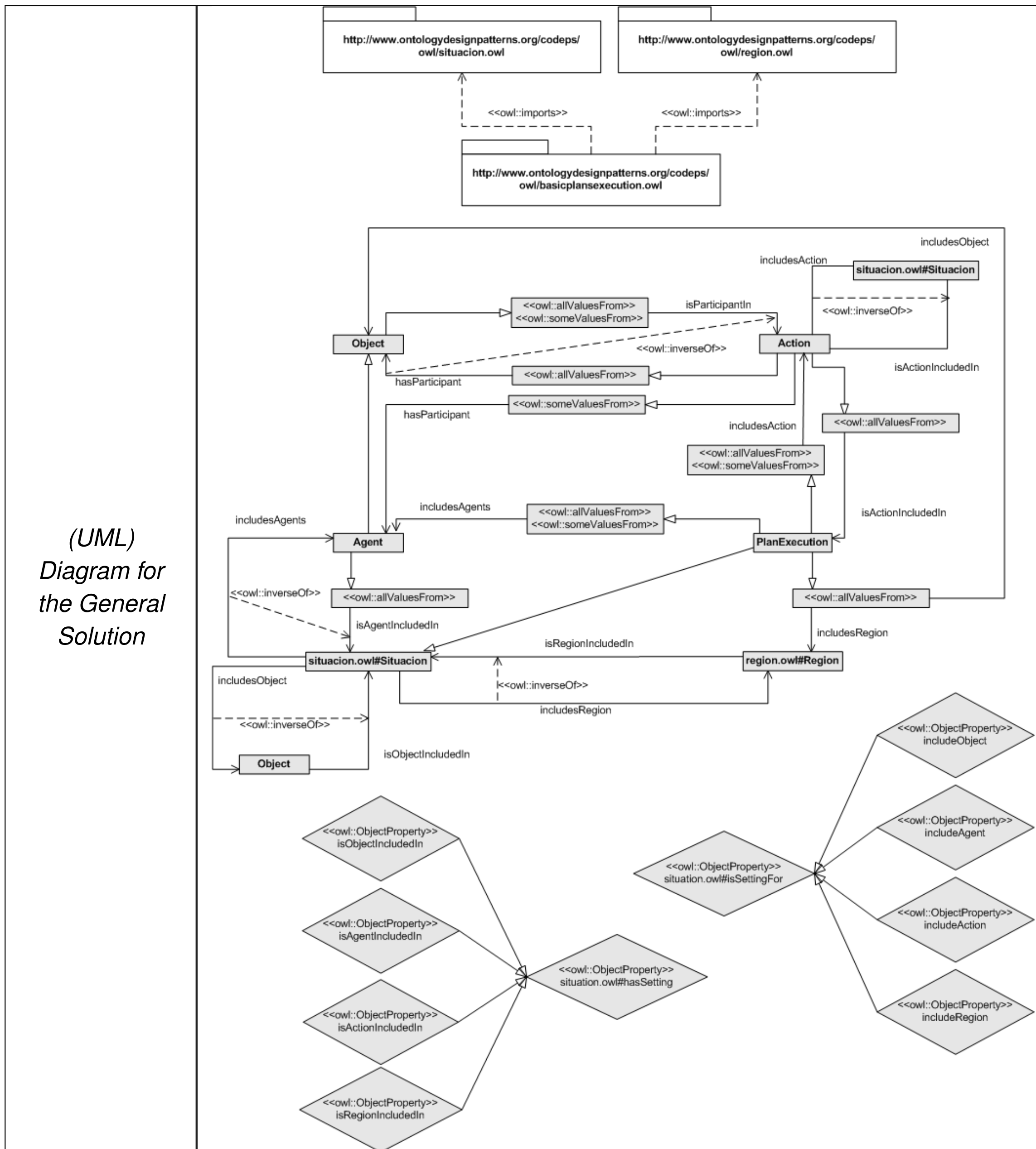
(UML) Diagram for the General Solution	
Formalization	
General	Imports(<http://www.ontologydesignpatterns.org/codeps/owl/ partof.owl>) Imports(<http://www.ontologydesignpatterns.org/codeps/owl/ taskrole.owl>) Imports(<http://www.ontologydesignpatterns.org/codeps/owl/ description.owl>) Imports(<http://www.ontologydesignpatterns.org/codeps/owl/ parameter.owl>) SubClassOf(Plan ObjectMaxCardinality(1 hasMainGoal)) SubClassOf(Plan ObjectAllValuesFrom(definesTask <taskrole.owl#Task>)) SubClassOf(Plan ObjectAllValuesFrom(definesRole <taskrole.owl#Role>)) SubClassOf(Plan ObjectSomeValuesFrom(<partof.owl#hasPart> Goal)) SubClassOf(Plan ObjectAllValuesFrom(hasMainGoal Goal)) SubClassOf(Plan <description.owl#Description>) SubClassOf(Goal <description.owl#Description>)

	EquivalentClasses(<description.owl#Concept> <parameter.owl#Concept>) SubClassOf(<taskrole.owl#Role> <description.owl#Concept>) SubClassOf(<taskrole.owl#Role> ObjectAllValuesFrom(isRoleDefinedIn Plan)) SubClassOf(<taskrole.owl#Task> <description.owl#Concept>) SubClassOf(<taskrole.owl#Task> ObjectAllValuesFrom(isTaskDefinedIn Plan)) InverseObjectProperties(definesTask isTaskDefinedIn) ObjectPropertyDomain(definesTask <description.owl#Description>) SubObjectPropertyOf(definesTask <description.owl#defines>) SubObjectPropertyOf(definesTask <description.owl#usesConcept>) ObjectPropertyRange(definesTask <taskrole.owl#Task>) ObjectPropertyRange(hasMainGoal Goal) InverseObjectProperties(isMainGoalIn hasMainGoal) ObjectPropertyDomain(hasMainGoal Plan) InverseObjectProperties(isRoleDefinedIn definesRole) ObjectPropertyDomain(definesRole <description.owl#Description>) SubObjectPropertyOf(definesRole <description.owl#defines>) ObjectPropertyRange(definesRole <taskrole.owl#Role>) SubObjectPropertyOf(definesRole <description.owl#usesConcept>) SubObjectPropertyOf(isTaskDefinedIn <description.owl#isConceptUsedIn>) ObjectPropertyDomain(isTaskDefinedIn <taskrole.owl#Task>) SubObjectPropertyOf(isTaskDefinedIn <description.owl#isDefinedIn>) ObjectPropertyRange(isTaskDefinedIn <description.owl#Description>) ObjectPropertyRange(isMainGoalIn Plan) ObjectPropertyDomain(isMainGoalIn Goal) ObjectPropertyRange(isRoleDefinedIn <description.owl#Description>) SubObjectPropertyOf(isRoleDefinedIn <description.owl#isDefinedIn>) ObjectPropertyDomain(isRoleDefinedIn <taskrole.owl#Role>) SubObjectPropertyOf(isRoleDefinedIn <description.owl#isConceptUsedIn>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-EQ, LP-SP, LP-UR, LP-ER, AP-MD, CP-Pm, CP-DES, CP-TR and CP-PO.

Slot	**Value**
General Information	
Name	Basic Plan Execution

Identifier	CP-BPE-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents the conceptualization of a plan and the concepts that are included in such a conceptualization.
Examples	Suppose that someone wants to represent something like 'this morning I've prepared my coffee and had my fingers burnt'.
Ontology Design Pattern	
Informal	
General	The pattern partially consist of the classes 'Goal' and 'Plan' and the following relations: 'definesTask' between 'Description' and 'Task'; 'hasMainGoal' between 'Plan' and 'Goal'; and 'definesRole' between 'Description' and 'Role'. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the classes *Class* and *ObjectProperty* with the universal restriction *allValuesFrom* and the existential restriction *someValuesFrom*. The object properties *inverseOf*, *subClassOf*, *subPropertyOf* and *equivalentClass* and the relation between ontologies *import* are also instantiated.
Graphical	

(UML) Diagram for the General Solution



Formalization

General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/region.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/situation.owl>) EquivalentClasses(<region.owl#Entity> <situation.owl#Entity>) SubClassOf(Object ObjectAllValuesFrom(isParticipantIn Action)) SubClassOf(Object ObjectSomeValuesFrom(isParticipantIn Action))

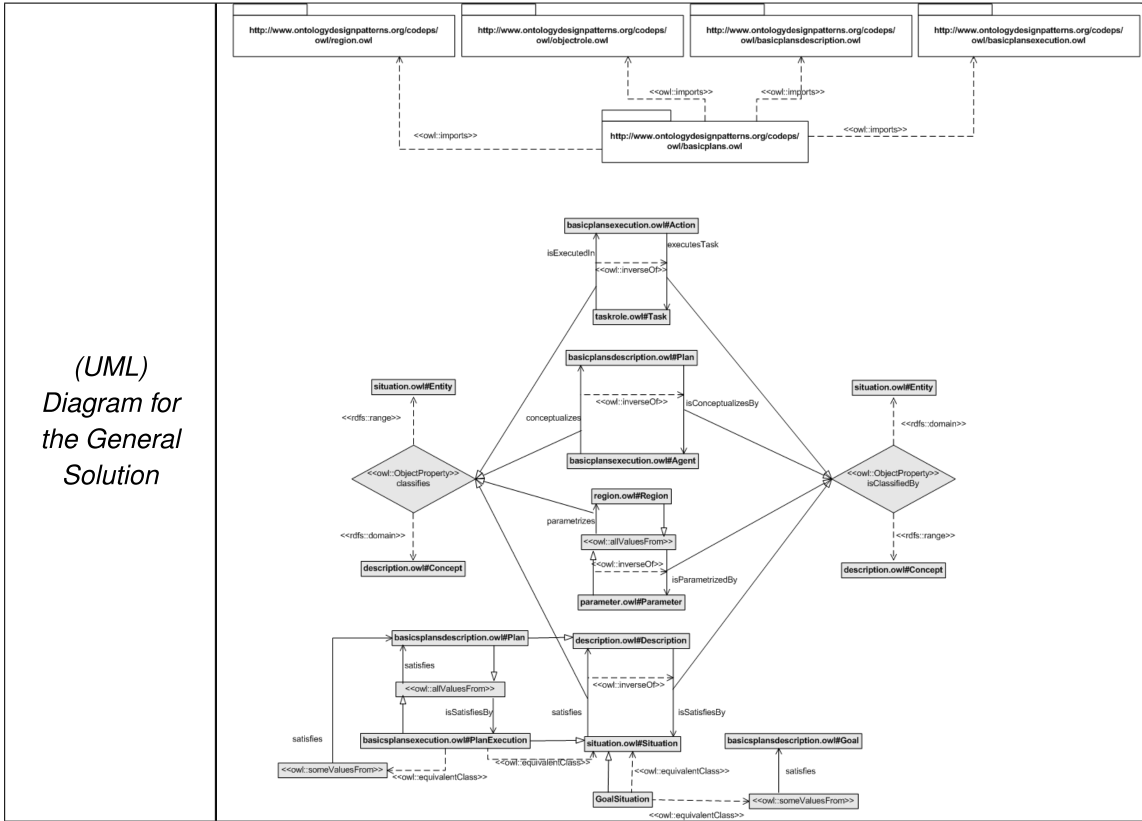
2006–2008 © Copyright lies with the respective authors and their institutions.

		SubClassOf(Action ObjectSomeValuesFrom(hasParticipant Agent)) SubClassOf(Action ObjectAllValuesFrom(hasParticipant Object)) SubClassOf(Action ObjectAllValuesFrom(isActionIncludedIn PlanExecution)) SubClassOf(PlanExecution <situation.owl#Situation>) SubClassOf(PlanExecution ObjectAllValuesFrom(includesRegion <region.owl#Region>)) SubClassOf(PlanExecution ObjectAllValuesFrom(includesObject Object)) SubClassOf(PlanExecution ObjectAllValuesFrom(includesAction Action)) SubClassOf(PlanExecution ObjectSomeValuesFrom(includesAgent Agent)) SubClassOf(PlanExecution ObjectAllValuesFrom(includesAgent Agent)) SubClassOf(PlanExecution ObjectSomeValuesFrom(includesAction Action)) SubClassOf(Agent ObjectAllValuesFrom(isAgentIncludedIn <situation.owl#Situation>)) SubClassOf(Agent Object) SubObjectPropertyOf(includesRegion <situation.owl#isSettingFor>) ObjectPropertyDomain(includesRegion <situation.owl#Situation>) ObjectPropertyRange(includesRegion <region.owl#Region>) InverseObjectProperties(includesRegion isRegionIncludedIn) ObjectPropertyRange(hasParticipant Object) ObjectPropertyDomain(hasParticipant Action) InverseObjectProperties(hasParticipant isParticipantIn) ObjectPropertyRange(isAgentIncludedIn <situation.owl#Situation>) ObjectPropertyDomain(isAgentIncludedIn Agent) SubObjectPropertyOf(isAgentIncludedIn <situation.owl#hasSetting>) InverseObjectProperties(includesAgent isAgentIncludedIn) ObjectPropertyRange(isRegionIncludedIn <situation.owl#Situation>) ObjectPropertyDomain(isRegionIncludedIn <region.owl#Region>) SubObjectPropertyOf(isRegionIncludedIn <situation.owl#hasSetting>) ObjectPropertyRange(includesAction Action) InverseObjectProperties(isActionIncludedIn includesAction) ObjectPropertyDomain(includesAction <situation.owl#Situation>) SubObjectPropertyOf(includesAction <situation.owl#isSettingFor>) ObjectPropertyDomain(includesAgent <situation.owl#Situation>) ObjectPropertyRange(includesAgent Agent) SubObjectPropertyOf(includesAgent <situation.owl#isSettingFor>) ObjectPropertyRange(includesObject Object) SubObjectPropertyOf(includesObject <situation.owl#isSettingFor>) ObjectPropertyDomain(includesObject <situation.owl#Situation>) InverseObjectProperties(isObjectIncludedIn includesObject)

	ObjectPropertyDomain(isActionIncludedIn Action) SubObjectPropertyOf(isActionIncludedIn <situation.owl#hasSetting>) ObjectPropertyRange(isActionIncludedIn <situation.owl#Situation>) ObjectPropertyDomain(isObjectIncludedIn Object) ObjectPropertyRange(isObjectIncludedIn <situation.owl#Situation>) SubObjectPropertyOf(isObjectIncludedIn <situation.owl#hasSetting>) ObjectPropertyRange(isParticipantIn Action) ObjectPropertyDomain(isParticipantIn Object)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-EQ, LP-SP, LP-UR, LP-ER, AP-MD, CP-Pm, CP-DES, CP-TR and CP-PO.

Slot	**Value**
General Information	
Name	Basic - Plan
Identifier	CP-BP-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents plans descriptions and their executions.
Examples	Suppose that someone wants to represent the description of a plan for quality assurance of a software product and a its specific execution applied to a certain software product.
Ontology Design Pattern	
Informal	

2006–2008 © Copyright lies with the respective authors and their institutions.

General	The pattern partially consist of the class 'GoalSituation' and the following relations: 'conceptualizes' between 'Agent' and 'Plan'; 'satisfies' between 'Situation' and 'Description'; 'parametrizes' between 'Parameter' and 'Region'; 'executesTask' between 'Action' and 'Task'; and 'classifies' between 'Concept' and 'Entity'. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the *classes* Class and *ObjectProperty* with the universal restriction *allValuesFrom* and the existential restriction *someValuesFrom*. The object properties *inverseOf*, *subClassOf* and *equivalentClass* and the relation between ontologies *import* are also instantiated.
Graphical	
(UML) Diagram for the General Solution	
Formalization	

General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/basicplansdescription.owl>) Imports(<http://www.loa-cnr.it/ontologies/PlansLite.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/objectrole.owl>) Imports(<http://www.ontologydesignpatterns.org/ odeps/owl/basicplansexecution.owl>) Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/region.owl>) DisjointClasses(<parameter.owl#Parameter> <taskrole.owl#Role>) DisjointClasses(<description.owl#Concept> <situation.owl#Situation>) DisjointClasses(<description.owl#Concept> <basicplansexecution.owl#Agent>) DisjointClasses(<basicplansexecution.owl#Object> <region.owl#Region>) DisjointClasses(<basicplansexecution.owl#Object> <basicplansexecution.owl#Action>) EquivalentClasses(<objectrole.owl#Role> <taskrole.owl#Role>)

2006–2008 ⓒ Copyright lies with the respective authors and their institutions.

EquivalentClasses(<situation.owl#Entity> <partof.owl#Entity>)
EquivalentClasses(<basicplansexecution.owl#Object>
<objectrole.owl#Object>)
EquivalentClasses(GoalSituation
ObjectIntersectionOf(ObjectSomeValuesFrom(satisfies
<basicplansdescription.owl#Goal>) <situation.owl#Situation>))
EquivalentClasses(ObjectIntersectionOf(<situation.owl#Situation>
ObjectSomeValuesFrom(satisfies <basicplansdescription.owl#Plan>))
<basicplansexecution.owl#PlanExecution>)
EquivalentClasses(<situation.owl#Entity> <partof.owl#Entity>)
EquivalentClasses(<basicplansexecution.owl#Object>
<objectrole.owl#Object>)
SubClassOf(<basicplansdescription.owl#Plan>
ObjectAllValuesFrom(isSatisfiedBy
<basicplansexecution.owl#PlanExecution>))
SubClassOf(<parameter.owl#Parameter>
ObjectAllValuesFrom(parametrizes <region.owl#Region>))
SubClassOf(<description.owl#Concept> ObjectAllValuesFrom(classifies
<situation.owl#Entity>))
SubClassOf(<taskrole.owl#Role>
ObjectAllValuesFrom(<objectrole.owl#isRoleOf> <objectrole.owl#Object>))
SubClassOf(<region.owl#Region> ObjectAllValuesFrom(isParametrizedBy
<parameter.owl#Parameter>))
SubClassOf(<basicplansexecution.owl#Object>
ObjectAllValuesFrom(<objectrole.owl#hasRole> <taskrole.owl#Role>))
SubClassOf(<basicplansexecution.owl#Object>
ObjectAllValuesFrom(isClassifiedBy <taskrole.owl#Role>))
SubClassOf(GoalSituation <situation.owl#Situation>)
SubClassOf(<basicplansexecution.owl#PlanExecution>
ObjectAllValuesFrom(satisfies <basicplansdescription.owl#Plan>))
SubClassOf(<taskrole.owl#Task> ObjectAllValuesFrom(isExecutedIn
<basicplansexecution.owl#Action>))
InverseObjectProperties(classifies isClassifiedBy)
ObjectPropertyDomain(isClassifiedBy <situation.owl#Entity>)
ObjectPropertyRange(isClassifiedBy <description.owl#Concept>)
SubObjectPropertyOf(<objectrole.owl#isRoleOf> classifies)
ObjectPropertyDomain(conceptualizes <basicplansexecution.owl#Agent>)
ObjectPropertyRange(conceptualizes <basicplansdescription.owl#Plan>)
InverseObjectProperties(conceptualizes isConceptualizedBy)
InverseObjectProperties(isParametrizedBy parametrizes)
ObjectPropertyDomain(isParametrizedBy <region.owl#Region>)

	ObjectPropertyRange(isParametrizedBy <parameter.owl#Parameter>) SubObjectPropertyOf(isParametrizedBy isClassifiedBy) SubObjectPropertyOf(isExecutedIn classifies) InverseObjectProperties(isExecutedIn executesTask) ObjectPropertyRange(isExecutedIn <basicplansexecution.owl#Action>) ObjectPropertyDomain(isExecutedIn <taskrole.owl#Task>) ObjectPropertyRange(satisfies <description.owl#Description>) ObjectPropertyDomain(satisfies <situation.owl#Situation>) InverseObjectProperties(satisfies isSatisfiedBy) ObjectPropertyRange(parametrizes <region.owl#Region>) ObjectPropertyDomain(parametrizes <parameter.owl#Parameter>) SubObjectPropertyOf(parametrizes classifies) ObjectPropertyRange(isSatisfiedBy <situation.owl#Situation>) ObjectPropertyDomain(isSatisfiedBy <description.owl#Description>) SubObjectPropertyOf(executesTask isClassifiedBy) ObjectPropertyDomain(executesTask <basicplansexecution.owl#Action>) ObjectPropertyRange(executesTask <taskrole.owl#Task>) ObjectPropertyRange(classifies <situation.owl#Entity>) ObjectPropertyDomain(classifies <description.owl#Concept>) SubObjectPropertyOf(<objectrole.owl#hasRole> isClassifiedBy) ObjectPropertyRange(isConceptualizedBy <basicplansexecution.owl#Agent>) ObjectPropertyDomain(isConceptualizedBy <basicplansdescription.owl#Plan>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-EQ, LP-SP, LP-UR, LP-ER, AP-MD, CP-Re, CP-OR, CP-BPD and CP-BPE.

A.7 Business

Slot	Value
General Information	
Name	Price
Identifier	CP-Pr-01
Type of Component	Content Pattern (CP)

2006–2008 © Copyright lies with the respective authors and their institutions.

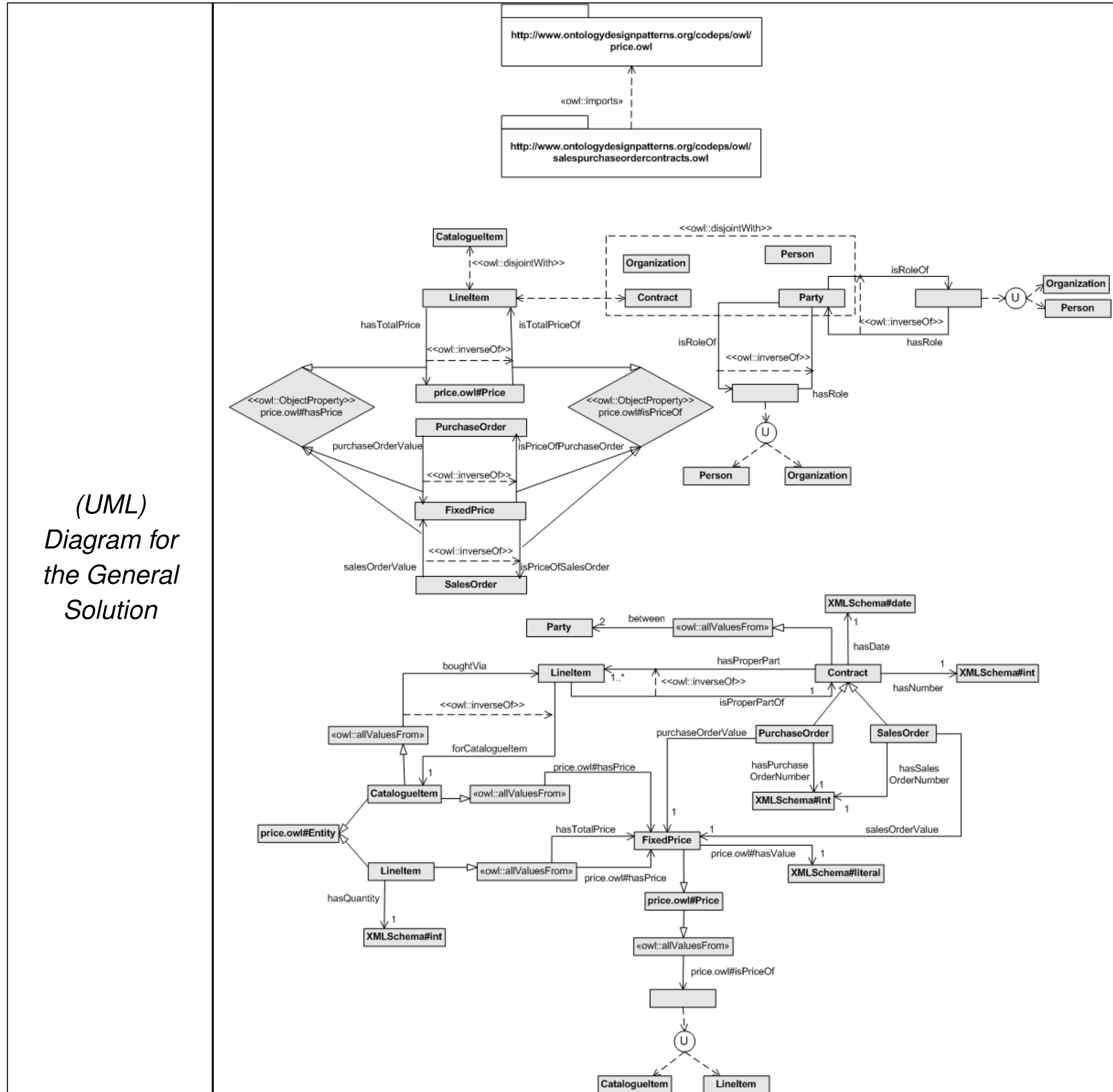
Use Case	
General	This pattern represents the price of an entity in a certain currency.
Examples	Suppose that someone wants to represent that the price of a coffee is one euro.
Ontology Design Pattern	
Informal	
General	The pattern consist of the classes ’Entity’, ’Price’ and ’Currency’ and the relations ’hasPrice’ and ’isPriceOf’ between ’Entity’ and ’Price’ and ’hasCurrency’ between ’Price’ and ’Currency’. The pattern should instantiate the classes *Class*, *DatatypeProperty* and *ObjectProperty*. The object properties *inverseOf* and *disjointWith* are also instantiated.
Graphical	
(UML) Diagram for the General Solution	isPriceOf <<owl::inverseOf>> hasPrice hasCurrency 1 Entity <<owl::disjointWith>> Price -hasValue: float <<owl::disjointWith>> Currency <<owl::disjointWith>>
Formalization	

General	DisjointClasses(Currency Price) DisjointClasses(Price Entity) DisjointClasses(Currency Entity) SubClassOf(Price DataMinCardinality(1 hasValue)) SubClassOf(Price ObjectExactCardinality(1 hasCurrency)) ObjectPropertyRange(isPriceOf Entity) ObjectPropertyDomain(isPriceOf Price) InverseObjectProperties(isPriceOf hasPrice) ObjectPropertyRange(hasCurrency Currency) ObjectPropertyDomain(hasCurrency Price) ObjectPropertyRange(hasPrice Price) ObjectPropertyDomain(hasPrice Entity) DataPropertyDomain(DataProperty(hasValue) OWLClass(Price)) DataPropertyRange(DataProperty(hasValue) xsd:float)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-DP and LP-Di.

Slot	Value
General Information	
Name	Sales Purchase Order Contracts
Identifier	CP-SPOC-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern describes the purchase orders and sales orders by which an organization buys and sells goods and services, and it extends these to encompass the more general concept of contract or agreement.
Examples	Suppose that someone wants to represent that he bought a stock of wine at a certain price.

2006–2008 © Copyright lies with the respective authors and their institutions.

Ontology Design Pattern	
Informal	
General	The pattern partially consist of the classes 'CatalogueItem', 'PurchaseOrder', 'Contract', 'LineItem', 'Organization', 'FixedPrice', 'Person', 'Party', and 'SalesOrder' and the following relations: 'hasProperPart' between 'Contract' and 'LineItem'; 'hasTotalPrice' between 'LineItem' and 'Price'; 'boughtVia' between 'LineItem' and 'CatalogueItem'; 'salesOrderValue' between 'FixedPrice' and 'SalesOrder'; 'hasRole' whose range is 'Party' and its domain is de union of 'Person' and 'Organization'; 'purchaseOrderValue' between 'FixedPrice' and 'PurchaseOrder'; and 'between' between 'Contract' and 'Party'. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the classes *Class*, *DatatypeProperty* and *ObjectProperty* with the universal restriction *allValuesFrom*. The object properties *inverseOf*, *unionOf*, *subClassOf*, *subPropertyOf* and *disjointWith* and the relation between ontologies *import* are also

	instantiated.
	Graphical
(UML) Diagram for the General Solution	
	Formalization
General	Imports(<http://www.ontologydesignpatterns.org/ codeps/owl/price.owl>) DisjointClasses(Person Organization) DisjointClasses(Person Party) DisjointClasses(Contract Person) DisjointClasses(Contract Organization) DisjointClasses(Contract Party)

DisjointClasses(Contract LineItem)
DisjointClasses(Party Organization)
DisjointClasses(CatalogueItem LineItem)
SubClassOf(Contract ObjectAllValuesFrom(between Party))
SubClassOf(Contract ObjectExactCardinality(2 between))
SubClassOf(Contract DataExactCardinality(1 hasNumber))
SubClassOf(Contract ObjectMinCardinality(1 hasProperPart))
SubClassOf(Contract DataExactCardinality(1 hasDate))
SubClassOf(Party owl:Thing)
SubClassOf(FixedPrice DataExactCardinality(1 <price.owl#hasValue>))
SubClassOf(FixedPrice <price.owl#Price>)
SubClassOf(SalesOrder Contract)
SubClassOf(SalesOrder ObjectExactCardinality(1 salesOrderValue))
SubClassOf(SalesOrder DataExactCardinality(1 hasDate))
SubClassOf(SalesOrder DataExactCardinality(1 hasSalesOrderNumber))
SubClassOf(<price.owl#Price> ObjectAllValuesFrom(<price.owl#isPriceOf>
ObjectUnionOf(CatalogueItem
SubClassOf(CatalogueItem <price.owl#Entity>)
SubClassOf(CatalogueItem ObjectAllValuesFrom(<price.owl#hasPrice>
FixedPrice))
SubClassOf(CatalogueItem ObjectAllValuesFrom(boughtVia LineItem))
SubClassOf(PurchaseOrder Contract)
SubClassOf(PurchaseOrder DataExactCardinality(1
hasPurchaseOrderNumber))
SubClassOf(PurchaseOrder ObjectExactCardinality(1
purchaseOrderValue))
SubClassOf(PurchaseOrder DataExactCardinality(1 hasDate))
SubClassOf(LineItem <price.owl#Entity>)
SubClassOf(LineItem ObjectAllValuesFrom(hasTotalPrice FixedPrice))
SubClassOf(LineItem DataMaxCardinality(1 hasQuantity))
SubClassOf(LineItem ObjectAllValuesFrom(<price.owl#hasPrice>
FixedPrice))
SubClassOf(LineItem ObjectExactCardinality(1 isProperPartOf))
SubClassOf(LineItem ObjectExactCardinality(1 forCatalogueItem))
SubObjectPropertyOf(isTotalPriceOf <price.owl#isPriceOf>)
ObjectPropertyRange(isTotalPriceOf LineItem)
InverseObjectProperties(isTotalPriceOf hasTotalPrice)
ObjectPropertyDomain(isTotalPriceOf <price.owl#Price>)
InverseObjectProperties(<price.owl#hasPrice> <price.owl#isPriceOf>)
ObjectPropertyDomain(forCatalogueItem LineItem)
ObjectPropertyRange(forCatalogueItem CatalogueItem)

InverseObjectProperties(boughtVia forCatalogueItem)
ObjectPropertyDomain(hasRole ObjectUnionOf(PersonOrganization))
InverseObjectProperties(isRoleOf hasRole)
ObjectPropertyRange(hasRole Party)
InverseObjectProperties(hasProperPart isProperPartOf)
ObjectPropertyRange(isProperPartOf Contract)
ObjectPropertyDomain(isProperPartOf LineItem)
ObjectPropertyRange(between Party)
ObjectPropertyDomain(between Contract)
InverseObjectProperties(isPriceOfSalesOrder salesOrderValue)
SubObjectPropertyOf(isPriceOfSalesOrder <price.owl#isPriceOf>)
ObjectPropertyRange(isPriceOfSalesOrder SalesOrder)
ObjectPropertyDomain(isPriceOfSalesOrder FixedPrice)
ObjectPropertyDomain(isRoleOf Party)
ObjectPropertyRange(isRoleOf ObjectUnionOf(Person Organization))
ObjectPropertyRange(boughtVia LineItem)
ObjectPropertyDomain(boughtVia CatalogueItem)
ObjectPropertyDomain(hasTotalPrice LineItem)
SubObjectPropertyOf(hasTotalPrice <price.owl#hasPrice>)
ObjectPropertyRange(hasTotalPrice <price.owl#Price>)
ObjectPropertyRange(purchaseOrderValue FixedPrice)
ObjectPropertyDomain(purchaseOrderValue PurchaseOrder)
InverseObjectProperties(isPriceOfPurchaseOrder purchaseOrderValue)
SubObjectPropertyOf(purchaseOrderValue <price.owl#hasPrice>)
ObjectPropertyRange(salesOrderValue FixedPrice)
SubObjectPropertyOf(salesOrderValue <price.owl#hasPrice>)
ObjectPropertyDomain(salesOrderValue SalesOrder)
ObjectPropertyRange(isPriceOfPurchaseOrder PurchaseOrder)
SubObjectPropertyOf(isPriceOfPurchaseOrder <price.owl#isPriceOf>)
ObjectPropertyDomain(isPriceOfPurchaseOrder FixedPrice)
ObjectPropertyDomain(hasProperPart Contract)
ObjectPropertyRange(hasProperPart LineItem)
Declaration(hasPurchaseOrderNumber)
DataPropertyRange(DataProperty(hasPurchaseOrderNumber) xsd:int)
SubDataPropertyOf(DataProperty(hasPurchaseOrderNumber)
DataProperty(hasNumber))
DataPropertyDomain(DataProperty(hasPurchaseOrderNumber)
OWLClass(PurchaseOrder))
Declaration(hasSalesOrderNumber)
DataPropertyDomain(DataProperty(hasSalesOrderNumber)
OWLClass(SalesOrder))
DataPropertyRange(DataProperty(hasSalesOrderNumber) xsd:int)
SubDataPropertyOf(DataProperty(hasSalesOrderNumber)
DataProperty(hasNumber))
Declaration(hasDate)

2006–2008 © Copyright lies with the respective authors and their institutions.

	DataPropertyDomain(DataProperty(hasDate) OWLClass(Contract)) DataPropertyRange(DataProperty(hasDate) xsd:date) Declaration(hasNumber) DataPropertyDomain(DataProperty(hasNumber) OWLClass(Contract)) DataPropertyRange(DataProperty(hasNumber) xsd:int) Declaration(hasQuantity) DataPropertyRange(DataProperty(hasQuantity) xsd:int) DataPropertyDomain(DataProperty(hasQuantity) OWLClass(LineItem))
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-DP, LP-SP, LP-UR, LP-Di, LP-UO, AP-MD, and CP-Pr.

A.8 Time

Slot	Value
General Information	
Name	Time - Interval
Identifier	CP-TI-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents the a time interval and associate a date with it. I also allows to express a starting and an ending date for the interval.
Examples	Suppose that someone wants to represent that the time period for registration in a congress starts on May 05, 2008 and ends on May 27, 2008.
Ontology Design Pattern	

Informal	
General	The pattern consist of the classes 'TimeInterval' and the relations 'hasIntervalStartDate' and 'hasIntervalEndDate' between 'TimeInterval' and 'xsd:date'. The pattern should instantiate the classes *Class* and *DatatypeProperty*. The object property *subPropertyOf* is also instantiated.
Graphical	
(UML) Diagram for the General Solution	hasIntervalStartDate TimeInterval <<rdfs::domain>> <<owl::DataTypeProperty>> hasIntervalDate <<rdfs::range>> XMLSchema#date hasIntervalEndDate

2006–2008 ⓒ Copyright lies with the respective authors and their institutions.

Formalization	
General	SubClassOf(TimeInterval DataMaxCardinality(1 hasIntervalEndDate)) SubClassOf(TimeInterval DataMaxCardinality(1 hasIntervalStartDate)) DataPropertyRange(hasIntervalStartDate xsd:date) DataPropertyDomain(hasIntervalStartDate TimeInterval) SubDataPropertyOf(hasIntervalStartDate hasIntervalDate) DataPropertyDomain(hasIntervalDate TimeInterval) DataPropertyRange(hasIntervalDate xsd:date) DataPropertyRange(hasIntervalEndDate xsd:date) DataPropertyDomain(hasIntervalEndDate TimeInterval) SubDataPropertyOf(hasIntervalEndDate hasIntervalDate)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SP and LP-DP.

A.9 Space

Slot	Value
General Information	
Name	Move
Identifier	CP-Mo-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents the action of moving a physical object from a place to another.
Examples	Suppose that someone wants to represent that Monet's painting "Impression sunrise" was moved for the first Impressionist exhibition.

Ontology Design Pattern	
Informal	
General	The pattern partially consist of the classes 'Move', 'PhisicalObject' and 'Place' and the following relations: 'moved' between 'Move' and 'PhisicalObject'; 'movedFrom' between 'Place' and 'Move'; 'witnessed' between 'Place' and 'Move'; 'movedTo' between 'Place' and 'Move'; and 'consistsOf' between 'Move' and 'Move'. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the classes *Class* and *ObjectProperty*. The object properties *disjointWith*, *inverseOf* and *subPropertyOf* are also instantiated.
Graphical	
(UML) Diagram for the General Solution	
Formalization	

2006–2008 © Copyright lies with the respective authors and their institutions.

General	DisjointClasses(Move Place) DisjointClasses(Move PhysicalObject) DisjointClasses(PhysicalObject Place) SubObjectPropertyOf(movedFrom tookPlaceAt) ObjectPropertyRange(movedFrom Place) ObjectPropertyDomain(movedFrom Move) InverseObjectProperties(wasOriginOf movedFrom) ObjectPropertyRange(tookPlaceAt Place) ObjectPropertyDomain(tookPlaceAt Move) InverseObjectProperties(tookPlaceAt witnessed) ObjectPropertyDomain(movedBy PhysicalObject) InverseObjectProperties(moved movedBy) ObjectPropertyRange(movedBy Move) ObjectPropertyDomain(consistsOf Move) ObjectPropertyRange(consistsOf Move) InverseObjectProperties(consistsOf formsPartOf) ObjectPropertyRange(formsPartOf Move) ObjectPropertyDomain(formsPartOf Move) ObjectPropertyDomain(moved Move) ObjectPropertyRange(moved PhysicalObject) SubObjectPropertyOf(wasDestinationOf witnessed) ObjectPropertyRange(wasDestinationOf Move) ObjectPropertyDomain(wasDestinationOf Place) InverseObjectProperties(wasDestinationOf movedTo) ObjectPropertyRange(witnessed Move) ObjectPropertyDomain(witnessed Place) ObjectPropertyDomain(movedTo Move)

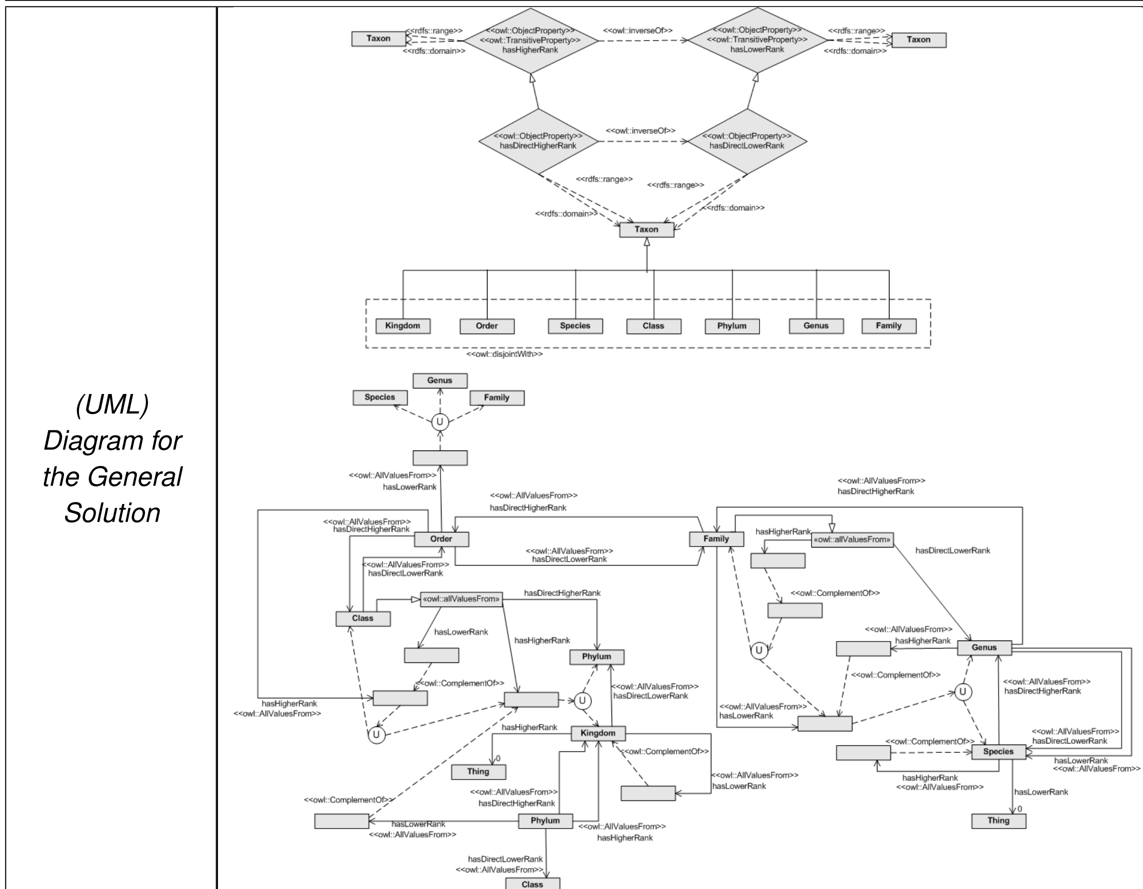
	ObjectPropertyRange(movedTo Place) SubObjectPropertyOf(movedTo tookPlaceAt) ObjectPropertyDomain(wasOriginOf Place) SubObjectPropertyOf(wasOriginOf witnessed) ObjectPropertyRange(wasOriginOf Move)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-OP, LP-SP and LP-Di.

A.10 Life Science

Slot	Value
General Information	
Name	Linnaean Taxonomy
Identifier	CP-LT-01
Type of Component	Content Pattern (CP)
Use Case	
General	This pattern represents the layered classification invented by Linneus in the XVIII century.
Examples	Suppose that someone wants to represent that the human's kingdom is 'Animalia'
Ontology Design Pattern	
Informal	

2006–2008 © Copyright lies with the respective authors and their institutions.

<table>
<tr><td>General</td><td>The pattern consist of the classes 'Order', 'Kingdom', 'Class', 'Taxon', 'Species', 'Family', 'Phylum' and 'Genus' and the following relations: 'hasDirectHigherRank', 'hasDirectLowerRank', 'hasHigherRank' and 'hasLowerRank' between 'Taxon' and 'Taxon'.

The pattern should instantiate the classes Class and ObjectProperty with the universal restriction allValuesFrom. The object properties inverseOf, subClassOf, unionOf, complementOf, disjointWith and subObjectProperty are also instantiated.</td></tr>
<tr><td colspan="2">Graphical</td></tr>
<tr><td>(UML) Diagram for the General Solution</td><td>


</td></tr>
<tr><td colspan="2">Formalization</td></tr>
</table>

General	DisjointClasses(Class Phylum) DisjointClasses(Class Kingdom) DisjointClasses(Class Family) DisjointClasses(Class Genus) DisjointClasses(Class Species) DisjointClasses(Class Order) DisjointClasses(Kingdom Phylum) DisjointClasses(Kingdom Order) DisjointClasses(Kingdom Family) DisjointClasses(Kingdom Species) DisjointClasses(Kingdom Genus) DisjointClasses(Order Species) DisjointClasses(Order Phylum) DisjointClasses(Order Family) DisjointClasses(Order Genus) DisjointClasses(Species Family) DisjointClasses(Species Phylum)

2006–2008 © Copyright lies with the respective authors and their institutions.

	DisjointClasses(Species Genus) DisjointClasses(Species Phylum) DisjointClasses(Phylum Family) DisjointClasses(Phylum Genus) DisjointClasses(Genus Family) SubClassOf(Class ObjectAllValuesFrom(hasDirectLowerRank Order)) SubClassOf(Class ObjectAllValuesFrom(hasHigherRank ObjectUnionOf(Kingdom Phylum))) SubClassOf(Class ObjectAllValuesFrom(hasDirectHigherRank Phylum)) SubClassOf(Class Taxon) SubClassOf(Class ObjectAllValuesFrom(hasLowerRank ObjectComplementOf(ObjectUnionOf(Class Kingdom Phylum)))) SubClassOf(Kingdom ObjectAllValuesFrom(hasDirectLowerRank Phylum)) SubClassOf(Kingdom ObjectAllValuesFrom(hasLowerRank ObjectComplementOf(Kingdom))) SubClassOf(Kingdom Taxon) SubClassOf(Kingdom ObjectExactCardinality(0 hasHigherRank)) SubClassOf(Order ObjectAllValuesFrom(hasDirectHigherRank Class)) SubClassOf(Order ObjectAllValuesFrom(hasHigherRank ObjectUnionOf(Class Kingdom Phylum))) SubClassOf(Order ObjectAllValuesFrom(hasLowerRank ObjectUnionOf(Species Genus Family))) SubClassOf(Order Taxon) SubClassOf(Order ObjectAllValuesFrom(hasDirectLowerRank Family)) SubClassOf(Species ObjectAllValuesFrom(hasHigherRank ObjectComplementOf(Species))) SubClassOf(Species ObjectAllValuesFrom(hasDirectHigherRank Genus)) SubClassOf(Species ObjectExactCardinality(0 hasLowerRank)) SubClassOf(Species Taxon) SubClassOf(Phylum ObjectAllValuesFrom(hasLowerRank ObjectComplementOf(ObjectUnionOf(Kingdom Phylum)))) SubClassOf(Phylum Taxon) SubClassOf(Phylum ObjectAllValuesFrom(hasHigherRank Kingdom)) SubClassOf(Phylum ObjectAllValuesFrom(hasDirectLowerRank Class)) SubClassOf(Phylum ObjectAllValuesFrom(hasDirectHigherRank Kingdom)) SubClassOf(Genus Taxon) SubClassOf(Genus ObjectAllValuesFrom(hasLowerRank Species)) SubClassOf(Genus ObjectAllValuesFrom(hasHigherRank ObjectComplementOf(ObjectUnionOf(Species Genus)))) SubClassOf(Genus ObjectAllValuesFrom(hasDirectHigherRank Family))

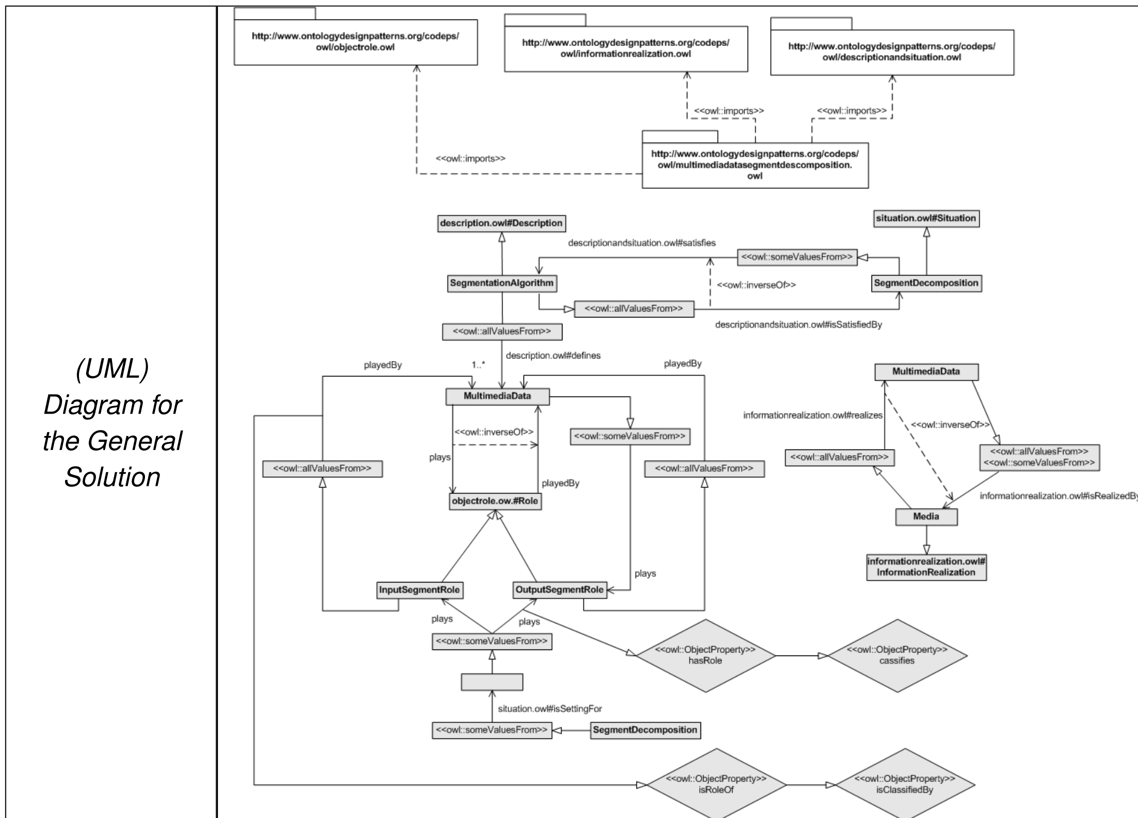
	SubClassOf(Genus ObjectAllValuesFrom(hasDirectLowerRank Species)) SubClassOf(Family ObjectAllValuesFrom(hasLowerRank ObjectUnionOf(Species Genus))) SubClassOf(Family ObjectAllValuesFrom(hasHigherRank ObjectComplementOf(ObjectUnionOf(Species Genus Family)))) SubClassOf(Family Taxon) SubClassOf(Family ObjectAllValuesFrom(hasDirectLowerRank Genus)) SubClassOf(Family ObjectAllValuesFrom(hasDirectHigherRank Order)) TransitiveObjectProperty(hasHigherRank) ObjectPropertyRange(hasHigherRank Taxon) InverseObjectProperties(hasLowerRank hasHigherRank) ObjectPropertyDomain(hasHigherRank Taxon) ObjectPropertyDomain(hasDirectLowerRank Taxon) ObjectPropertyRange(hasDirectLowerRank Taxon) SubObjectPropertyOf(hasDirectLowerRank hasLowerRank) InverseObjectProperties(hasDirectHigherRank hasDirectLowerRank) SubObjectPropertyOf(hasDirectHigherRank hasHigherRank) ObjectPropertyRange(hasDirectHigherRank Taxon) ObjectPropertyDomain(hasDirectHigherRank Taxon) TransitiveObjectProperty(hasLowerRank) ObjectPropertyRange(hasLowerRank Taxon) ObjectPropertyDomain(hasLowerRank Taxon)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-SP, LP-UR, LP-UO and LP-Di.

A.11 Multimedia

Slot	Value
General Information	
Name	Multimedia Data Segment Descomposition
Identifier	CP-MDSD-01
Type of Component	Content Pattern (CP)

2006–2008 © Copyright lies with the respective authors and their institutions.

Use Case	
General	This pattern represents decompositions of multimedia content into segments.
Examples	Suppose that someone wants to represent a specific algorithm for segmenting videos into temporal scenes.
Ontology Design Pattern	
Informal	
General	The pattern partially consist of the classes 'Media', 'SegmentDecomposition', 'InputSegmentRole', 'OutputSegmentRole', 'SegmentationAlgorithm' and 'MultimediaData' and relation 'plays' between 'MultimediaData' and 'Role'. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the classes *Class* and *ObjectProperty* with the universal restriction *allValuesFrom* and the existential restriction *someValuesFrom*. The object properties *inverseOf*, *subClassOf* and *subPropertyOf* and the relation between ontologies *import* are also instantiated.
Graphical	

<table>
<tr>
<td>(UML) Diagram for the General Solution</td>
<td>
</td>
</tr>
<tr>
<td colspan="2">Formalization</td>
</tr>
<tr>
<td>General</td>
<td>Imports(<http://www.ontologydesignpatterns.org/
codeps/owl/informationrealization.owl>
Imports(<http://www.ontologydesignpatterns.org/
codeps/owl/objectrole.owl>)
Imports(<http://www.ontologydesignpatterns.org/
codeps/owl/descriptionandsituation.owl>)
SubClassOf(MultimediaData ObjectSomeValuesFrom(plays
OutputSegmentRole))
SubClassOf(MultimediaData
ObjectSomeValuesFrom(<informationrealization.owl#isRealizedBy> Media))
SubClassOf(MultimediaData
ObjectAllValuesFrom(<informationrealization.owl#isRealizedBy> Media))
SubClassOf(SegmentationAlgorithm
DataAllValuesFrom(<description.owl#defines> MultimediaData))
SubClassOf(SegmentationAlgorithm
ObjectAllValuesFrom(<descriptionandsituation.owl#isSatisfiedBy>
SegmentDecomposition))</td>
</tr>
</table>

2006–2008 © Copyright lies with the respective authors and their institutions.

	SubClassOf(SegmentationAlgorithm <description.owl#Description>) SubClassOf(SegmentationAlgorithm DataMinCardinality(1 <description.owl#defines>)) SubClassOf(OutputSegmentRole ObjectAllValuesFrom(playedBy MultimediaData)) SubClassOf(OutputSegmentRole <objectrole.owl#Role>) SubClassOf(InputSegmentRole <objectrole.owl#Role>) SubClassOf(InputSegmentRole ObjectAllValuesFrom(playedBy MultimediaData)) SubClassOf(SegmentDecomposition ObjectSomeValuesFrom(<situation.owl#isSettingFor> ObjectIntersectionOf(ObjectSomeValuesFrom(plays OutputSegmentRole) MultimediaData))) SubClassOf(SegmentDecomposition <situation.owl#Situation>) SubClassOf(SegmentDecomposition ObjectSomeValuesFrom(<situation.owl#isSettingFor> ObjectIntersectionOf(MultimediaData SubClassOf(SegmentDecomposition ObjectSomeValuesFrom(<descriptionandsituation.owl#satisfies> SegmentationAlgorithm)) SubClassOf(Media <informationrealization.owl#InformationRealization>) SubClassOf(Media ObjectAllValuesFrom(<informationrealization.owl#realizes> MultimediaData)) SubObjectPropertyOf(<objectrole.owl#isRoleOf> <classification.owl#isClassifiedBy>) ObjectPropertyRange(playedBy MultimediaData) ObjectPropertyDomain(playedBy <objectrole.owl#Role>) SubObjectPropertyOf(playedBy <objectrole.owl#isRoleOf>) InverseObjectProperties(plays playedBy) ObjectPropertyRange(plays <objectrole.owl#Role>) ObjectPropertyDomain(plays MultimediaData) SubObjectPropertyOf(plays <objectrole.owl#hasRole>) SubObjectPropertyOf(<objectrole.owl#hasRole> <classification.owl#classifies>)
Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC, LP-SC, LP-OP, LP-SP, LP-UR, LP-ER, AP-MD, CP-OR, CP-IR and CP-DAS.

Bibliography

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford Press, 1979.

[ATS+07] 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.

[BFL98] 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.

[Blo05] 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.

[BM05] D. Brickley and L. Miller. Foaf vocabulary specification. Working draft, 2005.

[Bra77] Ronald J. Brachman. A Structural Paradigm for Representing Knowledge. Ph.d. thesis, Harvard University, USA, 1977.

[BS05] Eva Blomqvist and Kurt Sandkuhl. Patterns in ontology engineering: Classification of ontology patterns. In Chin-Sheng Chen, Joaquim Filipe, Isabel Seruca, and JosŽ Cordeiro, editors, *ICEIS (3)*, pages 413–416, 2005.

[CLN+07] Carola Catenacci, Jos Lehmann, Malvina Nissim, Valentina Presutti, and Geri Steve. Design rationales for collaborative development of networked ontologies Ð state of the art and the collaborative ontology design ontology. Deliverable D2.1.1, NeOn project, 2007.

[CTP00] 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.

[Dol] DOLCE - Project Home Page. http://dolce.semanticweb.org.

[DWW04] 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.

[ea05] Kendall E. et al. Ontology Definition Metamodel. http://www.omg.org/docs/ad/05-04-13.pdf, 2005. Revised submission to OMG.

2006–2008 © Copyright lies with the respective authors and their institutions.

[Gan05] 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.

[Gar83] Howard Gardner. *Frames of Mind: The Theory of Multiple Intelligences.* Basic Books, New York, 1983.

[GB04] Aldo Gangemi and Stefano Borgo, editors. *Core Ontologies in Ontology Engineering 2004. (Un)Successful cases and best practices for ontology engineering: reusing well-founded ontologies for domain content specification.*, Proceedings of the EKAW*04 Workshop on Core Ontologies in Ontology Engineering, Northamptonshire (UK), October 2004. Laboratory for Applied Ontology (ISTC-CNR) Italy, CEUR.

[GBCL05] Aldo Gangemi, Stefano Borgo, Carola Catenacci, and Jos Lehmann. Task Taxonomies for Knowledge Content. Deliverable D07 of the Metokis Project. http://www.loa-cnr.it/Papers/D07_v21a.pdf, 2005.

[GCB04] 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.

[GCCJ06] 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.

[GF94] M. Gruninger and M. Fox. The role of competency questions in enterprise engineering, 1994.

[GFK+04] 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 Aldo Gangemi Stefano Borgo, editor, *Proceedings of the EKAW*04 Workshop on Core Ontologies in Ontology Engineering*, Northamptonshire (UK), 2004. CEUR.

[GHJV95] 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.

[GLP+07] 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.

[GP07] 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.

[GPS01] 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.

[Gru93] T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[GW99] J.-M. Le Goff and I. Willers. Design patterns for description-driven systems, 1999.

[GW04] 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.

[Hay96] David C. Hay. *Data Model Patterns*. Dorset House Publishing, 1996.

[HBP+07] Peter Haase, Saartje Brockmans, Raul Palma, Jerome Euzenat, and Mathieu dÕAquin. Updated Version of the Networked Ontology Model. Deliverable D1.1.2, NeOn project, 2007.

[Hea92] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 539–545, 1992.

[MB05] 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/.

[MGG+05] Claudio Masolo, Aldo Gangemi, Nicola Guarino, Alessandro Oltramari, and Luc Schneider. The wonderweb library of foundational ontologies. Wonderweb deliverable d18, Laboratory for Applied Ontology (ISTC-CNR), 2005.

[MHG01] 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.

[MHG02] 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.

[ML00] 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.

[MPPdC+07] Elena Montiel-Ponsoda, Wim Peters, Guadalupe Aguado de Cea, Mauricio Espinoza, Mari Carmen Suárez-Figueroa, José Ángel, Ramos Gargantilla, Asunción Gómez-Pérez, Inmaculada Álvarez de Mon, Margherita Sini, Aldo Gangemi, Óscar Munoz, and Raúl Palma. Multilingual ontology support. Deliverable D2.4.1, NeOn project, 2007.

[NA05] 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).

[Obe06] Daniel Oberle. *Semantic Management of Middleware*, volume I of *The Semantic Web and Beyond*. Springer, New York, FEB 2006.

[Obj04] Object Management Group (OMG). Unified modeling language specification: Version 2, revised final adopted specification (ptc/04-10-02), 2004.

[oM03] National Library of Medicine. Unified medical language system UMLS, 2003. http://www.nlm.nih.gov/research/umls/.

[OMGS04] 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.

[ont] Ontology design patterns web portal. http://www.ontologydesignpatterns.org.

[OWL04] OWL Web Ontology Language Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics/, 2004.

[PNL02] Adam Pease, Ian Niles, and John Li. The Suggested Upper Merged Ontology: A large ontology for the semantic web and its applications. In *Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web*, Edmonton, Canada, 2002.

2006–2008 © Copyright lies with the respective authors and their institutions.

[RDH+04] Alan L. Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. Owl pizzas: Practical experience of teaching owl-dl: Common errors & common patterns. In *EKAW*, pages 63–81, 2004.

[Rei00] Jacqueline RenŽe Reich. Ontological design patterns: Metadata of molecular biological ontologies, information and knowledge. In Mohamed T. Ibrahim, Josef KŸng, and Norman Revell, editors, *DEXA*, volume 1873 of *Lecture Notes in Computer Science*, pages 698–709. Springer, 2000.

[RR04] 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.

[SAd+07] Marta Sabou, Sofia Angeletou, Mathieu d'Aquin, 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.

[San91] B. Santorini. Part-of-speech tagging guidelines for the penn treebank project. http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/Penn-Treebank-Tagset.ps, 1991.

[SFBG+07] 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.

[SJN04] R. Snow, D. Jurafsky, and A.Y Ng. Learning syntactic patterns for automatic hypernym discovery. *Advances in Neural Information Processing Systems*, (17), 2004.

[SLM06] 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.

[Sos03] 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.

[Sva04] Vojtech Svatek. Design patterns for semantic web ontologies: Motivation and discussion. In *Proceedings of the 7th Conference on Business Information Systems*, Poznan, 2004.

[VDATHKB03] 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.

[VG06] 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.

[Vra05] Denny Vrandecic. Explicit knowledge engineering patterns with macros. In Chris Welty and Aldo Gangemi, editors, *Proceedings of the Ontology Patterns for the Semantic Web Workshop at the ISWC 2005*, Galway, Ireland, NOV 2005.

[VS07] 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.

[VSPS07] Denny Vrandecic, York Sure, Raul Palma, and Francisco Santana. Ontology repository and content evaluation. Deliverable D1.2.10v2, KnowledgeWeb project, 2007.

[VVS06] J. Völker, D. Vrandečić, and Y. Sure. Data-driven change discovery. Deliverable 3.3.3, SEKT Project, 2006.

2006–2008 © Copyright lies with the respective authors and their institutions.