Diploma Thesis
Model-driven Engeneering of Ontology APIs

Stefan Scheglmann

April 14, 2010

WeST

Institute for Web Science and Technologies

Universität Koblenz-Landau

Contents

1 Introduction

In recent years ontologies have become common on the World-Wide Web to provide high level descriptions of specific domains. These descriptions could be effectively used to build applications with the ability to find implicit consequences of their represented knowledge [5]. The W3C developed RDF the Resource Description Framework a language to describe the semantics of the data on the Web and OWL the Ontology Web Language, a family of knowledge representation languages for authoring ontologies. In this thesis we propose an ontology Application Programm Interfaces (API) engineering framework, that makes use of the state of the art ontology modeling technologies as well as software engineering technologies. This system simplifies the design and implementation process of developing dedicated APIs for ontologies. Developers of semantic web applications usually face the problem of mapping entities or complex relations described in the ontology to object oriented representations. Mapping complex relationship structures comming with complex ontologies to an useful API requires more complicated API representations than just mapping concepts to classes. The implementation of correct object persistence functions in such class representations becomes complex too.
The following should show the structure of the thesis and how the different discussions are related. In Section 2 we introduce the related work. In Section 3, we discuss the particularities of create, read, update and delete (CRUD) for ontology based datasets. CRUD are the four basic functions for persistence. We focus on the graph like structures of ontologies and datasets and on the influence of concept semantic on the CRUD operations. Based on this, we propose a model driven approach aware of the complex relationsship structures comming with ODP based ontologies. As initial model, we define the Model for Ontologies (MoOn), a UML based model for logic based ontologies in Section 4. We discuss different requirements to the model, give a specification and an example. This model also serves as vizualization of the ontology and gives the user the opportunity to specialize the ontology and to customize the to-API mapping. In the next Section 5, we discuss the particularities of APIs for ODP based ontologies. This discussion serves us as basis for the introduction of the Ontology API Model (OAM). The OAM serves as programming language independent intermediate model for the pattern-based ontology API. This model gives the user (user of the model driven approach) the opportunity to customize the API. Finally in Section 6, we describe our prototype implementation that supports the two models defined in the previous sections.

1.1 The Running Example

In this section we present a running example. We use this example in all of our discussions about ontology configurations and their influence on ontology API functionalities. Figures 1.1 shows our running example.

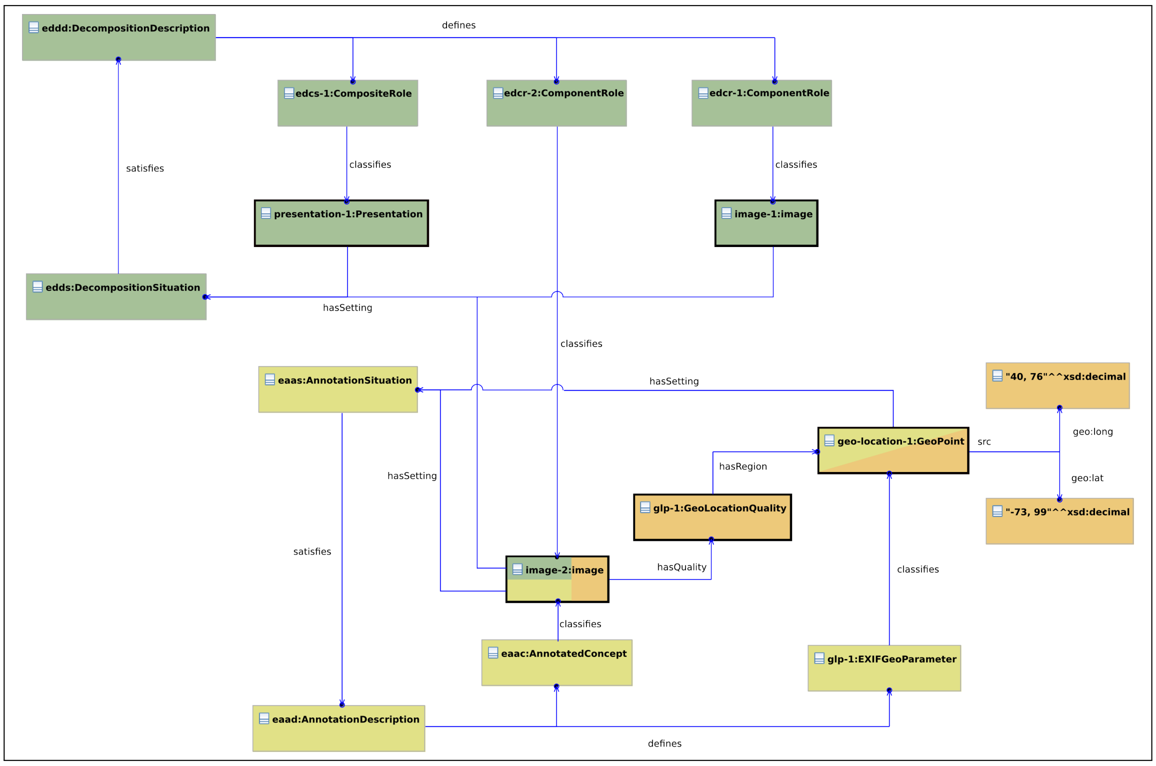


Figure 1.1: Decomposition Pattern and Annotation Pattern applied to an multi-media Presentation

The example presented here should give the reader a idea of combining multiples Ontology Design Patterns (ODPs) in an ontology. ODPs are small ontologies design for a particular problem type. The example consists of configurations (individual structures based on an ODP) of different patterns of the M3O, presented in 2.4. The M3O is an pattern based ontology for multimedia meta data. In the example we use three different Patterns of the M3O. The `AnnotationPattern`, a pattern to model annotation of arbitrary multimedia objects. The `DecompositionPattern`, a pattern to model the decomposition of complex multimedia objects and the `DataValuePattern` designed for the encoding of concrete data. For a detailed explanation of the M3O and the single patterns refer to Section 2.3.1 in the related work section and the explanation of the M3O in the Appendix A.1. We have to denote here that the example represents a individual structure. It describes the concrete individual and not the ontology schema like in the pattern definitions. Figure 1.1 shows our example. The example describes a multimedia object, in this case the `presentation-1:Presentation`, that is decomposed into two

individuals of `Image`, `image-1` and `image-2`. The second image, `image-2` is subsequently annotated with geo-coordinates based on EXIF[1], represented by two `xsd:decimals`, the latitude and the longitude.
The example is build of configurations of the `DecompositionPattern`, the `DataValuePattern` and the `AnnotationPattern` from the M3O. The different individuals are differently colored according to their pattern origin. Individuals that take place in multiple patterns are multicolored. The dark green individuals belong to the `DecompositionPattern`, the light green to the `AnnotationPattern` and the orange individuals belong to the `DataValuePattern`. The multicolored individuals, like the tricolored `image-2:image` and the two colored `geo-location-1:GeoPoint` belong to multiple patterns. In case of `image-1:Image`, the individual plays a role in the three different patterns used here. In the `DecompositionPattern` it is the individual of `InformationObject` classified by one of the `ComponentRole` individuals , `edcr-2:ComponentRole`. One of the components, decomposed from the `Presentation`. As part of the `AnnotationPattern` it takes on the role of the annotated `InformationObject` individual classified by a `AnnotatedConcept`. The third pattern, the `DataValuePattern`, is employed to represent the EXIF geo coordinates. The `image-1` has a `GeoLocationQuality` attached. This `Quality` has a `Region`. The description defines a `GeoPoint` individual, `geo-location-1`. In the `DataValuePattern` this is the `Region` individual, that represents the data space of the geo coordinate. The `AnnotationPattern` parameterizes this `GeoPoint`, by a `EXIFGeoParameter` individual, `glp-1`. Attached to this are latitude and longitude using the WGS84[2] vocabulary.

1.2 The Process Chain

In this section, we outline the process chain of our application. Many of the problems discussed later, have an impact on multiple parts of our process chain. We give the reader an idea of the single parts and their position in the process chain.

This Figure 1.2 gives you a short overview on the single elements of our the process chain in our application. Initially, we would have an pattern based ontology, ideally already in an UML Ontology Model (MoOn) basing on the ODM OWL profile. If not in an UML2, the user has to create an MoOn representation of the ontology. The MoOn is a UML Class Diagram based model providing the user a visualization of the ontology and functionalities to customize the following transformation step. Starting from the MoOn the first step of our work-flow is to transform it to the Ontology API Model (OAM). The OAM is a UML Class Diagram based model. It works as initial representation of the ontology API in the next transformation step of the workflow, the code generation. The OAM provides a programming language independent visualization and opportunities to customize the API structure. Between the single transformation steps the user has the opportunity to customize the single models to adapt the ontology or the API to the concerns of the domain or intended application.

[1]EXIF Exchangeable Image File Format `http://www.exif.org/`
[2]The World Geodetic System `http://en.wikipedia.org/wiki/WGS84`

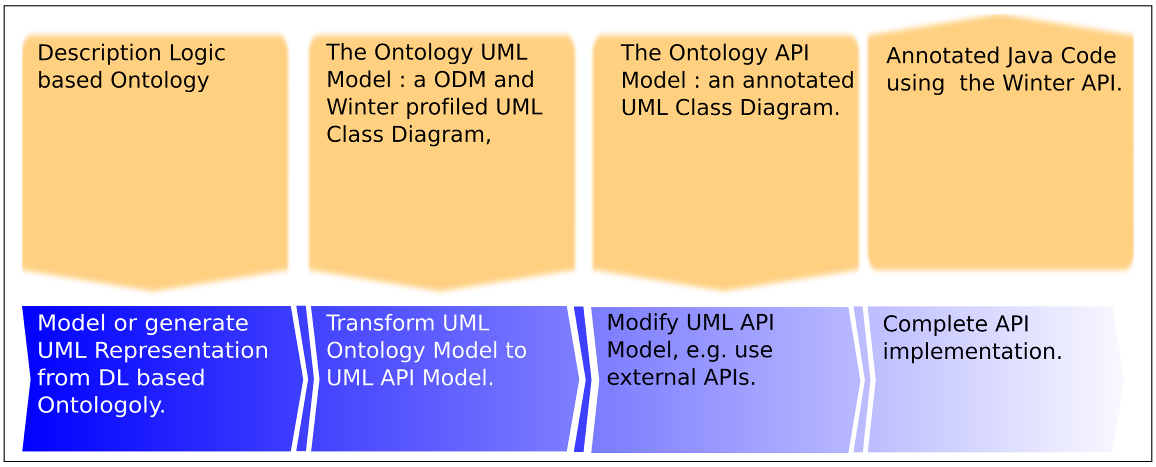

Description Logic based Ontology
The Ontology UML Model : a ODM and Winter profiled UML Class Diagram,
The Ontology API Model : an annotated UML Class Diagram.
Annotated Java Code using the Winter API.
Model or generate UML Representation from DL based Ontologoly.
Transform UML Ontology Model to UML API Model.
Modify UML API Model, e.g. use external APIs.
Complete API implementation.

Figure 1.2: The process Chain of our Approach

2 Related Work

There are different approaches and technologies for API generation. Generating ontology APIs involves various technologies from different fields of computer science. In this chapter, we introduce the basic terms and fundamental concepts. First, we give a short introduction to ontologies in Section 2.1 and discuss the recent work on ontology design patterns (ODP) in Section 2.2. Our process starts with an UML representation of an pattern based ontology. As a general example to patterns in ontologies we introduce some fundamental patterns taken from DOLCE in Section 2.3. These patterns are used as basis for the patterns in the two ontologies presented in Section 2.4. Some of those patterns are used in the running examples. We introduce common ontology representation technologies in Section 2.5 and in Section 2.6. The first sections deals with logic based ontology languages. Later in this work we discuss the influence of these logic based languages on our generation process and on ontology APIs design in general. In the second section we discuss graphical ontology representation and especially ontologies in UML2. We present an extension to UML2, the Ontology Definition Metamodel (ODM), for ontologies. We use ontologies defined in this language as initial models in our approach. In the last Section 2.7 of this chapter we present existing systems for ontology API creation and object persistence APIs for ontology data.

2.1 Ontologies in general

The idea of ontological development arose out of metaphysics, a branch of philosophy dealing with the "fundamental nature of being and the world" [15]. Traditionally, the so called western metaphysics was divided into tree main branches the theology, the ontology and the universal science. Whereas ontology is the study of existence and being. The universe of discourse of Ontology enfolds a semantic for basic classes of entities (objects, properties and the nature of change) and their structural relationships. In computer science, ontologies become more relevant in the 1990s in the field of knowledge acquisition and management. "An ontology provides a specification of a conceptualization of generic notions like time and space or of an application domain like knowledge management or life science" [36]. In the next years various definitions were developed and lead to the in nowadays most frequently seen definition "An Ontology is a formal, explicit specification of a shared conceptualization" [37]. Formal refers to the fact that an Ontology should be defined in a language with a given formal syntax and semantic in combination with an explicit definition of all elements of an Ontology. This results in a "machine executable and interpretable ontology description" [36]. In the end the ontology finally represents "consensual knowledge that has been agreed on by a group of people, typically as a result of a social process." [36]

2.2 Ontology Design Patterns

Design patterns in general are proposed by the architect and mathematician Christopher Alexander in the seventies of the last century as a shared guidelines that helps solving design problems [2]. These basic ideas and the *architectural metaphor* are present for years in the various disciplines working on computational Ontologies [2, 12]. The notion of Ontology Design Patterns (ODP) and its current allocation are mainly embossed by Aldo Gangemi and Valentina Presutti from the Institute for Cognitive Sciences and Technology Rome, Italy. They proposed the idea of ODPs, as patterns for ontology design to increase reusability on the design side. Adopting experiences from the field of software engineering, they developed a group of “small (or even cleverly modularized) ontologies with explicit documentation of design rationales, and best reengineering practices.“ [2]. These ontologies are designed under the assumption that there exist classes of problems that can be solved by applying common solutions. [2, 27] This group of ontologies is used as basic building blocks in the field of ontology design. They are called Content Ontology Design Patterns (CPs). In fact such patterns are “small ontologies that mediate between use cases (problem types) and design solutions“ [2]. They were suggested to increase the possible reuse in Ontology design. But not only in the design case benefits could be achieved, also in the disciplines of Ontology evaluation, matching, modularization, interoperability and in the field of API design. The predescribed basic structure that comes with the use of ODPs, could lead to advantages in all these disciplines. Ontologies in information systems must match both domain and task [27] because their description of entities, entity properties and relations are relevant in the tasks performed by the information system. Gangemi and Presutti categorize Ontology Patterns (OP) based of their field of use and level of application. We will mention the categories here and discuss those who are of relevance for our task.

2.2.1 Structural OPs

Structural OPs include Logical and Architectural OPs. Logical OPs are constructs that solve expressivity problems that could occur through the restricted expressivity of the logical formalism that is used for representation. Architectural OPs covering the overall shape of the Ontology. They describe how the Ontology should look like *internal* and *external*. Whereas *internal* refers to the “Logical OPs that have to be exclusively employed when designing an ontology“ [2], and *external* defines “in terms of meta-level constructs, e.g. the modular architecture,..., where the involved ontologies play the role of modules“ [2].

2.2.2 Reasoning OPs

Reasoning OPs provide reasoning engine behavior control for a certain Ontology. They are declared on top of the ontology and give the reasoning system additional information about the state of the ontology and what reasoning has to be performed in order to carry out queries, evaluation etc.

2.2.3 Correspondance OPs

Correspondance OPs include Reengineering OPs and Mapping OPs. Reengineering OPs are described in terms of meta-model transformation rules in order to create a target model (in this case an Ontology) from a starting model (neither an Ontology or a non-ontological source). Gangemi et al. distinguish two different types of Reengineering OPs, the schema reengineering patterns to start from a non OWL DL meta-model and transform it into an OWL DL Ontology and the so called refactoring patterns that provide rules for Ontology transfromation inside of an given ontology language. Mapping OPs provide inter-Ontology realization solutions, so that designers are able to relate two Ontologies without changing one or both of the involved Ontologies types.

2.2.4 Content OPs

> "CPs are distinguished ontologies. They address a specific set of competency questions, which represent the problem they provide a solution for. Furthermore, CPs show certain characteristics, i.e., they are: computational, small and autonomous, hierarchical, cognitively relevant, linguistically relevant and best practices.“ [27]

Content OPs are small autonomous ontologies that provide solutions for domain modeling problems, therefor they address *content* problems [2, 27]. So they deal with the classes and properties that populates a specific ontology and provide solutions for small use cases. Following Gangemi and Presutti, CPs are built from a domain task [27]. Due to their transparency with respect to the concrete design of an ontology, they can additionally work as a tool for ontology evaluation, matching and modularization, etc. [27] Evaluation tasks for example can be applied by testing a concrete ontology against the presence of a certain pattern. *Unit tests* for ontologies discussed by Denny Vrandečić and Aldo Gangemi cf. [27,38]. Mapping or composition of ontologies can also be simplified on ontologies drafted from CPs, by just using the same CPs in the ontologies or by defining mappings between different CPs.

2.2.5 Presentation OPs

Presentation OPs facilitate the readability and usability of ontologies by the user. Examples of these are Naming OPs and Annotation OPs. Whereas annotation is meant here as additional information in order to improve the comprehension of ontologies and their elements. Naming OPs could, following Gangemi and Presutti be understood as a good naming practice, that can boost ontology readability and comprehension by humans. [2]

2.2.6 Lexico-Syntactic OPs

Lexico-Syntactic OPs are basically used to associate simple *Logical* or *Content* OPs with natural language sentences, e.g., for didactic purpose [2, 19].

2.2.7 Leveled ODP classification

Another approach of pattern classification could be found in the publication [6] of Blomqvist and Sandkuhl. They choose a leveled top-down approach starting from an application level and then refine the granularity in every step down to a syntactic level. As we can see in Figure 2.1 and the list below, the whole classification of Blomqvist et al. focuses on the concrete ontology development process based on patterns. The *Design Level* Patterns similar to the Content Patterns in the classification of Gangemi et.al., represent the main building blocks for the concrete development process. The first two Pattern class layers, Application Level and Architecture Level could be seen as the glue that sticks the *Design Level* Patterns together and the last two layers provide a mapping mechanism to be language independent. Figure 2.1 displays a diagram of the five layer classification denoted below.

- *Application Level* Patterns contain information about scope, usage, purpose and context of the implemented ontologies and information about interfaces and relations to other systems

- *Architecture Level* Patterns supply information about the combination and arrangement of all Design Level Patterns.

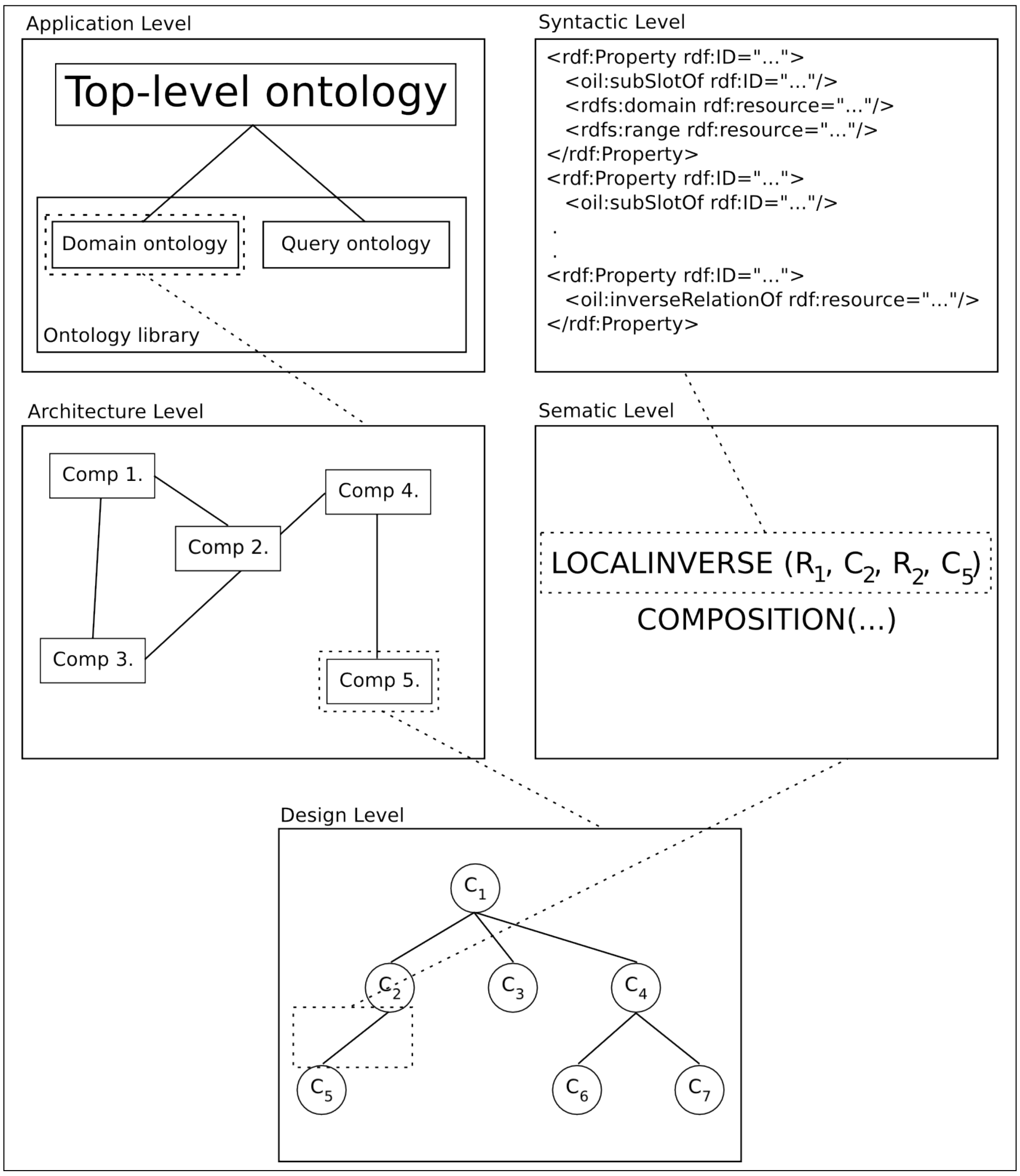
- *Design Level* Patterns are very similar to the content Patterns of Gangemi et al. Design Level Patterns are small semantic building-blocks for constructing ontologies.

- *Semantic Level* Patterns deliver a language independent description of the basic concepts, relations and axioms.

- *Syntactic Level* Patterns are more language specific and thus they provide a mapping for the Semantic Level Patterns.

2.2.8 Summary

As a part of this work we analyze the benefits in the design of an ontology API that could be retrieved from a pattern based ontology. We will focus in this work on the



```
<rdf:Property rdf:ID="...">
  <oil:subSlotOf rdf:ID="..."/>
  <rdfs:domain rdf:resource="..."/>
  <rdfs:range rdf:resource="..."/>
</rdf:Property>
<rdf:Property rdf:ID="...">
  <oil:subSlotOf rdf:ID="..."/>
 .
 .
<rdf:Property rdf:ID="...">
  <oil:inverseRelationOf rdf:resource="..."/>
</rdf:Property>
```

Figure 2.1: The classification levels as continuous increase of granularity

definition of Gangemi et al. The patterns relevant for us are mainly the *Structural OPs* and in this category especially the *Logical OPs* and the *Content OPs*. These are the most common patterns and be found frequently in the current ontology design practice. Nearly every adaptation of a special modeling technique for a special modeling problem might be described as the used of a pattern based approach, even without the knowledge of the pattern idea. Neither if it was used to compensate insufficient expressiveness or to model a specific conceptualization. Whereas a closer formalization of patterns lead to a higher precision and so to a higher re-usability.

2.3 Foundational Ontologies and Core Ontologies

In information science a foundational ontology is an ontology dealing with very general concepts. These conceptual handles should be the same across a broad spectrum of domains or even all domains. This makes it possible to support a broad semantic interoperability for ontologies designed under the assumptions made in this foundational ontology. Foundational ontologies like DOLCE [7,18] are motivated from "philosophical considerations for the construction, comparison, organization and assessment of the ontologies themselves " [7]. The strong generality of foundational ontologies and the fact that a single content pattern could describe or use very general and basic concepts (like object, event, quality or role) and relations (like parthood, participation or dependence) suggest that these well founded ontologies supply a wide range of basic patterns. All these patterns represent more or less formal and domain independent abstractions of philosophical theories. For the designer they provide conceptual handles to carry out coherent and structured analysis of the domain of interest and based on this a toolbox to model this domain [7]. Because of the strong generality and the high degree of abstraction in foundational ontologies, ODPs often implement basic concepts of foundational ontologies. But ODPs are tailored tighter to their domain. At this point we like to mention the core ontologies defined, developed and brought into applications by Oberle (2006), a broad introduction into Core Ontologies can be found in [3]. Core ontologies in general make extensive use of Foundational Ontology *conceptual handles* and following a pattern-oriented design approach in the ontology structure. All these patterns target for a specific field in the area to model and refine very generic foundational ontology patterns by adding detailed concepts and relations of their specific field [3]. Most of the introduced design patterns (*core patterns*) usually base on generic *foundational patterns*. Most of the classes used in the design patterns are derived from the generic classes definitions in *foundational ontologies* like DOLCE+DNS Ultralight (DUL). We will discuss DUL in detail in Section 2.3.1. This ensures that these patterns can be easily applied to express their task with respect to an arbitrary application domain. We will give a closer introduction to two core ontologies, the Event-Model-F and the M3O in Section 2.4. The implementation of the prototype of our system will adopt a combination of those ontologies as testing and evaluation example. But not only the reuse of patterns defined in the foundational ontologies leads to a cleaner, better and even more useful ontologies. The specialization of foundational ontologies basic classes, like mentioned on page 17 in association with Event-Model-F, can lead to a higher formalization even in domain specific ontologies. Therefor in the work with these ontologies, one can utilize all the benefits coming with the use of patterns and specializations of foundational conceptual handles.

2.3.1 DOLCE+DNS Ultralight

All patterns presented here are originally from the DOLCE+DNS Ultralight (DUL) [14], a lightweight version of the DOLCE [18] library and its extensions. The DOLCE+DNS Ultralight ontology is a lightweight foundational ontology for modeling physical or social contexts. DUL presents multiple patterns for modeling such context. The patterns defined in DUL are very complex and provide multiple conceptual handles used in the philosophical definition of the modeled context. These patterns can specialized for the use in domain specific ontologies. In specializations of DUL patterns often not all the conceptual handles from the original patterns are of interest. Multiple specializations for the same domain are possible from one DUL pattern. We present an example for pattern specialization from DUL patterns in the discussion on the ontology of information object (OIO). To avoid ambiguity, we will only introduce a set of specializations per pattern and in order to be unambiguous we recommend to only use one form of each pattern in one ontology. To increase inter ontology compatibility we suggest to use only one form in all the ontologies one designs. Here we can see the strong need for a more formal definition of ODPs to reduce ambiguity and to increase compatibility without the need of Mapping OPs. Using the example of the Ontology of Information Object (OiO) and its realization, the Information Realization Pattern we will also show in this section how to simplify complex Foundational Ontologies to usable Patterns.

The D&S Patterns

The Description and Situation Pattern (D&S) is part of the DOLCE+DNS Ultralight foundational ontology [13]. It is a commonly used pattern in core ontologies, e.g. the M30 or the Eventmodel F. “The D&S pattern allows for the representation of contextualized views on the relations of a set of individuals“ [31].
This feature is highly required in many different domains, e.g. multimedia [31] and event description [33]. The pattern itself consist of a `Description` and a `Situation`, whereas the Situation `satisfies` the `Description`. The `Description` itself `defines` the roles and types present in a concrete context, the `Concepts`. Such a `concept classifies` the entities relevant in the given context. To make the loop complete the entities, themself are connected to the `Situation` via a `hasSetting` relation. With the D&S Pattern it is possible to represent required relationships, which are difficult to represent within the commonly used infrastructures like RDF. The mechanism used therefor is called reification. Reification projects second order entities, like relations, into the first order space. Through that we are able to express information about this second order entities,

which would not be possible without reification, like relationships between more than two items. It allows us to represent contexualized views on relationsships between sets of individuals [31]. You can see the structure of the D&S Pattern in Figure 2.2

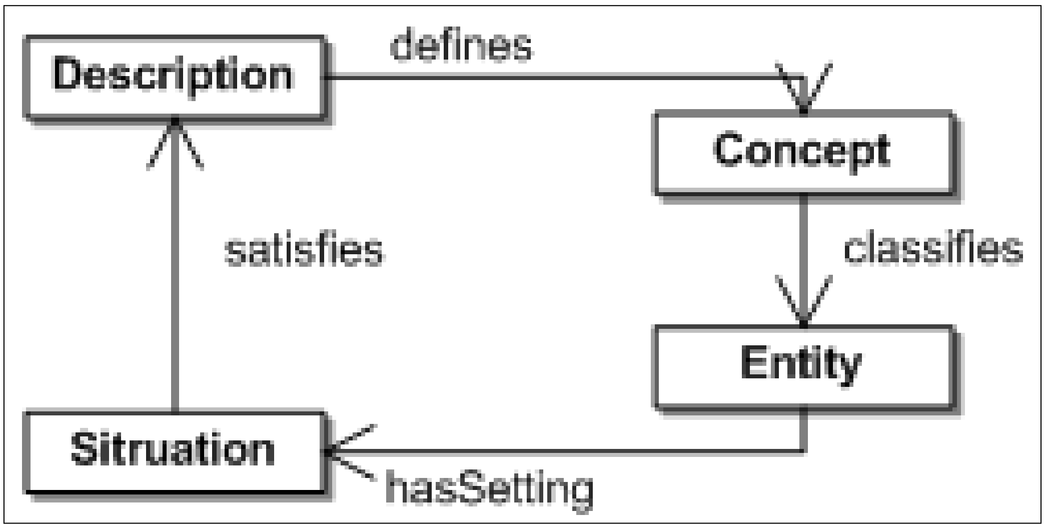


Figure 2.2: The D&S Pattern in its simplest form

The Data Value Pattern

In DUL there exist more than one way to encode concrete data values, but all of them using `Quality` and `Region`. `Quality` is a concept to represent aspects of an `Entity` that *inher* the Entity [18]. `Region` represent the values of qualities and the data space they belong to. The Data Value Pattern we propose here is taken from the M3O [31]. Figure 2.3 shows the structure of the DataValuePattern.

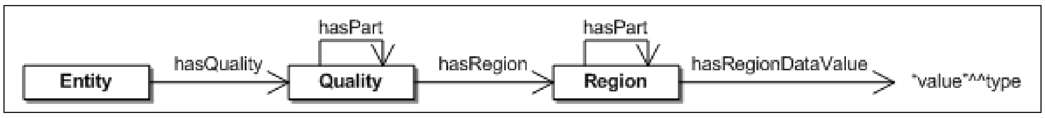


Figure 2.3: The Data Value Pattern

The Ontology of Information Object Pattern

Another Pattern from the DUL is the Ontology of Information Object (OIO) pattern and the therefrom derived Information Realization Pattern. It models the disparity between the information object and its realizations. As a concrete example we will take a digital image and its realization. We introduce an abstract image entity called

InformationObject. This representation is independent from the various realizations, the InformationRealizations. As you can imagine a image could be stored in various different formats and/or resolutions (like the image and a thumbnail). All of these are the realizations. In an ontology all realizations refer to the same InformationObject. “Information realization therefore represents the difference between information as an abstract concept and its concrete realization.“ [?] We will use the Ontology of Information Object and the therefrom derived Pattern, the InformationRealizationPattern to show the simplification of a Foundational Ontology to a Core Ontology Pattern and the requirements and the assumptions this process is based on. Nearly all of the Patterns from DOLCE come with additional philosophically founded classes to ensure correctness in the philosophical approach of modeling the given domain. Not all of these classes are needed in the concrete knowledge representation, especially when taking the *open world assumption* into account. In case of the OIO Pattern we can easily see that for example the agent interpreting the InformationObject is only of theoretical use, answering the question, “exist an entity if it is not observed? “. The particular agent or information about the Information-encoding-System are of more theoretical use than for a clean information representation. They do not appear in the proposed InformationRealizationPattern. In Figure 2.4 we show the Ontology of Information Object and in Figure 2.5 the therefrom derived InformationRealizationPattern [31]. As you can see the OIO originally was also derived from the D&S pattern.

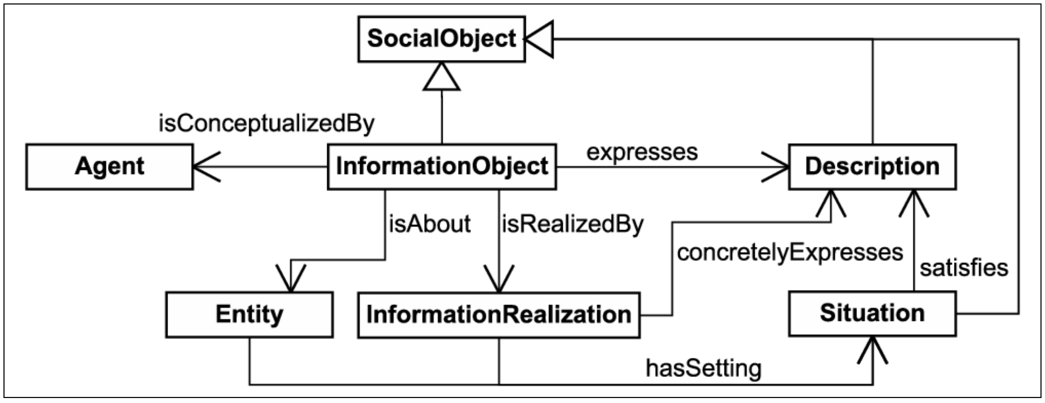


Figure 2.4: The Ontology of Information Object

2.4 Event-Model-F & M3O

In this subsection we will introduce two core ontologies, namely the M3O [31] and the Event-Model-F [33]. Both ontologies include patterns from the preceding section and most of the patterns in these ontologies here are derived from or make use of patterns introduced in the DUL. For a detailed description of the patterns refer to the related work [31, 33] or to the discussion in the Appendix A.1 and A.2.

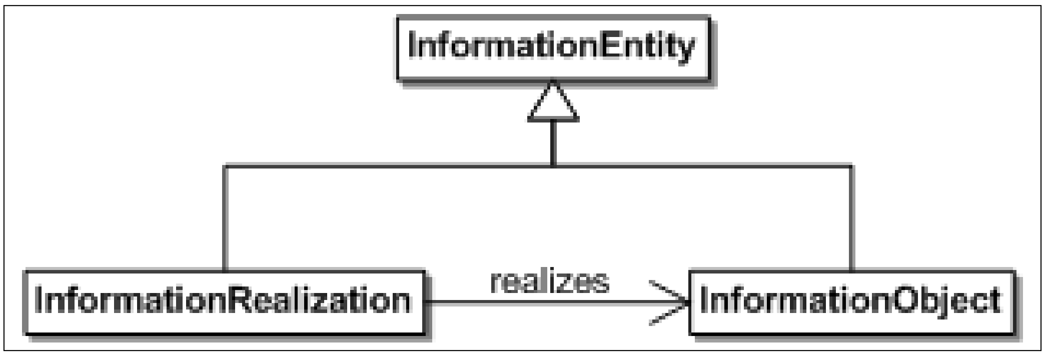


Figure 2.5: The Information Realisation Pattern

2.4.1 The M3O

The Multimedia Metadata Ontology “provides a comprehensive modeling framework for representing arbitrary multimedia meta-data“ [31]. The main motivation to design it was the fact that meta-data and semantic annotations on multimedia content is the decisive factor to improve services, like retrieval, archiving and management, on multimedia content. Existing meta-data models and standards for multimedia data often serve specific purposes and goals, and have different scopes and levels of detail and so they are not combinable. Not least they are often semantically ambiguous “i.e., they do not provide a well-defined interpretation of the meta-data“. [31] To solve these problems the M3O provides several commonly used patterns “underlying existing meta-data models and meta-data standards“ [31] as ODPs. We use some of the patterns from the M3O in our running examples. For all patterns from the M3O, not covered by the previous section refer to the Appendix A.1.

2.4.2 The Event-Model-F

This Ontology defines a formal model of events, it provides “comprehensive support to represent time and space, objects and persons, as well as mereological, causal, and corelative relationships between events.“ [33] With the Event-Model-F it is possible to create ontological representations of events, event causalities and event correlations. The design of the Event-Model-F is aligned to patterns defined in the DUL, especially the D&S pattern is heavily used. Like in the M3O most of the first level entities in the Event-Model-F have been specialized from the classes defined in DUL, e.g. `DUL:Event` ,`DUL:Object` or `DUL:Abstract`.

2.5 Ontologies in the Semantic Web

In this section we give an introduction to ontology representation in general and in the the Semantic Web context. We start with todays standard formalism for knowledge representation the Description Logics(DL). Then we introduce the Resource Description Framework (RDF) as basis for Ontology Web Language (OWL) [5, 20, 34] and language of our serialization. At last we focus on OWL defined by the W3C. OWl is a family of DL based languages designed for ontologies in the Semantic Web.

2.5.1 Description Logics

Description Logics are not a knowledge formalism designed for the Semantic Web. We introduce them here as the formal basis for the knowledge representation techniques use by the Semantic Web technologies. Many of our following discussions will first focus on the DL constructs and then on the Semantic Web technologies. Description Logics are a subset of first-order logics designed to represent the knowledge of a given application domain. Basically logics provides two disjoint alphabets of symbols. The first is used to denote *atomic concepts*, designated by unary predicate symbols, individuals designated by constants and *atomic roles*, designated by binary predicate symbols - the latter are used to express relationships between concepts. [5]. First we introduce the basic concepts of the domain (the terminology), and then we specify properties of the objects and individuals that occur in the domain, by using the concepts defined before. So if we want to describe the concept of e.g. "A man that is married to a doctor and has at least 5 children, all of whom are professors" (this and the following examples are taken from [11]) we can use the following concept description

$$\text{Human} \sqcap \neg\text{Female} \sqcap \exists\text{married.Doctor} \sqcap (\geq 5\text{hasChild}) \sqcap \forall\text{hasChild.Professor}$$

As we can see, some commonly known operators are used in this description. In the context of DLs these operators take on the role of constructors, e.g. the *conjunction* ($\sqcap$) is interpreted as a set intersection constructor, the *negation* ($\neg$) as a set complement constructor, the *existential restriction* constructor ($\exists$R.C), the *value restriction* constructor ($\forall$R.C) and the *number restriction* constructor such as ($\geq$ nR) are interpreted as constructors with the functionality known from common algebra [11]. These constructors could have impact on our ontology to API mapping. In a later Chapter we will discuss the direct influence those constructors have. Table 2.1 shows the DL concept constructors and Table 2.2 the role constructors. Each particular DL comes with its concept and role constructors that could be used to create expressions from the atomic concepts and roles. These constructors determine the expressiveness of the DL and hence also the computational complexity of services performed on the knowledge base expressed in this DL. We will talk about the connection of computational complexity and expressiveness of the language, and the impact on the design of DLs later in this section. DLs define usually in addition to this descriptive formalism a terminological and an assertional formalism. Terminological axioms are basically used to define names for complex descriptions. Assertional formalisms otherwise could be used to state properties of individuals. This separation of concerns between terminological definition and assertional definition mechanisms leads us to the definition of the so called T-Box (terminological) and the A-Box(assertional). In the T-Box new concepts can be defined in terms of previously defined concepts, some common assumptions about most DLs terminologies are:

- only one definition per concept name is allowed;
- definitions are *acyclic*, that means that concepts are not defined in terms of themselves or in terms of concept that indirectly refer to them.

This restriction implies that defined “concepts can be expanded in an unique way into complex expressions containing only atomic concepts by replacing every defined concept with the righthand side of its definition.“ [5]. For example:

$$\text{Man} \equiv \text{Person} \sqcap \text{Male}$$

The A-Box on the other hand contains extensional knowledge about the domain. This additional knowledge could be separated into *concept assertions* such as

$$\text{Male} \sqcap \text{Person(Bob)}$$

and *role assertions.*

$$\text{hasSon(Bob, Jacob)}$$

Whereas in assertions of the first kind general concept expressions are typically allowed, while in *role assertions* they are not allowed.

Table 2.1: DL concept constructors

NAME	SYNTAX	SEMANTICS
Intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Complement	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
Value Restriction	$\forall R.C$	$\{a \in \Delta^{\mathcal{I}} \vert \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$
Existential quantifier	$\exists R.C$	$\{a \in \Delta^{\mathcal{I}} \vert \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
Unqualified number restriction	$\leqslant n\ R$ $\geqslant n\ R$ $= n\ R$	$\{a \in \Delta^{\mathcal{I}} \vert \vert\{b \in \Delta^{\mathcal{I}} \vert (a,b) \in R^{\mathcal{I}}\}\vert \geq n\}$ $\{a \in \Delta^{\mathcal{I}} \vert \vert\{b \in \Delta^{\mathcal{I}} \vert (a,b) \in R^{\mathcal{I}}\}\vert \leq n\}$ $\{a \in \Delta^{\mathcal{I}} \vert \vert\{b \in \Delta^{\mathcal{I}} \vert (a,b) \in R^{\mathcal{I}}\}\vert = n\}$
Qualified number restriction	$\leqslant n\ R.C$ $\geqslant n\ R.C$ $= n\ R.C$	$\{a \in \Delta^{\mathcal{I}} \vert \vert\{b \in \Delta^{\mathcal{I}} \vert (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\vert \geq n\}$ $\{a \in \Delta^{\mathcal{I}} \vert \vert\{b \in \Delta^{\mathcal{I}} \vert (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\vert \leq n\}$ $\{a \in \Delta^{\mathcal{I}} \vert \vert\{b \in \Delta^{\mathcal{I}} \vert (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\vert = n\}$
Equivalence Non-Equivalence	$u_1 \equiv u_2$ $u_1 \not\equiv u_2$	$\{a \in \Delta^{\mathcal{I}} \vert \exists b \in \Delta^{\mathcal{I}}.u_1^{\mathcal{I}}(a) = b = u_2^{\mathcal{I}}(a)\}$ $\{a \in \Delta^{\mathcal{I}} \vert \exists b_1, b_2 \in \Delta^{\mathcal{I}}.u_1^{\mathcal{I}}(a) = b_1 \neq b_2 = u_2^{\mathcal{I}}(a)\}$

Table 2.2: DL role constructors

NAME	SYNTAX	SEMATICS
Universal role	U	$\Delta^\mathcal{I} \times \Delta^\mathcal{I}$
Intersection	$R \sqcap S$	$R^\mathcal{I} \cap S^\mathcal{I}$
Union	$C \sqcup D$	$R^\mathcal{I} \cup S^\mathcal{I}$
Complement	$\neg R$	$\Delta^\mathcal{I} \times \Delta^\mathcal{I} \backslash R^\mathcal{I}$
Inverse	R^-	$\{(b,a) \in \Delta^\mathcal{I} \times \Delta^\mathcal{I} \| (a,b) \in R^\mathcal{I}\}$
Composition	$R \circ S$	$R^\mathcal{I} \circ S^\mathcal{I}$
Transitive closure	R^+	$\bigcup_{n\geq 1}(R^\mathcal{I})^n$
Reflexive-transitive closure	R^*	$\bigcup_{n\geq 0}(R^\mathcal{I})^n$
Role restriction	$R\|_c$	$R^\mathcal{I} \cap (\Delta^\mathcal{I} \times C^\mathcal{I})$
Identity	$id(C)$	$\{(d,d) \| d \in C^\mathcal{I}\}$

Computational Complexity

DLs can be classified in different classes of expressiveness, depending on the constructors the DLs provide for expression creation. This expressiveness has a general impact on the computational complexity of reasoning tasks performed on the knowledge representation. There are different kinds of reasoning that could be performed on a DL system and most of the DL systems are especially designed to support such inferencing. The inferences could be separated depending on what they address, the concepts alone, the T-Box, the A-Box or T-Box and A-Box together. Interesting relationships between concepts are (all these definitions are taken from [5]): Let $\mathcal{T}$ be a T-Box.

- Satisfiability: A concept C is *satisfiable* w.r.t $\mathcal{T}$ if there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^\mathcal{I}$ is nonempty. In this case we say also that $\mathcal{I}$ is a *model* of C.
- Subsumption: A concept C is *subsumed* by a concept D w.r.t $\mathcal{T}$ if $C^\mathcal{I} \subseteq D^\mathcal{I}$ for every model $\mathcal{I}$ of $\mathcal{T}$. In this case we write $C \sqsubseteq_\mathcal{T} D$ or $\mathcal{T} \models C \sqsubseteq D$.
- Equivalence: Two concepts C and D are *equivalent* w.r.t $\mathcal{T}$ if $\mathcal{C}^\mathcal{I} = D^\mathcal{I}$ for every model $D^\mathcal{I}$ for every model $\mathcal{I}$ of $\mathcal{T}$. In this case write $C \equiv_\mathcal{T} D$ or $\mathcal{T} \models C \equiv D$.
- Disjointness: Two concepts C and D are *disjoint* with respect to $\mathcal{T}$ if $\mathcal{C}^\mathcal{I} \cap D^\mathcal{I} = 0$ for every model $\mathcal{I}$ of $\mathcal{T}$.

Taking in account the fact that all actual DL systems provide the intersection operator $\sqcap$ and most of them contain the unsatisfiable concept $\bot$ the following propositions are possible. According to Proposition 1 all the four inferences can be reduced to subsumption. Additionally in DL systems that allow the negation of concepts, according to Proposition 2 all can be reduced to the satisfiability problem.(The Proposition are taken from [5]). Proposition 3 shows us that unsatisfiability is a special case of the other problems.

Proposition 1. *(Reduction to Subsumption)*

i C *is unsatisfiable* $\Leftrightarrow C$ *is subsumed by* $\bot$*;*

ii C *and* D *are equivalent* $\Leftrightarrow C$ *is subsumed by* D *and* D *is subsumed by* C*;*

iii C *and* D *are disjoint* $\Leftrightarrow C \sqcap D$ *is subsumed by* $\bot$*;*

Proposition 2. *(Reduction to Unsaticfiability)*

i C *is subsumed by* $D \Leftrightarrow C \sqcap \neg D$ *is unsatisfiable;*

ii C *and* D *are equivalent* $\Leftrightarrow$ *both* $(C \sqcap \neg D)$ *and* $(\neg C \sqcap D)$ *are unsatisfiable;*

iii C *and* D *are disjoint* $\Leftrightarrow C \sqcap D$ *is unsatisfiable;*

Proposition 3. *(Reducing Unsatisfiability) Let* C *be a concept. Then the following are equivalent:*

i C *is unsatisfiable;*

ii C *is subsumed by* $\bot$

iii C *and* $\bot$ *are equivalent;*

iv C *and* $\top$ *are disjoint;*

Many research has been done on this topic and especially from the viewpoint of complexity assessment in the recent years. For further details and proofs refer to [5] and the additional literature mentioned there. The point of interesting, for us, in this field of research is, as we can see, that possible reductions and thereby the complexity of reasoning depends on how expressive our language is. So the allowed constructors in the ontology representation of the system has a direct impact on how efficient certain task can be performed on the serialization from the created API representation.

2.5.2 RDF - The Resource Description Framework

The family of specifications called Resource Description Framework (RDF) defined by the W3C were originally designed as a meta-data meta model. In the Semantic Web RDF is used as basic model for conceptual description or modeling of information in web resources. There are various syntax formats defined for RDF. In our examples we use the Notation3 (N3) a shorthand serialization of RDF models. With RDF it is possible to make statements about resources in form of subject-predicate-object triples. The subject, a resource is represented through an URI[1]. The predicate expresses a particular relationship between the subject and the object. The predicate is also an URI. The object can be a URI or a literal. A literal is a possibly typed string. For example, a way to express "The dog has the color black" could be a triple: subject denoting "the dog", a predicate denoting "has the color" and a object denoting "black". In N3 the exampe would look like this:

[1]Unified Resource Identifier refer to `http://www.w3.org/Addressing/`

```
@prefix ex: <http://www.example.de/>
ex:thedog ex:hasColor ex:black
```

Multiple RDF statements form a directed labeled multigraph, wherein edges represent predicates and the nodes subject and object. Such graph structures can be stored in one of its serializations in files or in so called triplestores. Triplestores are the databases for triple based data.
Some ontology languages build upon RDF such as RDF Schema (RDFS) or OWL. RDF/RDFS allows the user "the representation of *some* ontological knowledge " [4]. RDFS focuses on the "organization of vocabularies in typed hierarchies: subclass and subproperty relationship, domain and range restrictions and instances of classes" [4], but many features needed for appropriate ontology description are missing. According to [4] these are:

- Local scope of properties
- Disjointness of classes
- Boolean combinations of classes
- Cardinality restrictions
- Special characteristics of properties

For more details on RDF/RDFS refer to [?,8,26], for more details on the missing features refer to [4].

2.5.3 OWL - The Ontology Web Language

OWL is not a single language but a hierarchy of three different sublanguages. OWL bases on RDF/RDFS but OWL is aware of the trade-off between expressive power and efficient reasoning. Ideally OWL would only add what is needed to support the expressiveness that is missing in RDF/RDFS, as identified before. In this case OWL would be just an extension of RDF Schema and would use the RDF meanings of classes and properties. But unfortunately, according to [4], this attempt of just extending RDF Schema clashes with the trade-off between expressive power and computational efficient of reasoning mentioned in Section 2.5.1. Ans so OWL restricts the very expressive modeling primitives, such as `rdfs:Class` and `rdfs:Property` [4]. The W3C defined OWL, in this case OWL 1.1[2], as a hierarchy of the three sublanguages, thereby each of them aims to fulfill a specified subset:

- *OWL FULL* provides maximum expressiveness and full first-order logic support there is no reasoning software that supports every feature of OWL Full.
- *OWL DL* is a subset of OWL FULL for those users how want maximum expressiveness without using any computational completeness and decidability. Most of the ontologies, are modeled using OWL DL.

[2]`http://www.w3.org/Submission/owl11-overview/`

- *OWL Lite* is a subset of OWL DL and only provides a very simple classification hierarchy and simple constraint features.

Due to the ongoing development in the Semantic Web the W3C decided to release a new version of OWL, namely OWL 2[3]. OWL 2 has no longer the strict hierarchy of OWL 1.1. It rather separates into different sublanguages, in the OWL 2 context commonly called profiles, for different concerns. A diagram of the OWL 2 profiles is shown in figure 2.6.

- *OWL 2 EL:* Ontologies formulated in this profile of OWL 2 all basic reasoning services can be performed in polynominal time with respect to the size of the ontology.
- *OWL 2 QL:* This profile is specialized for supporting query support, especially using conventional relational databases.
- *OWL 2 RL:* This profile provides a sublanguage specialized for rule based reasoning engines.

Constructors in OWL DL

Refering to table 2.1 all DL constructors we could find are supported in OWL DL. Intersection as `owl:intersectionOf`, union as `owl:unionOf` e.t.c. , the number restrictions are represented as the different cardinalities, equivalence between individuals as `owl:sameAs`, equivalence between classes `owl:equivalentClass` etc.

2.6 Graphical Ontology Languages

Different notations were developed by different scientific communities in the past, e.g. conceptual graphs [35] and topic maps [32]. Generally all these logical graphs based on the several versions of "graph-theoretic formal languages" [25, `http://en.wikipedia.org/wiki/Logical_graph`] proposed by Charles Sanders Peirce in his papers on qualitative logic, entitative graphs and existential graphs in the late 19th century. Because of their logic nature a popular way to describe and design ontologies is the use of a graphical representation. We used such graphical representation to present all of our examples. The graphical representation of ontologies or even the design of ontologies in a visual syntax implicates several advantages that simplify the conceptual modeling, increases the readability of the ontologies and decreases syntactic and semantic errors [9] made in the design process. The selection of an appropriate modeling language and the available tools for that language could also speed up the modeling process of ontologies. Because of the basic structure of ontologies, the UML Class Diagram and simplifications of it became a popular way to describe Ontologies. All patterns introduced here are displayed in a simplified form of the UML Class Diagram. In the following Sections 2.6.1 we will give an overlook over the related research work dealing with Ontologies in UML

[3]`http://www.w3.org/TR/owl2-profiles/`

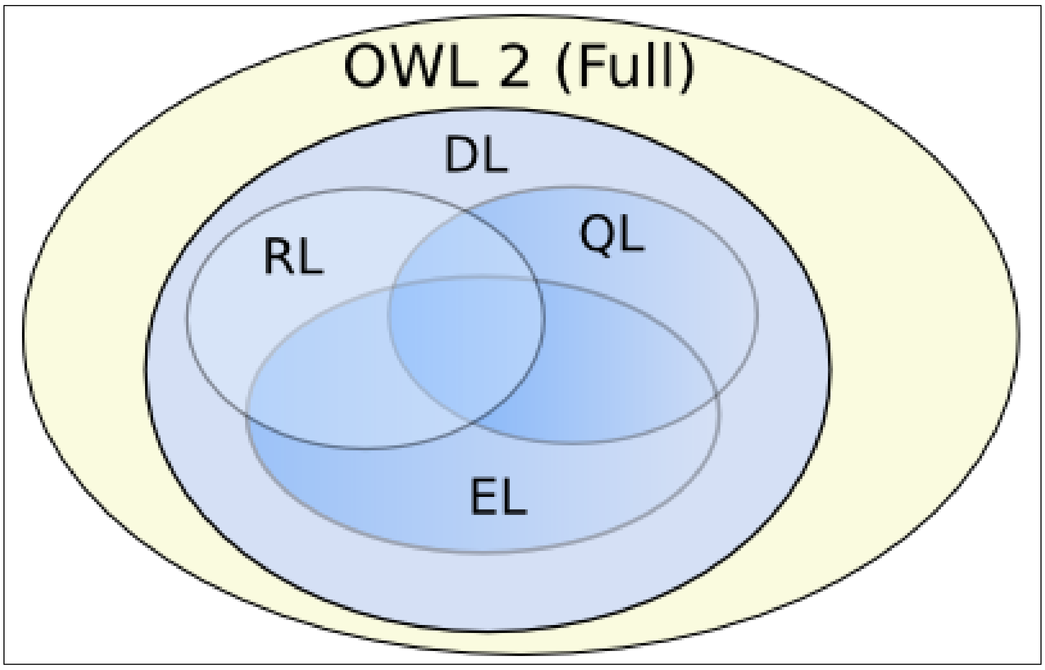


Figure 2.6: Venn Diagram of OWL Syntactic Subsets (Profiles)

2.6.1 Ontologies in UML2

As mentioned the UML Class Diagram is a popular way to describe ontologies. If we refer to a UML model for an ontology in the following, a UML Class Diagram is meant. Because of the quite similar nature including classes and class attributes (properties in UML), class/subclass hierarchies, inter class relationships and concepts to specify constrains, UML has been successfully used to visualize ontologies. However when UML is used in ontology design/visualization often only the inconographic part of the language is used in a way more or less aligned to the specification. Otherwise efforts are made to formalize a model for ontology design based on UML. Originating from the Object Management Group (OMG)[4] and Sandpiper Software[5] the Ontology Definition Metamodel (ODM) [23] was proposed first in 2004. Based on a comparison of OWL Full and UML 2 [16] as a preliminary analysis for the design of the ODM, the OMG[6] specified the ODM in version 1.0 and made it available[7]. In this section, we introduce the ODM and take a closer look at how it bridges the gap between the two paradigms of the object-oriented UML meta-model and the logic based ontology world.

[4]OMG website `http://www.omg.org/`

[5]Sandpiper Software website `http://www.sandsoft.com/`

[6]Defined by the Ontology Working Group in the OMG `http://www.omg.org/ontology/`

[7]Version 1.0 can be found here `http://www.omg.org/spec/ODM/1.0/PDF`

2.6.2 The Ontology Definition Metamodel (ODM)

The OMG, responsible for the specification of UML2, has released the Ontology Definition Metamodel (ODM). It is a collection of meta-models and mappings, capable for ontology design and visualization tasks. The ODM defines a collection of UML metamodels for different ontology languages, such as RDFS/OWL, Topicmaps, common logic and others. The ODM should help ontology designers by giving them a graphical language to design ontologies and transformations from the graphical representation to the different ontology languages. As part of the ODM, the OMG defines lightweight extensions to UML2 for several logic based languages and mappings to be able to transform ontologies modeled in UML2 directly into one of those languages. So an ontology can be designed using UML in combination with the ODM and then be transformed in the intended ontology language. The ODM includes extensions and mappings for, e.g., OWL, topicmaps and common logic. In the case of OWL this extension consists of an UML2 profile. In this profile multiple stereotypes are defined to enrich basic UML2 entities with their OWL nature, such as `owlOntology`, `owlClass` or `owlProperty`. These stereotypes are applicable to one or more UML elements, adequate for modeling concepts or constructors in UML. For example `owlOntology` could be applied to *UML:Package* or *UML:Model*, `owlClass` to *UML:Class* and `objectProperty` to *UML:Properties* or associations. In some cases it is possible to model a specific semantic construct in different syntactical ways. For example we could represent an `objectProperty` through an `objectProperty` stereotyped `UML:Property` or and similarly stereotyped association connecting the class that defines the property with the class representing the type of the property. You can see this in Figure 2.7.

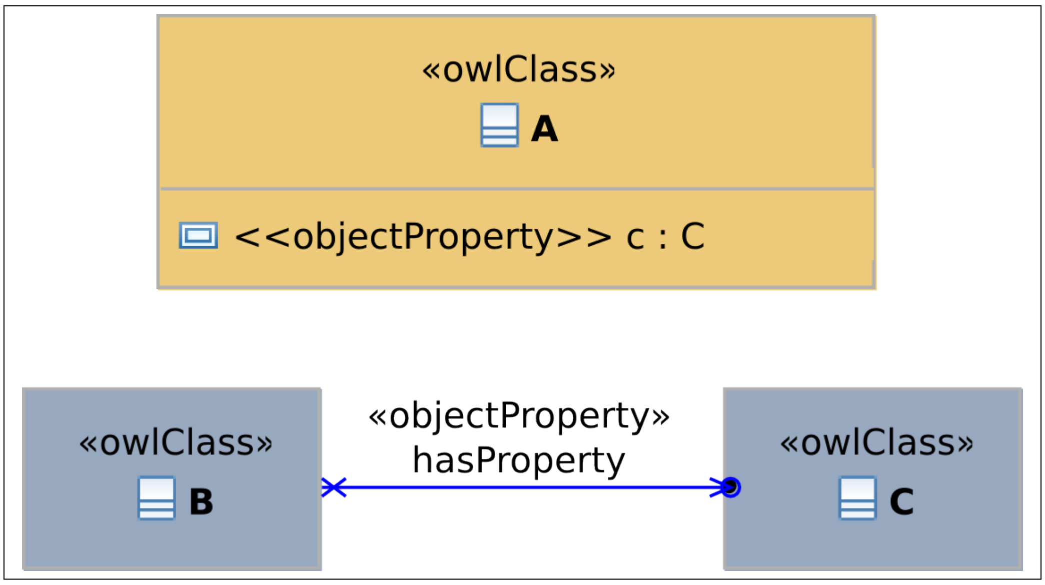


Figure 2.7: The two possible models for ObjectProperty

For some of the semantic constructs in OWL we could not find any adequate coun-

terpart in the UML meta-model. For such a construct the ODM defines modeling conventions (modeling patterns) using multiple UML elements. A good example for such a construct, is `intersectionOf`. According to DL an `owl:intersectionOf` statement describes an anonymous class, its extension contains all the extension of all intersected classes. In UML it is not possible to define a class using set operations on instance sets. The definition of the ODM in version 1.0 defines the `owl:intersectionOf` stereotype as applicable to a *UML:Constraint*. The way to to model an `intersectionOf` according to the definition looks like shown in Figure 2.8. The notation below shows an model with three classes A, B and C, there C inherits from A and B. To denote that the generalizations between C and A, B should not be understood in the UML way but representing the `owl:intersectionOf` we used an stereotyped *UML:Constraint* connected to both of the generalizations. This model misses some of the specifications of `owl:intersectionOf`, for instance the intersection `class` is not anonymous if we model `intersectionOf` in this way. Additionally modeling intersection like this is not well-suited for a following transformation. So we propose to model intersectionOf in another way, closer to the DL meaning and more suitable to the transormation in our approach. The intended way to model an `owl:intersectionOf` would be to use ≪intersectionOf≫ stereotyped generalizations. These are used to define an anonymous class stereotyped with ≪intersectionClass≫, as shown in Figure 2.9. We also modified the models for some other logical concepts with no matching counterpart in UML, like *union*. We will use these modified modeling conventions for our Model for Ontologies (MoOn).
Others like disjointness could be modeled just through stereotyped association. Table 5.1 on Page 61 and following gives a full lists OWL constructs and their corresponding construct in the ODM/UML meta-model. For all OWL constructs with more complicated counterparts in the ODM we present a figure in the appendix.

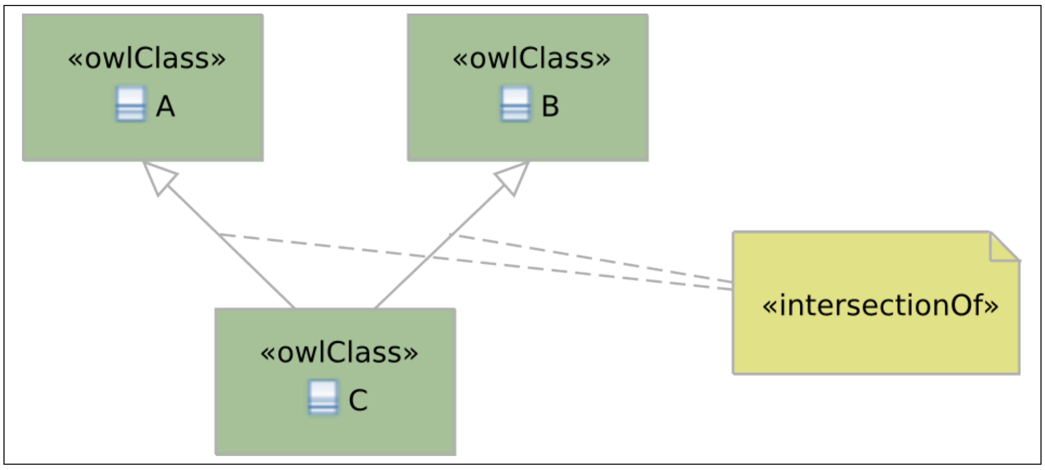


Figure 2.8: IntersectionOf modeled according to ODM v1.0

The real intention behind the definition of the ODM was to define a graphical approach for ontology design. Most of the mappings defined in the ODM are directed from the different UML 2 models towards other models or a ontology language. It is

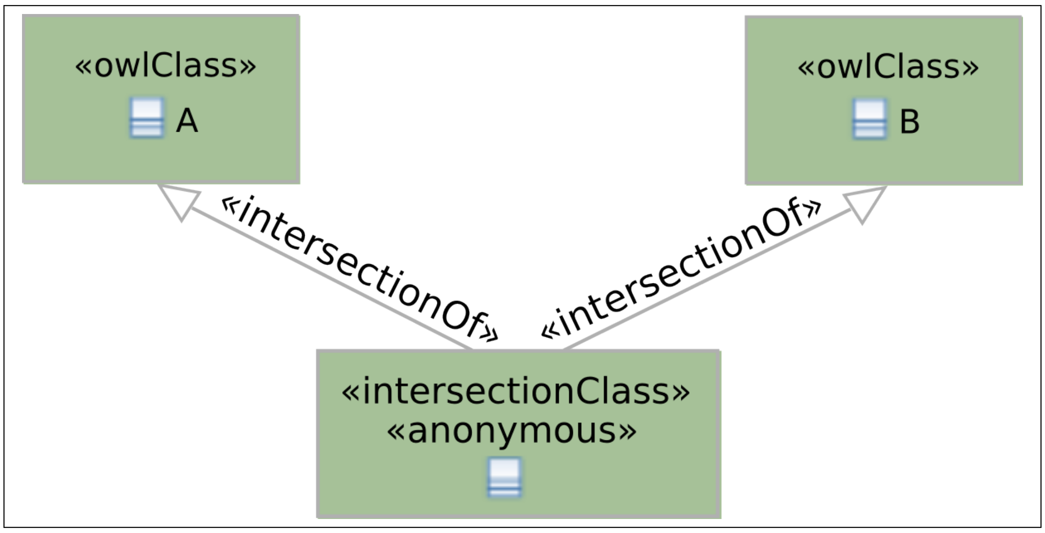


Figure 2.9: Modified IntersectionOf Model

intended to design the ontology in UML and then transform it to the intended synatx like OWL or common logic. Currently there exists only one implementation of an OWL to UML transformation using the inverse of the mappping defined in the ODM. This transformation is present in a plugin for Protege4[8]. This plugin just visualizes OWL using UML2 and the ODM specification. It was developed at the University of Latvia by Igor Istocnik. Unfortunately no more information about it is available.

2.7 Ontology API Frameworks and Automated Ontology API generation

In web applications and enterprise solutions relational data bases (RDBs) and thus object-persistence to such stores were used for years. With RDF and hence also triple stores coming more and more into focus several persistence layers especially designed for the non relational triple stores showed up in the last years. Most of them try to apply the well known RDB object persistence patterns to the non relational world of triplestores. Such an persistence API often comes with a simple ontology API generation tollkit. These simple attempt mostly use concept-to-class mapping for generation and object-to-statements mapping for persistence. Such mappings are suitable for simple structured ontologies like foaf[9]. But for complex ontologies this simple object-to-statement mapping leads us to multiple problems. We come to this in Chapter 5. In this section we give the reader a short look on the most popular projects dealing with object-to-statement persistence and ontology API generation.

[8]The plugin URL `http://protegewiki.stanford.edu/index.php/OWL2UML` last access 01.03.10
[9]The foaf project website `http://www.foaf-project.org` last access 01.03.10

Elmo and AliBaba

Elmo is a role based Java persistent bean pool on top of a SESAME repository. SESAME is a very popular opensource RDF triple store. Aduna Software[10] is the leading company behind SESAME, Elmo and AliBaba. They develope and maintain SESAME and other projects based on SESAME. SESAME comes with its own database infrastructure and provides access to multiple other databases like, jena[11] and mulgara[12]. Elmo provides a simple API to access ontology oriented data. The next generation of the Elmo codebase is called AliBaba[13], both are developed at Aduna Software. Elmos functionalities are base Java-Annotations providing the metadata needed for adequate mapping. As a further developement of Elmo, AliBaba also uses Java annotation to encode mapping meta data. But AliBaba goes much further than Elmo did. AliBaba is a subject-oriented client/server library providing RESTful services, distributed data and file persistence on RDF metadata.

Som(m)er

Som(m)er[14] the „Sematic Object (Metadata) MappeR“, is a very simple library for mapping POJOs (Plain Old Java Objects) to RDF graphs, developed by Henry Story. Som(m)er uses Java-Annotations to add the metadata needed for mapping to Java classes. It uses SESAME as backround repository. Based on the JAVA annotation Som(m)er additionally provides a source code rewriting facility to simplify the use of the API and to speed up the development. It enriches the written and annotated code with the methods needed for mapping. Like the other solutions presented below, Som(m)er provides only simple object-to-statement mapping functionality and thus hardly fits for API development for complex ontologies. Some of the fundamental ideas of the implementation of Som(m)er have been taken over into the implementation of Winter.

Winter

Winter is a object-to triplestore persistence layer based on the basic ideas behind So(m)mer. Like So(m)mer it is Annotation based. Winter provides special features targeting the issues coming with complex ontology structures and especially with the use of ODPs. Winter expands the standard object-to-statement approach in a way that it simplifies the development of ontology APIs for the integration of ontology based semantics into an application. In chapter 3 we examine ontologies and especially ODPs according to the desired behavior in applications using, manipulating and creating ontology aligned semantic data. Winter provides functionalities needed for such applications. Different to other persistence APIs winter supports the mapping of objects to multiple statements. In object-to-statement mapping each object is mapped to a singe statement like,

[10]The Aduna Software website `http://www.aduna-software.com/` last access 01.03.10
[11]The Jena sematic web framework project website `http://jena.sourceforge.net/` last access 01.03.10
[12]The Mulgara Sematic Store website `http://www.mulgara.org/` last access 01.03.10
[13]AliBaba project website `http://www.openrdf.org/doc/alibaba/2.0-alpha4/` last access 01.03.10
[14]The Som(m)er project `https://sommer.dev.java.net/sommer/index.html` last visit 10.01.10

$ID < rdfType > TYPE$. Winter allows us to annotate classes with patterns representing multiple statements for the class. Winter uses SPARQL [28] as pattern language in the annotation. Additionally to this Winter provides basic functionalities to solve completeness or validity problems that could arose in the serialization. In Winter it is possible to generate differnt mapping behavior for properties or classes according to their relationships. Winter can distinguish between mandatory and optional relationships of objects and can behave in different ways depending on the relevant relationships. The Winter annotation allows us to declare special types in the declaration. Those types lead to customized behavior in the mapping performed on the annotated class. For more on Winter refer to Section 6.2 on Page 79.

Jena Semantic Web Framework

The Jena Semantic Web Framework (Jena)[15] is a popular and powerful framework not only providing an RDF-Store. Jena also includes an OWL API for loading and working with OWL ontologies. This API provides functionalities for basic reasoning task, ontology modification and instance model processing. Initially the Jena framework was developed in the HP Labs Sematic Web Programm[16].

Owl2Java

Owl2Java [17] is a java code generator for OWL ontologies with support for the Jena semantic web framework developed by Michael Zimmermann at the the Chair of Naval Architecture at the University of Rostock. Additionally it defines another layer called Owl2Db4o that provides a native interface to the object oriented database db4o. The part of the framework with jena support is called Owl2Jena and it generates Java APIs from given ontology schema. Basically it provides a java class for each owl class and data access through methods. Internally these actions are translated to triple constructs of the Jena DB and the mapping task are performed by Jena. OWL2Java does not provide any CRUD behavior control and an intermediate control layer has to implemented on top of the classes generated by OWL2Java.

SWeDE and Kazuki

SWeDE[17], the Semantic Web Developement Environment is an eclipse IDE based framework for the developement of sematic web tools. Currently it integrates apart from an OWL editor with syntax highlighting, spell-checking and auto-completion, useful tools like the OWL Validator, the DumpOnt a visualizer for Ontologies and finally Kazuki. Kazuki is a Java API generation toolkit based on Jena. The Kazuki library provides functionality for generating object oriented interfaces for individuals from OWL ontologies. Kazuki automatically creates Java Interfaces for concepts contained in the choosen OWL Ontology. From each OWL class Kazuki generates two Interfaces, one standard

[15]The Jena sematic web framework project website `http://jena.sourceforge.net/` last visit 10.01.10
[16]`http://www.hpl.hp.com/semweb/` last visit 15.03.10
[17]The official Pulgin Site `http://owl-eclipse.projects.semwebcentral.org/` last access 01.03.10

implementation and one for user customization. All classes build by Kazuki are build up on the Jena2 Ontology-API.

JenaBeans

JenaBean[18] is again an approach using the Jena RDF/OWL api. Jenabean is a typical representative of a Java annotation based object-to-statement mapper.

RDFReactor

RDFReactor[19] realizes the RDF data model in the object oriented environment through Java dynamic proxies[20]. It provides a code generator to generate class representations from ontologies.

ActiveRDF

ActiveRDF[21] is a data layer to access RDF data in Ruby-on-Rails, similar to ActiveRecord (O/R mapping (ORM[22])for relational databases). ActiveRDF provides a Domain Specific Language (DSL) to define the RDF model. Through that model the user can address RDF entities like resources, classes and properties programmatically, without queries. ActiveRDF is store independent, adapters to multiple stores exist. If a adapter to a specific store does not exist it could be written easily.

Topaz

Topaz[23] is a object to RDF persistence library loosely based on ORM last access 01.03.10. Topaz supports multiple different triple stores like SESAME or mulgara. Topaz allows us to use the underlying store's query language and/or to use its own language (OQL[24]).

Agogo

Agogo [24] is an approach in automated generation of ontology APIs aware of the complex mappings coming with APIs for upper-level or core ontologies. The approach defines a Domain Specific Language (DSL), agogo, to encode the structure of the intended API in a platform independent manner. The agogo DSL is aligned to the SPARQL syntax and it is possible to generate agogo code directly from an ontology definition in OWL. From this DSL it is possible to generate APIs in arbitrary programming languages.

[18]The JenaBean project website `http://code.google.com/p/jenabean/`

[19]The RDFReactor project website `http://semanticweb.org/wiki/RDFReactor` last visit 01.03.10

[20]For Java Proxies see `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`

[21]The ActiveRDF project website `http://www.activerdf.org/` last visit 01.03.10

[22]Object-relational mapping `http://www.agiledata.com/essays/mappingObjects.html` last visit 01.04.10

[23]The Topaz project website `http://www.topazproject.org` last visit 01.03.10

[24]Object Query Language `http://www.topazproject.org/trac/wiki/Topaz/Manual/Section11`

2.7.1 Summary

Investigating all these different ontology API frameworks and RDF persistence layers, we could say that in this field multiple strategies have been developed to receive persistence toward an ontology instance model. Many of the presented frameworks use common RDB persistance patterns and modify the functionality towards triple based stores. We can find frameworks using typical java bean conventions[25], others are aligned to the popular ActiveRecord[26] concept in Ruby. Most of them seem to be tailored to a specific store API like Jena or Sesame. In most cases that does not affect the compatibility to other stores because most of the store implementations are able to integrate foreign stores. What most of these API generation and/or store concepts have in common is the fact that they perform an simple concept-to-class mapping and as an result an object-to-statement mapping in object persistence. Thereby inter object relationships are realized implicite in the active object structure. Only Winter provides explicit relationship declaration through pattern classes, and can separates the instance relationships from the sematic relationships. This new feature of WINTER gives as a powerful tool in implementing a validity preserving CRUD operations for object persistence towards RDF. We discuss this in the following chapters.

[25]Summary of the java bean conventions `http://en.wikipedia.org/wiki/JavaBean#JavaBean_conventions`

[26]Ruby ActiveRecord at RubyForge `http://rubyforge.org/projects/activerecord/`

3 Object Persistence and CRUD Operations on Triple Stores

Generating valid and complete object persistence always requires validity and completeness preserving CRUD, C(reate) R(ead) U(pate) D(elete), operations. Object persistence for ontologies means that software representations of individuals (instances), are created and serialized or read and deserialized to/from an triple store. These instances and the corresponding serialization should be modifiable and erasable. Apart from a valid and complete structure in the ontology API, the most important thing is that the knowledge in the serialization is valid according to the ontology. All operations performed on instances of the ontology API affect the corresponding statements in the triple store. The operations performed on the triple store could be reduced to the CRUD operation set. In our discussion about CRUD, we have to have both in mind, the software based representation of the ontology API and the conceptual based knowledge in the triple store. Only a valid and complete API representation leads to a valid and complete serialization and vice versa. In this section we discuss the influence of the special characteristics of conceptual knowledge in triple stores on CRUD operations. We define how an ontology API should be structured and how it should behave to support CRUD that leads to a valid and complete serialization.
According to our target, an API working on a ontology based dataset, we must denote that the problems described below are situated in a closed world software environment. And so, we must care about the completeness of the data set while performing CRUD operations. In contrast to the logic driven world of ontologies, where undefined individuals exist, we just do not know them. In the closed object-oriented world, instances that are not declared can not be referenced and do not exist. When working on semantic web data such completeness is often not fulfilled and this could lead to problems an API working on them must deal with. All problems described here originate from the concept declaration and the relationship structure of ontologies. But all of them are of fundamental concern to the operations performed sets of individuals and must be discussed on individual level. We use our running example defined in Section 1.1 to clarify our observations.
In the first Section 3.1, we examine the scope of ontology concepts and individuals. We show that it is possible, especially in ODP based ontologies, to distinguish concepts and their individuals according to the part of the ontology they are related with. In the second Section 3.2, we discuss the underlying structures of ontologies and their influence on CRUD operations. Section 3.3 deals with the semantic of concepts and the concept constructors they based on. We discuss the necessity to represent complex concepts

constructors in ontology APIs and their influence on CRUD behavior. In this discussion we take the observations made in Section 3.1 and 3.2 into account.

3.1 Global and local Scope of Concepts and Individuals

Two different ODPs in a pattern-based ontology could declare the same concept as part of the pattern. Configurations of different ODPs or two different configurations of one and the same ODP could refer to the same individual, whereas configuration means a individual set fully allocated a pattern. We can see this in our running example, it represents a pattern configuration for each of the three involved patterns, the `AnnotationPattern`, the `DecompositionPattern` and the `DataValuePattern`. All of them are declaring a concept `InformationObject` to be part of the pattern configuration. And as we can see in the configuration defined by the example, all three patterns referring to the `image-2`. Regarding concepts in ODPs, taking this behavior into account, let us distinguish them into two different sets. The ones referred by multiple patterns and the ones only referred in the declaring pattern. This observations could also be made on individual level, with the difference that a single individual could belong to single configuration or multiple configurations of one or more patterns. This leads us to distinguish concepts into two sets:

Global Scope Concepts: In a pattern-based ontology we can identify concepts referenced in multiple pattern specifications. A concrete individual of such a concept could probably be referenced by pattern configurations of multiple patterns or multiple configurations of one and the same pattern. We call such a behavior global scope, a concept with global scope could be used in multiple pattern declarations and a individual of such a concept could play a role in multiple configuration of the same pattern. For example the `InformationObject` in the M3O is used in multiple patterns and a individual of this concept can play a role in, e.g., multiple `AnnotationPatterns` (an image with multiple annotations).

Local Scope Concepts: In contrast to the concepts with global scope, such concepts are only referenced in one pattern of a pattern-based ontology. On individual level, individuals of such concepts have only relationships in a single pattern configuration. For example the `Description` concepts in the different patterns are only of relevance in the concrete pattern and each configuration of such pattern refers to its own `Description` individual.

According to this the API representations of concepts could also be distinguished. This makes sense because the concrete scope of an individual(instance) has influence on how CRUD should be performed to preserve validity and completeness of the dataset. For example individuals with local scope are only of concern in the declaring pattern but unfulfilled mandatory dependencies to individuals with local scope hinders us to instantiate the whole pattern representations. In contrast to this, the existence of an individual from an concept with global scope is not depending on any relationships declared inside of a pattern referencing such a concept. Concepts with global scope come with their own dependency structure (their properties). Otherwise the existence of a whole pattern could depend on the existence of a individual with global scope. In an API representation we would distinguish like this:

Global Scope Objects: An object representing an individual with local scope exists once and is referred by every pattern instance using it. Global scope objects could exists without being referenced in a pattern. For example an image instance could exist without being annotated or decomposed.

Local Scope Objects: An object with local scope exists only inside a specific pattern instance. Without the pattern object the local scope object is useless and without the local scope object we could not instantiate the pattern object. For example a `AnnotationDescribtion` instance makes only sense in a `AnnotationPattern`, otherwise the `AnnotationPattern` is incomplete and thus useless without the corresponding `AnnotationDescription` instance.

In our example, we shown some individuals that have relationships in configurations of multiple patterns and some individuals only having relationships inside of one pattern configuration. Individuals like the `image-1` or the `geo-location-1` with relationships in multiple patterns are individuals from concepts with a global scope. Additionally some other concepts used in our example have a global scope, although their individuals have only relationships in one of the pattern configurations. The second image, `image-1` has also global scope because it is individual from the `Image` and this concept has global scope. If we look at Figure 1.1 all individuals with global scope have a black frame.
These observations could be generalized regarding D&S based patterns. The concepts with local scope are often the ones defining the contextualized view provided by this pattern. With a few exceptions, context independent classes will always have global scope. In Figure 3.1 the basic D&S pattern with a possible class separation is shown. We must denote that in other applications of the foundational patterns, it is possible that concepts now denoted as of local scope, become concepts with global scope. For example in the second introduced ontology, the Eventmodel-F, the `InterpretationPattern` uses the `Situation` subconcepts of the different patterns. In Section A.2.3 of the Appendix, you find a detail discussion of the interpretation pattern, Figure A.9 in the appendix shows the interpretation pattern. In the interpretation pattern the situations are used to refer to the pattern instance relevant in the particular `interpretation`. So if we want to refer to the contextualized view declared by such a pattern from outside, it makes sense to declare a `Situation` concepts with a global scope. This is shown in Figure 3.2. Such variations in concept scope can come from the different pattern applications or with different ontologies. So the scope of a particular concept in a given ODP could not be reliable derived only from the ontology itself. What we can do is analyzing the concepts of all patterns. Concepts used in multiple patterns could be considered to be of global scope. This approach fits for the example of the two different D&S based patterns presented above and in the Figures 3.1 and 3.2. But there are multiple cases, the scope of an given class depends on a deeper knowledge of the intended application and/or the domain of the underlying ontology.

As we can see, it is possible to distinguish concepts/individuals according to their scope in the ontology, but such separation depends on several factors. With separating the individuals into the two proposed sets, we could predict the complexity for CRUD. And later, in Chapter 5, this information helps us designing our ontology API.

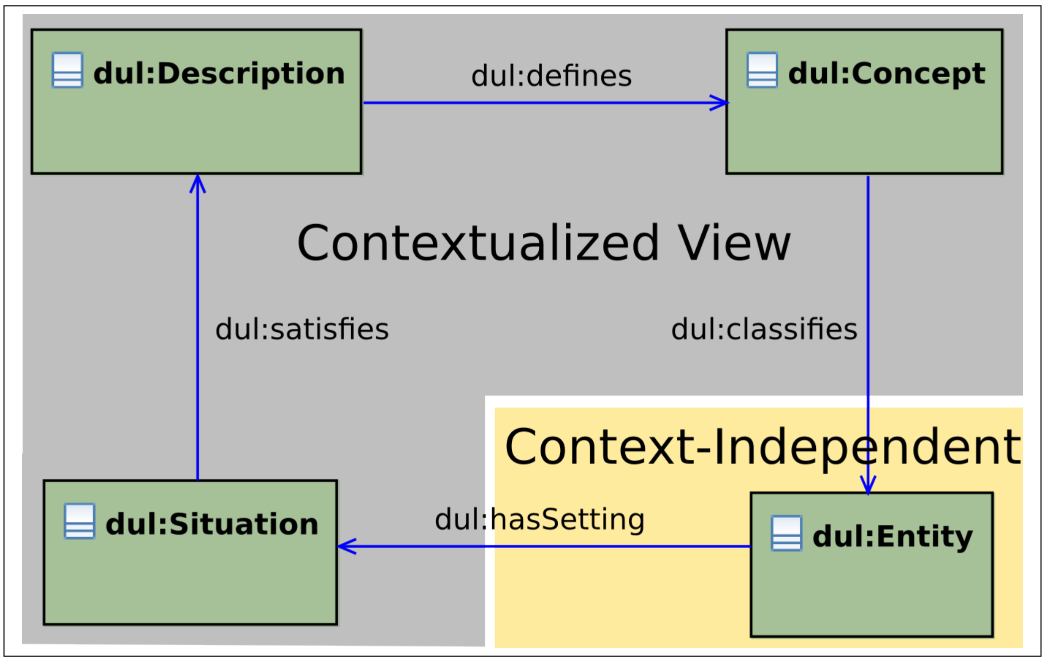


Figure 3.1: Concept scope in D&S

3.2 Ontology and ODP Structures

Considering the underlying structure of ontologies as a set of nodes, where pairs of nodes are connected by links, leads us to an graph theoretical approach to ontology structures. Based on this we can distinguish between different structures found in ontologies or ODPs. Basic ontology and ODP structures could be distinguished according to the purpose of the structure or substructure and according to the conditions the structure satisfies. Based on the purpose we distinguish between inter-concept relationship structures and simple declarative structures of single concepts. The two relevant basic conditional structures we could distinguish are graph structures and tree like structures. Tree structures could be distinguished according to the number of parent nodes for one child into trees and rooted trees[1]. In rooted trees, only single parental objects are allowed in trees in general multiple. By contrast to the tree structures, graphs are not strictly directed and we could find undirected or bidirectional links and self references in the inner structure. As a result of this graphs could contain cycles. In figure 3.3 you

[1]Explanation of rooted trees `http://mathworld.wolfram.com/RootedTree.html`

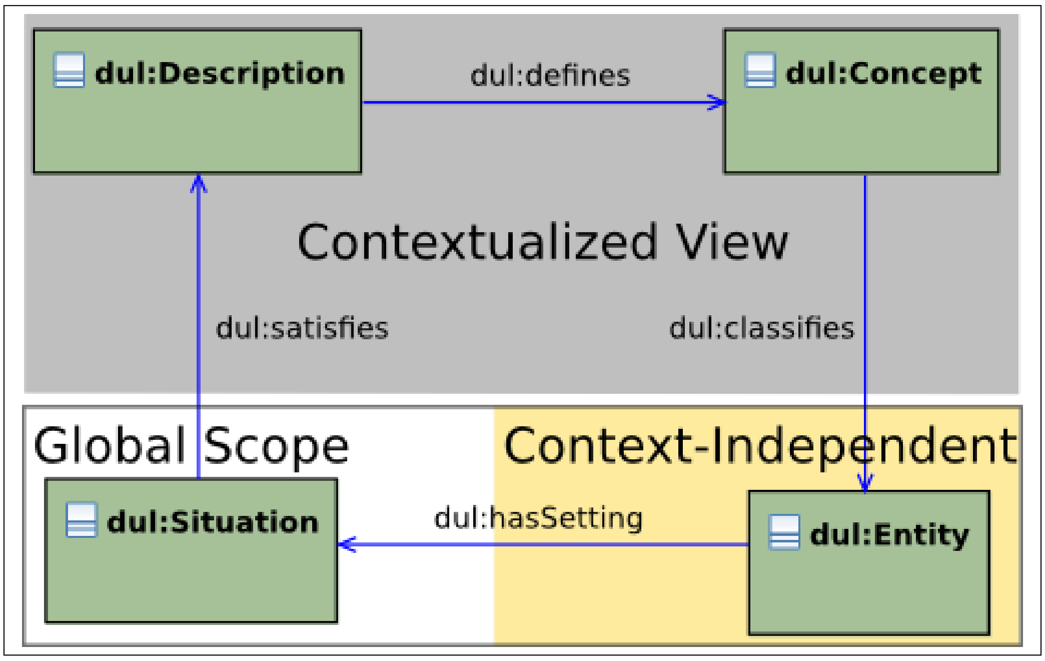


Figure 3.2: Concept scope in Eventmodel-F D&S application

see simple examples of the three different structures that could be found in ontology based datasets. In the rooted tree structure, every node expect the root has exactly one parent and as a tree there exists only one path between two particular nodes. In the tree the limitation to one single parental node was removed, so that nodes can have multiple parents. The last example shows us a graph, in this very general structure most of the limitations have been removed. In graphs nodes can have multiple parents, there can be more than one path between two nodes, self references are allowed, undirected or bidirectional links are allowed. As a result of general nature of graphs cycles in the structure are possible.
On individual level, in datasets based on ontologies, the individual structure is always based on the concept structure of the particular ontology. When we look at a single configuration of an ODP, the individual structures is not dissimilar to the concept structure of the ODP. In this case we do not have to care about cross references between different instance structures. Regarding multiple configurations of ODPs or ODP families referring to joint individuals, the individual-structure probably expands the concept structures. What is rooted tree on the level of a single ODP configuration could result in a graph structure, regarding the whole ODP family or multiple configurations. That could force us to tread a substructure in the manner of the referencing superstructure. But due to the hierarchical order in the graph theory, based on the continually limitation of conditions the structure has to satisfy. We could show that an solution working for a graph will always work for a tree or a rooted tree.

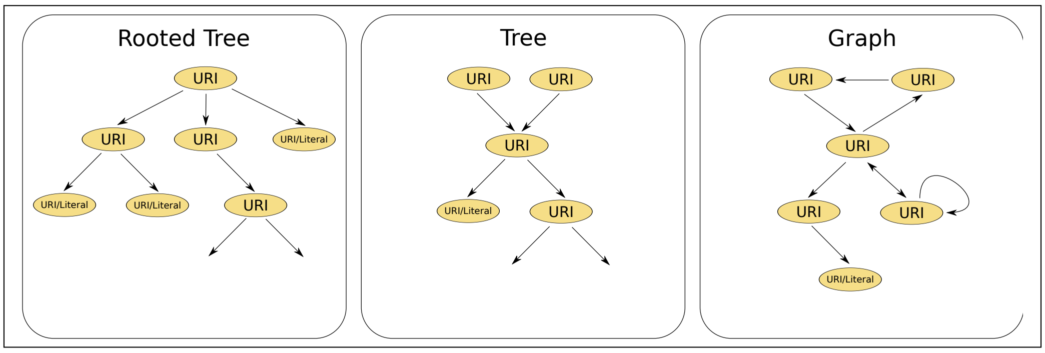


Figure 3.3: Different ontology structures

3.2.1 CRUD Operations on Individuals

Here we will examine the influence of the structure of ontology based data sets on the different CRUD operations. We will focus in our discussion on graph like structures and discuss in detail the particularities of such structures and their influence on CRUD. The other two structures, tree and rooted tree, are just limited graphs. Therefor solutions developed for graphs could also be applied to the simpler tree structures. In our examination on structures we take the observations made in the previous sections into account. CRUD operations always affect both sides the ontology API primitives and the serialization in the triple store. Due to the closeness of the operations to ontology API objects we use the object-oriented terms class and instance. All relationships and properties regarded here are of mandatory nature for the existence of the class or pattern. Optional dependencies have no direct influence on the existence of an instances. We point out the problems arising from the different CRUD operations and examine mechanisms to solve them.

Create: Creation in object persistence always involves two steps, the creation of instances from classes and the serialization of those to the store. Regarding a single pattern instantiation; generating a valid and complete serialization depends on the instances representing a valid and complete configuration of the ontology pattern. When regarding multiple instances or multiple cross referencing pattern instantiations, we must ensure completeness and validity for the whole structure. We recognize that a instantiation of a particular class in the dependency structure depends only on the declarative dependencies of the class and the instantiation of the classes connected by the outgoing links. Regarding dependencies as a hierarchical tree, creation is a bottom up process. First we have to instantiate from the classes on the lowest level and only then we are able to instantiate from classes on a higher level.
Ensuring that the created structure is valid and complete could be performed in different ways. We can use predicated create operations, that ensure completeness on creation by passing all mandatory relationships to the create operation. Such a create operation expects all instances, mandatory to the instance to cre-

ate, to be delivered as parameter in the create operation call. This is synonymous with checking all outgoing links and that is one of the biggest disadvantage of predicated create operations. If we look at your example, we can easily identify the inner loops over e.g `AnnotationDescription`, `AnnotatedIEConcept`, `InformationEntity` and `AnnotationDescription`. The loops in our example come with the specialization of the D&S pattern. For creating an instance in a loop like structure, predicated create methods do not work. We run into an infinite dependency loop. A instance of `AnnotationDiscription` depends on a `AnnotatedConcept` instance, this depends on a `InformationEntity` instance, this on a instance of `AnnotationSituation` and this again depends on the `AnnotationDiscription` instance. Figure 3.4 shows the loop dependency. If we take a look at the Annotation pattern in our running example we can also observe this problem. Every instance defined there has a direct dependency to another instance. Following the loop described above we recognize the cyclic dependency structure in the instances of our example. As shown we are not able to instantiate from classes in loop like structures with predicated create operations. Because in case of simple concept-to-object mapping in combination with cyclic structures predicated create methods do not work. We need another solution to preserve validity and completeness.
A possible solution is an independent completeness preserving process after creation. Such a process should analyze the whole created structure before serialization. This includes instance and `property` dependency checking. Starting from the concept representation to create such method has to check recursively if all outgoing mandatory dependencies are fulfilled. Methods for dependency checks are also influenced by the underlying structure. Performing such checks on loops forces us to a define dedicated starting point (the concept representation to create) to ensure a proper ending of the method. From this starting point the method has to check all mandatory dependencies and remember already checked representations. Serializations should only be possible from a valid and complete structure. To ensure this the user of the API has to implement proper create method to prevent or useful feedback to solve occurring incompleteness or validity.problems. All this applies for instantiation of whole patterns and in there especially for classes representing concepts with a local scope. Instantiation from classes representing concepts with a global scope should be inter-concept relationship independent. Only the declarative dependencies have to be solved when instantiating from a class representing such a concept.

Read: The main source of problems in read operations is an invalid or incomplete dataset to deserialize from. When working on complete data sets the read operation is quite straight forward. We can deserialize all we want. To ensure that the instances represent a valid state, we only have to deserialize recursively all mandatory instances and their declarations from the store. Like in the create operation we should be able to deserialize instances representing a concept with global scope independently from the inter-concept relationships. We only have to take the concept`'s` own declarative structure into account. For deserializing instances from

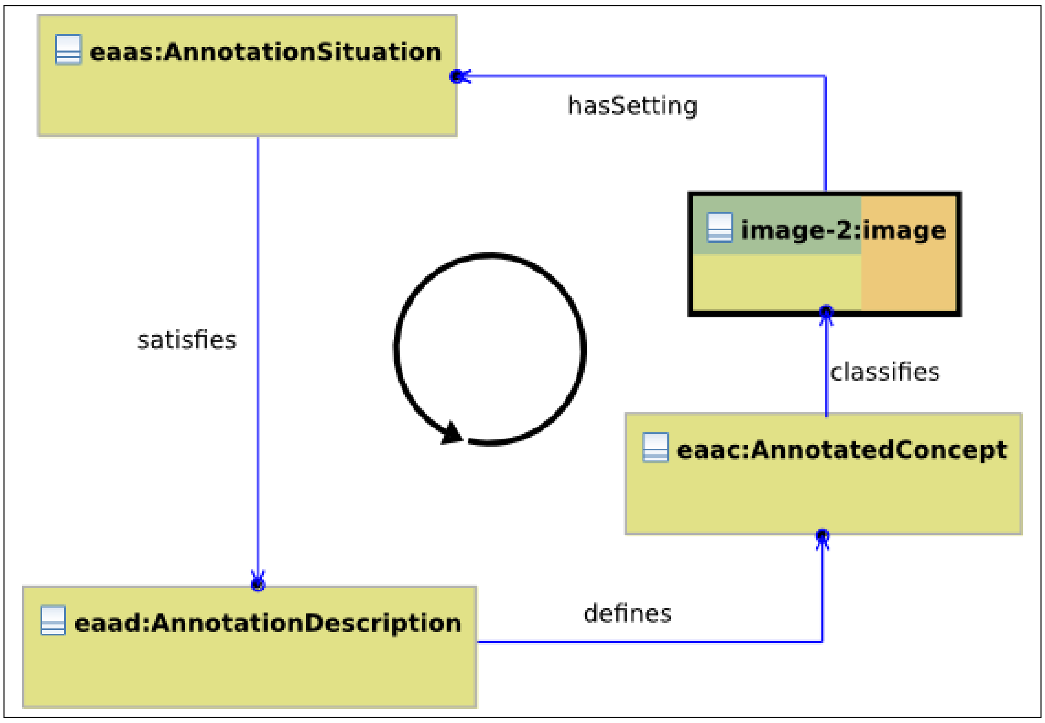


Figure 3.4: A D&S Loop from the Example

a serialization representing concepts with local scope we have to instantiate the whole pattern. Following all mandatory outgoing links defined in the concrete pattern recursively. For our running example this means, we can deserialize an object representing the `image-1` at any time without being forced to deserialize anything else. But deserializing ,e.g., the decomposition description (`eddd`) makes only sense when deserializing the whole Decomposition Pattern. For that concern, we have to deserialize objects for all individuals defines in this pattern. In case of the running example this would be the presentation, the two images, the classifying concepts, situation and describtion.
If we want to be able to work on data from arbitrary sources we have to be able to deal with incomplete data sets. To preserve completeness and validity of the `instances`, it is strongly recommended to develop strategies to handle deserialization from incomplete data sets. Uncontrolled deserialization from incomplete data sets always ends in an incomplete set of `instances`. That could constrains us to further reduce our result set until we have a set of nodes without any missing outgoing link. That means we have to reduce until we can fulfill every dependency. We are not able to deserialize a pattern instance from an incomplete serialization. Classes representing concepts with local scope within such a incomplete pattern, could be deserialized. But because they are only of concern in the concrete pattern context and the whole pattern could not be deserialized it makes not sense to instantiate local classes from such incomplete serialization. Classes represent-

ing concepts with global scope could be deserialized pattern independent but we must check the pattern concepts declarative structure. So performing read on incomplete data could lead to four different cases.

- First a serialization of an instance with local scope lacks while deserializing a pattern. In this case we are not able to deserialize the pattern. We only could deserialize instances with global scope. Same example as above but we want to deserialize the annotation. In this case we are not able to deserialize the pattern.
- In the second case a serialization of an instance with global scope lacks. In this case we are not able to deserialize any pattern this instance is involved in. Like in the case above we could deserialize all the other instances with global scope and patterns this instance is not included in. If in the serialization of our example the central instance, the `image-2:image` is missing non of the patterns could be deserialized.
- In the third case the serialization of the declarative structure of an instance with global scope is incomplete. If the incompleteness concerning a mandatory property, this is similar to missing the whole instance in the serialization. Consindering the `geo:long` and `geo:lat` relationships to be mandatory for the `GeoPoint` concept. Now missing one of the `xsd:decimals` literals in the serialization, hinders us in deserializing the `geo-location-1:GeoPoint`. Thus we are not able to deserialize annotation and datavalue.

Update: In the update operation we have to modify the instances and according to this modify/replace the statements in the serialization. Depending on the scope of the operation we have to modify a single field in an instance, replace a whole instance or modify/replace multiple instances. According to this we have to modify/replace the statements representing the single value, the instances or the instances in the serialization. The manipulation of the instances is trivial, the interesting part is to mirror this manipulation to the triple store. So our discussion in this part revers much to the serialization and the single statements in it. Referring to the graph representations build up by the statements, we can distinguish possible changes according to the nodes involved. Changing literals or leaf URIs, is trivial. Thereby leaf URIs are URIs that occure just as object in all statements. We just find the right statement and change the object URI/Literal in it. For example changing the `geo:long` in the `geo-location-1` just involves one statement.

```
geo-location-1  <geo:long>        "40,76"^^xsd:decimal.
```

To change the longitude we have to replace this statement through a new one with the new longitude like:

```
geo-location-1  <geo:long>        "50,68"^^xsd:decimal.
```

If we want to change an inner node of the graph, a URI also present as subject in some statements of the structure, we have to deal with the slightly different

nature of triple stores. Such an inner node is subject in the statements in which it declares its properties and relationships. Additionally this node is also used as object if some other node refers to it. To perform an update operation we have to find all statements where this particular node references or is referenced. We have to replace each of those statement through a new one. So updating leaf nodes that could only occur as objects is easy, but dealing with the inner nodes always leads to replace multiple statements in the graph structure. Below we will examine update(replace) operations on our example, that affects an inner nodes.
Replacing the `GeoPoint` instance `geo-location-1` through another instance of `GeoPoint`, e.g.`geo-location-2` affects multiple statements in the serialization of our example. First of all the particular instance and its statements in the serialization changes. And with it references to the URI of `geo-location-1` should change to the one of `geo-location-2`. This affects all statements representing incoming links, like `classifies` and `hasRegion`. In such links the URI has the object role. In all statements representing outgoing links, like `hasSetting`, `geo:long` and `geo:lat` the URI represents the subject. If such a link represents a property declaration, like `geo:long` and `geo:lat`, it is obvious that the Literals change with the new `GeoPoint`. In this case the whole statement should be replaced through the new ones defined by `geo-location-2`.
In the listings 3.1 and 3.2 we present the statement structure of the `AnnotationPattern` in our example, before and after the update operation described here. As you can see the statements affected are the type declaration in line 5 and the ones in from line 11 to 14. Those define the concrete pattern instance. There all appearance of `geo-location-1` changes to `geo-location-2`. And as mentioned above the values for longitude and latitude also changes. If optional properties or relationships are present in the serialization, the update also affects all statements representing such relations.

Listing 3.1: `AnnotationPattern` statements before update

```
1  eaad              <rdfType>          AnnotationDescription.
2  eaac              <rdfType>          AnnotatedConcept.
3  image-2           <rdfType>          Image.
4  glp-1             <rdfType>          EXIFGeoParameter.

5  geo-location-1    <rdfType>          GeoPoint.

6  eaas              <rdfType>          AnnotationSituation.

7  eaas              <dul:defines>      eaac.
8  eaad              <dul:defines>      glp-1.
9  eaac              <dul:classifies>        image-2.

11 glp-1             <dul:classifies>        geo-location-1.
12 geo-location-1    <geo:long>         "40,76"^^xsd:decimal.
13 geo-location-1    <geo:lat>          "-73,99"^^xsd:decimal.
```

```
geo-location-1  <hasSetting>     eaas.

image-2         <hasSetting>     eaas.
eaas            <satisfies>      eaad.
```

Listing 3.2: `AnnotationPattern` statements after update

```
eaad            <rdfType>        AnnotationDescription.
eaac            <rdfType>        AnnotatedConcept.
image-2         <rdfType>        Image.
glp-1           <rdfType>        EXIFGeoParameter.

geo-location-2  <rdfType>        GeoPoint.

eaas            <rdfType>        AnnotationSituation.

eaas            <dul:defines>    eaac.
eaad            <dul:defines>    glp-1.
eaac            <dul:classifies>          image-2.

glp-1           <dul:classifies>          geo-location-2.
geo-location-2  <geo:long>       "55,92"^^xsd:decimal.
geo-location-2  <geo:lat>        "-24,68"^^xsd:decimal.
geo-location-2  <hasSetting>     eaas.

image-2         <hasSetting>     eaas.
eaas            <satisfies>      eaad.
```

Delete: The delete operations is, if wrongly performed, the operation with most side-effects on the validity and completeness of serialization. Deleting an instance and its serialization with completeness preservation, always means to delete recursively all instances depending on the existence of the deleted instance. We will discuss the deletion of mandatory inner nodes and additionally the deletion of mandatory leaf nodes. It should be clear that such mandatory relationships could be directed or indirected. That means that both parent and child could mutually depend on each other. Regarding delete operations, we have to ensure that performing this operations does not violate the existential constraint through the modification performed on the instances and their serialization. We should distinguish between delete operations performed on whole patterns and on instances. In a pattern based ontology delete operations often affects a whole pattern, e.g., if we want to delete the geopoint annotation from our running example it makes sense to delete the whole AnnotationPattern and DataValuePattern instate of just deleting the annotation instance. Regarding the class scope recommends us to distinguish ones more between instances with local and those with global scope. When deleteing the AnnotationPattern/DataValuePattern instance like proposed above only the

classes of local scope should be affected and not the image itself also member of the two pattern instances.
The simplest case is to delete an instance with no mandatory relationship to any other instance. Such an operation could not violate any constrains and the scope of the object is irrelevant, thus the operation could be performed unchecked. Deleting an instance with local scope, mandatory to the declaring pattern forces us to delete the whole pattern. For example if we delete the `eaas:AnnotationSituation` in our running example. The whole Annotation pattern becomes incomplete and thus has to be deleted also. The instances with global scope have a special role in this case because their existence does not depend on the pattern. This leads us to the deletion of whole patterns.For deleting a pattern all the local instances have to be deleted. Such an operation has no direct influence on instances with global scope referenced by this pattern but we have to decide how to handle a global instance if no pattern refers to it. Should it be kept or erased? This behavior should be made controllable. And to provide a functionality dealing with such behavior we have to count references for global instances. Last we consider the deletion of instances with global scope. Deleting instances with global scope could affect multiple patterns. If we delete the `image-2` from our example, we have to delete the the whole `AnnotationPattern` and DataValuePattern and all other patterns referring to `image-2`. There is nothing to annotate anymore. The decomposition pattern is affected in parts, the `ComponentRole` defining the concept of `image-2` is no longer needed. The pattern itself still defines a decomposition and so should only be touched in the parts depending on the existence of `image-2`.

3.3 Ontology Semantics and their influence on CRUD

Performing CRUD operations on ontology based data set, is not only affected by the structural particularities of ontologies. The semantics of ontologies also leads to subsequently peculiarities of CRUD. In this section we observe what kind of concepts representation could be useful on API side to provide a basis for valid CRUD. The concrete representation structure strongly depends on the concepts defined in an ontology and thus on the constructors used in the definition. We analyze concept constructors introduced in 2.5.1 and complex concepts resulting from constructor combinations. In many cases constructors or combinations of them could lead to additional problems regarding the CRUD operations. We present some of these cases in detail and discuss how to implement CRUD that can handle such situations.
Generally we constrain that only named concepts are allowed for concept that should be represented in the API and more general in ontologies of which an API should be build. In complex concept definition we only allow named concepts as role filler. We support expressions in the form, e.g., $A \equiv B \sqcup C$ and as role filler we support $A \sqsubseteq \exists R.B$ and $A \sqsubseteq \forall R.B$
Ontologies not fulfilling these limitation could easily be transformed. For example a complex concept A defined by a constructor combination $A \equiv \exists R.(\forall P.(C \cup D))$, could

easily be decomposed to $A \equiv \exists R.X$, $X \equiv \forall P.Y$ and $Y \equiv (C \sqcup D)$. As you can see we named all anonymous concepts, $(C \cup D)$, $(\forall P.(C \cup D))$ and $\exists R.(\forall P.(C \cup D))$. With naming all the anonymous concepts we build up a chain of named concepts. Approximating a API representation for A means to iterate over this concept chain and combine the mapping instructions, if they exist, for the single constructors to a mapping for the whole chain. Often it might be necessary when iterating over such constructor chains to stop the iteration at a certain point. But we have to choose such a point carefully, so that we are still able to build an API representation of the intended concept in a way, that the out-coming serialization is still valid regarding to the declaring ontology.

3.3.1 Named Concept Representations

Generally we can denote that for every named concept, there should be some kind of representation in the ontology API Model (OAM). Further discussion will be needed on how those representations should look like. Regarding rudimentary named concepts without any properties, we can denote that an individual to such a concept always serialize to a RDF triple in the form:

```
IndividualIDURI <rdf:type>        ConceptIDURI.
```

Most of the ontology API frameworks mentioned in Section **??** map their instances (objects) in this way. These two URIs are the basic information needed in every API representation of any concept to be able to serialize it. Based on this observation we can distinguish between two different abstractions for concepts in our API. We can define a class for a named concept holding the two URIs or we can just represent a individual through two URIs without a wrapping class. In Section **??** we talked about the two different sets of concepts, global and local. In combination with the observations made here we denote that we represent local scope instances through two URIs an identifier URI, unique to the single instance and an type URI unique for the concept. Otherwise sp with global scope, who often have a property structure carrying additional information of the particular instances, will become a full featured classes representation.

3.3.2 Concept Constructor Representations

Ontologies do not only consist of Atomic or named concepts, often constructors are used to define the ontologies semantic. In DL we define complex concepts with concept and `role` constructors in arbitrary combination. Regarding such complex concepts, it is not enough to examine single CRUD operations performed on an class representation of a complex concept. Sequential combinations of CRUD operations on instances of such classes could lead to side effects and thus problems in further CRUD operatons. In this subsection we will discuss DL constructors and their influence on CRUD. Additionaly, we will analyze the side effects that could occur when performing CRUD on complex concept representations. We provide basic patterns to generate API representations for concepts.

1. **Equivalence:** The equivalent ($\equiv$) to a concept A is a concept B, where the extension of B is exactly the same as the extension of A. The axiom denoting equivalence between the two concepts allows us to interfere that all individuals of type A are also of type B and vice versa. In the knowledge base, especially the A-box, this information is normally implicit. If we only have one individual x, e.g. $x : A$, the information $x : B$ could be interfered. Through the mapping to an API and the performance of CRUD operations this implicit interferable information becomes explicit. Regarding the following example that becomes obvious: In the T-box we have two concepts A and B and an axiom denoting them to be equivalent $A \equiv B$. In the corresponding API environment we will have two classes, `classA` and `classB`, one for each of the concepts. In the A-box, in our case the serialization, we have a single individual of type A, $x : A$. If we perform a read operation on this serialization, like give me all instances of type B - `getAllObjectsOfType(classB)` - we expect, according to the equivalence, to get back an instance of `classB` for the A-box individual $x : A$ (the serialization instance `x:classB`). So after the read (deserialization) we will have an instance `x:classB`. If we now write back (serialize) our current instance state, we will make the implicit information, that all individuals of type A are also of type B, for x explicite. We add $x : B$ by serializing `x:classB`. This new information could lead to different problems performing other CRUD operations on the knowledge base. For example deleting our instance `x:classB` we could not only delete the corresponding individual $x : B$, we maybe have to delete $x : A$ from the serialization. Otherwise reading once again - `getAllObjectsOfType(classB)` - will return x as instance again.
   This observation makes the implementation of valid and complete CRUD more complicated as expected. To solve the problems arising from the use of the equivalence constructor we have to use reasoning services to identify all possible serializations of and instance when deleting. Such a reasoning task has to identify all relevant individuals, so that we can perform the CRUD operation on all corresponding serializations.

2. **Union:** The concrete creation of an instance of a class representation of an concept defined using union depends strongly, on the mapping of the union constructor. Independent from the concrete mapping we could observe the intended serialization result from an `union-class` instance. In many cases such a `union` class is of no interest in an ontology API. It is rather useful in reasoning performed on the dataset or the ontology itself. A class representation is of interest in an ontology API if it should be possible to perform a cast to the union class. This could happen if other classes in the ontology API refer to the union class. If an API is mapping the concepts to corresponding class representations, we could simulate the union through a common supertype. This approximates the semantic of the union in a object-oriented environment. In all cases, the serialization result is essential, whether if an API representation of the `union` concept is needed or not. For example if we start with two concepts $father$ and $mother$, we could define a new concept $parent$ based on the other concepts as follows: $parents \equiv father \sqcup mother$.

Regarding this example of a union, an API approximation should allow us to cast a instance of an class representing $father$ or *mother* to the type of the class representing *parent*. This makes sense in environments where it should be possible to declare or reference instances of classes who are combined to define such a `union-class`, as a members of the union-class. If such a behavior is not needed we do not need a class representation for the union. Similar to the equivalence constructor and due to the fact that in the definition of named concepts based on an union always an equivalence constructor is involved too, e.g., $parent \equiv father \sqcup mother$ we could run into additional problems with use of `union classes` in our API. We would be able to instanciate a instance of the `union class`, e.g., *parent* from serializations made from instance of $father$ and *mother*. Despite from the problems arising with the use of the union constructor in the declaration, the abstraction of an union to a common superclass could produce its own side effects regarding CRUD. For example a $father$ class referenced from a *parent* field has to be serialized to a $father$ individual but we have to keep the *parent* nature in mind for deserialization.

3. **Intersection:** In intersection like $C \equiv A \sqcap B$ every individual of C is also individual of A and B. For example a concept *women* could be defined as an intersection of the concepts *person* and $female$, $women \equiv person \sqcap female$. Like in the case of union it might be useful to introduce a object-oriented representation for *woman*, in our API. By contrast to union a `intersection-class` should be defined in the object-oriented world by a subclass inheriting from all classes to be combined. This would enable us to use an instance of type `intersection-class` everywhere where a type is expected from which is intersected. A new class representing the *woman* concept should implement both, the *person* and $female$ representation.

4. **Complement:** The complement of concept A, $\neg A$ represents a new concept, its extension consists of all extensions not in the extension of concept A. The complement constructor is important in the definition of concepts and their individual sets, but in a object-oriented environment all classes and thus their extensions are disjoint unless stated otherwise. This would make all other classes to be the complement of a particular class. It is possible to abstract this generality in DLs trough common supertypes but this would be very verbose and does not make sense.

5. **Number Restrictions:** Cardinality based on qualified or unqualified number restrictions constructors strongly influences the create process. First of all cardinalities let us distinguish between mandatory and optional dependences on create time. And it denotes if we have to deal with a set of individuals in this relationship or with a single individual. `minCardinality` above 0, always indicates a mandatory dependency, one of 0 indicates an dependency and thus the class property to be optional. These dependencies have to be taken into account in various CRUD operations to preserve completeness and validity of the instances and serialization. A `maxCardinalities` allows us to define the maximum number of individuals a

relation could refer to. This is a indicator how to implement such property in a class representation. In the case of a `maxCardinalities` greater than one, the property represents a set of instances. In the case of one, a `maxCardinality` less then one makes no sense, the property represents a single instance. In a class representation this leads to set property fields or single property fields, Despite of sets one of the most usual mandatory case is a `minmaxCardinality` of one. This represents a mandatory dependency to one other instance of the specified class, a single property field in the implementing class.

6. **Existential quantifier:** An existential quantifier like defined in 2.5.1 provides us with useful information about mandatory dependencies on individual existence. On API side this information could be integrated, like the number restrictions, into the cardinality information of properties. For example we can define *mother* as a *woman* who has a child $mother \equiv woman \sqcap \exists hasChild.person$. If we create a class representation for a concept like *mother* we can create it as subclass of the class representing *women*. On instantiation a *mother* instance has to have at least one child. This could be ensured by the class constructor with only providing constructors for the *mother* class that expect a child. For sure a mother can have more than one child. To model this the child field could become a children field (a set of childs). In this case it is recommended to implement an additional class constructor that expects a set of children.

7. **Value Restriction:** A value restriction $\forall R.C$ defines a universal quantifier, that allows us to define a concept, based on a set of individuals, involved in a particular role definition. For example we could define the concept *women* in another way using value restriction, $women \equiv person \sqcap \forall hasGender.female$. In an object-oriented environment it is not possible to define a class in such a way. But it is possible to restrict class properties in object-orientation, for example we could denote a class women to have a property gender that has to be female by default. This constructor is usually used in the definition of complex concepts, we discuss in 3.3.2. Through the versatile fields of application of this quantifier it is impossible to define a unique mapping instruction for value restrictions. Depending on the concrete intended semantic of the definition, multiple ways of mapping this to an object oriented approximation are possible.

8. **Disjointness:** Disjointness is not one of the concept constructors of DL, but a often used OWL restriction, cause of this we will discuss it here. In DL disjointness is just syntactic sugar and can be reduced, e.g. to subsumption and complement or intersection and equivalence. The fact that the classes A and B are disjoint could be formulated like $A \sqsubseteq \neg B$ or $A \sqcap B \equiv \perp$.
The disjoint to an arbitrary `class` A is a `class` A', whereby the extension of A' is disjoint to the extension of A. In a software environment the semantic of disjointness could be used to make consistency check on the instance representation. For this service the disjointness between classes could be approximated with the introduction of an superclass all disjoint have in common. Such a superclass would

be a construct with no direct effect on the serialization and thus on CRUD. It just ensures in a programmatic way that the extension of the superclass is divided into disjoint extensions, one for each of the disjoint subclasses.

9. **Subsumption:** Subsumption, like $A \sqsubseteq B$ means that all A are also B but not not vice versa. In an object-oriented environment, there the definition of a class on top of sets of individuals is not possible, exists no adequate equivalence mechanism for the subsumption operator.

4 The Model for Ontologies (MoOn)

In this chapter we will discuss and define our Model for Ontologies (MoOn) based on the observations made in the previous chapter. The model itself should base on the UML2 Class Diagram. The MoOn should serve as initial representation of the ontology in the API generation process. This purpose requires additional functionality in the model despite from just representing a logic based ontology. But even for mapping logic based concepts to UML2 various problems must be solved. Because of the object oriented nature some ontological concepts have no direct counterpart in the UML2 Class Diagram. To be able to model ontology semantics in UML2 we need to extened the UML2 Class Diagram meta-model. Several different options are available to extend or restrict UML2 to suit to a particular domain, from featherweight extension through keywords, via light weight extension through profiles to heavy weight extensions customizing the behavior of UML2 or even change the basic concepts used to define the UML2 Class Diagram itself [10]. Most of the customizations possible for UML2 could be assigned to one of the following classes of extensions.

1. Give a terminology that is adapted to a particular domain.
2. Give a syntax for constructs that do not have a notation in UML2.
3. Give a different notation for already existing symbols.
4. Add semantics unspecified in the meta-model.
5. Add constraints that restrict the use of the meta-model.
6. Add information that can be used when transforming a model to another model or code.

The intended customization of the UML2 Class Diagram has to provide models for the constructs of OWL that could not be represented directly in UML2. Different constructors in OWL could not be mapped to single UML2 elements because of the different origin of the two languages families. UML2 is aligned to the object-oriented approach of software engineering. It is closed world and provides a strong hierarchical separation in class definition and instance information. The semantics of DL constructors, shown in Tables 2.1 and 2.2 are defined on sets of interpretations of the used concepts. Complex concepts in OWL are defined on top of interpretation sets of other concepts. This violates the strong separation of class definition and instance informations in UML2. We discussed the different constructors and combinations of those in Chapter 3. For example constructors like complement or disjointness have no similar counterparts in UML2,

so it is necessary to define modeling conventions for them. Such modeling conventions define combinations of UML2 elements (models) to represent OWL constructs with no direct counterpart in the UML Class Diagram. We found such an extension to UML for OWL in the OWL profile and mapping conventions defined in the ODM. For a closer look on the ODM take a look at Section 2.6.2 in the related wok chapter.
Despite from the mapping of OWL additional informations useful for further transformation in our API generation process should also be integrated into the MoOn. This topic is also discussed in this chapter. Based on the observations made in Chapter 3 we analyze what kind of informations useful for transformation should also be integrated the MoOn and how to integrate it. We propose several extension for the UML2 class diagram to integrate the previously identified information.

4.1 Requirements of the MoOn

We can identify two general origins of requirements to our MoOn. Requirements concerning the representation of logical based ontologies in an object-oriented UML Class Diagram and requirements concerning the needs of our model driven API generation process. To be able to map from logic based ODPs to an UML2 Class Diagram we have to make some assumptions regarding the ODP definition and representation to obtain a model we could work on and we need an extension that gives us a similar expressiveness in the MoOn as in logical ontology languages. As denoted we will use the ODM, as OWL extension to UML2 Class Diagram for our model. In the following we will focus on OWL DL as an ontology language and on the OWL extension for UML2 defined in the ODM. As another source for requirements the model driven API generation process forces us to introduce additional extension for the MoOn to be able to serve the needs of it. After the discussion on requirements regarding the mapping, we focus on informations that could be useful supporting the transformation and generation. Parts of such information could be derived from the underlying ontology. Other parts of such information should represent user driven decision regarding the intended implementation and API structure.

4.1.1 Assumptions to ODP Definition and Representation

We already denoted in Chapter 3 that we have to make some assumptions regarding the definition of ODPs. It is difficult to define a UML2 representation and transformations for arbitrary complex ODP definitions. To define proper mappings and transformations we have to make some assumptions regarding the complexity in the representation of the MoOn. Pattern based ontology design helps us, despite from other advantages, to decompose complex ontologies into a set of more or less independent micro ontologies with a characteristic purpose in the domain of the whole pattern family. The first assumption regarding ODPs is that in an ODP based ontology very ODP should come in its own OWL file. This eases the identification of the single patterns. In combinations of role and concept constructors in the ontology, like those described in Section 3.3 we have to make some limitations. Unlimited complexity in the concept definition could not properly be mapped to an UML2 representation. The limitation regarding concept

or role definitions should be that we do not allow anonymous concepts in combined constructors or role fillers. This means that in complex concept or role definitions, inner concepts defined by the use of constructors should be named and these named concepts should be combined in the following. See the example in 3.3. Nearly every definition that uses anonymous concepts could be brought in a form without anonymous concepts, so this does not limit the expressiveness. Otherwise decomposing ontologies in this way leads to an introduction of multiple new named concepts and we have to decide which of the named concepts should have a direct representation in the API. We do not need API representations for all of the newly introduced named concepts. To be able to control this we need information in the model to control the transformation and separate concepts with API representation from those without. Such information should also be placed in the MoOn. In the following we discuss the different information that should be integrated in the MoOn. Where such information came from and how to integrate it into the MoOn.
Additional assumptions could be made regarding the degree of axiomatization made in the ontology and ODP definitions. Often ODPs are not strong axiomatized to retain their generality, refer to [29]. But especially cardinality informations about properties might be very useful in an API generation process. They lead to cleaner less complex classes. We propose to use stronger axiomatized ODPs in an API generation process We also support lesser axiomatized ODPs but it requires more user decisions.

4.1.2 Requirements from OWL

Most of the requirements concerning the presentableness of logic based ontologies in UML2 are covered by the ODM introduced in Section 2.6.2. But some of the mapping conventions made in the current version of the ODM, like those discussed in the introduction of the ODM does not fit for our transformation process. The modeling convention made for `owl:intersectionOf`, mentioned in the related work section, is a good example for that. Denoting OWL nature of the UML constructs through the use of `UML:Constraints`, like in case of `owl:intersectionOf`, could lead to problems because multiple editors do not allow constraints on generalizations. Additional as mentioned, the modeling convention for `owl:intersectionOf` misses some of the specification. So we decided to use the upcoming conventions for such constructs. Namely these are `owl:intersectionOf` and `owl:union`. You can find Figures showning the old an the new mapping conventions of these and other constructors in the Appendix A.3.1.
Our MoOn bases on UML2 and the ODM OWL profile, modified regarding some OWL constructors like mentioned above.

4.1.3 Additional Requirements

The MoOn serves as starting point for the following transformation and the final generation of JAVA code. The model must represent the ontology and display all information needed for the following transformation tasks. Such information can have two different origins, first it can come with the ontology, so we must be able to identify this

informations in the model. We have to prevent that we loose useful information when representing the ontology in UML2.
Second it represents ontology independent information the user (user means the user of the MoOn) gives. This information can control the structure of the API to be generated, e.g., indicates if a particular concept should have an class representation in the API or not, see Section 4.1.1. For such information we have to provide additional extensions in the definition of the MoOn. Despite from placing it in the model, user driven information could be defined in the transformation process through wizards triggering a user driven transformation. In this subsection we analyze the observations made in Chapter 3, regarding informations that could be useful in our transformations. Of special interest for us are the observations on ontology semantics, concepts scope and constructor combinations made in the Sections 3.1 and 3.3. In the following we list different informations needed in the MoOn. We describe the motivation behind the decision to integrate this information in the MoOn and we analyze their origin.

Concept information: In the MoOn it must be possible to denote for a single concept if an API representation is required or not: As mentioned in Section 4.1.1 we have to provide information about the relevance of concepts for the API generation. Especially regarding the named concepts newly introduced when decomposing the ODP definition according to the procedure described in 3.3. In general we could say that we need class representation for all the named concepts in the regarded ontology. But with decomposing ontologies and introducing named concepts for previously anonymous concepts we need and indicator which were the initial named concepts. Additionally it might be useful to provide class representations for selected newly named concepts in special cases. To derive the initial named concepts is an easy task, we just have to remember all named concepts present before decomposition. But even in this case it could happen that we do not want representations for some of them and we still have to care about the newly introduced named concepts. So signing concepts for class representation in the MoOn is a task that could be automated in parts but in some cases user decisions are needed. Sometimes,e.g., in the case of the different `Situations` used in the `InterpretationPattern` to refer to whole patterns, it could be necessary to take superconcepts into account. In this special case is seams useful to add an general situation concept to the MoOn. From this we could generate a Situation superclass. this class eases the definition of a class representation for the `InterpretationPattern`.

ODP Informations: The MoOn should reflect the pattern based character of the Ontology: The separation of concern coming with ODPs helps us in organizing the functionalities the ontology API should provide. If we refer to our running example, it uses different patterns with different concerns. The `DecompositionPattern` to decompose the presentation into the two images. The `AnnotationPattern` and the `DataValuePattern` to annotate image-2. These patterns could be used as indicators for functionalities the API should provide. For example the `AnnotationPattern` indicates that we want to have an annotation functionality in the generated API. Additionally we can identify the main classes such a task is performed on. In the case of annotation this would be the `InformationEntity` representation to be annotated and the `Entity` representation for the annotation. To preserve this distinct advantage in the UML MoOn, we have to find a way to divide the model representing the whole ontology into different clearly separated parts that represent the single patterns of the ontology. The concept scope information mentioned later is also partly ODP related.

Cardinality Informations: To generate useful APIs we need further information about the concept and their properties. A useful set of informations that should be integrated in the MoOn is the information about the cardinalities of relationships. Cardinality informations in an ontology comes with the axiomatization of the ontology. Through number restrictions and existential quantifiers we limit the domain of properties of an concept. In case of OWL such cardinality information comes with ,e.g., `some`, `owl:maxCardinality` or `owl:minCardinality`. Regarding a single concept and all of its relationships, cardinality makes it possible to distinguish these relationships into mandatory dependencies and optional dependencies. If an relationship is mandatory or optional dependencies has direct impact on the CRUD behavior that has to be used, as we have shown in 3.3. And thus such cardinality information is very important in the definition of the concrete implementation of such behavior as described in 3.2.1. As mentioned, cardinality information in ontologies depends on the concrete degree of axiomatization in the observed ontology or ODP. So in weak axiomatized ontologies such cardinality information might not be present. For a concrete application of such ontologies in APIs it is recommended to the user to add that information to the MoOn.

Concept Scope Informations: The scope of a concept must be specified in the MoOn: As shown in 3.1 informations about the scope of concepts and thus their extensions, let us identify the area a CRUD operation has to be performed on. Like mentioned above, concept scope has a connection to the idea of ODPs, but it could also be determined in non pattern based ontologies. The separation of concepts according to their scope benefits from the pattern based character of the ontology. For this reason and the big impact on CRUD behavior, we mention the scope independent from the informations derived from ODPs. To distinguish between classes representing concept with ontology wide scope and classes with pattern wide scope we could analyze the whole ontology, to find out which `classes` are used in multiple patterns an which are only used in one pattern. But this would give us only hints about the concrete scope of an concept. In some cases concepts intended to have global scope are only referred by a single pattern. Automated computation of such information from the ontologies could help the user, but the single decision are strongly influenced by application concerns and thus by the user.

Implementation Information: Implementation related information must be included in the MoOn: The last set of useful information is strongly tailored to the targeted implementation. Especially to the chosen programming language and the persistence framework used in the implementation. The concrete implementation

depends strongly on the used programming language. Every programming language is limited in the features it supports. If we choose Java for example we have to deal with the peculiarities of Java, e.g., Java does not support multiple inheritance, it is only possible to implement multiple interfaces not to extend from multiple classes. In cases we depend on multiple inheritance, it is useful to place information about the intended API realization in the MoOn. What should be interface and what class?
Another origin of implementation related information is the persistence layer or database API that should be used by the intended ontology API. Nearly every persistence layer needs some kind of meta infoamtion about the objects/classes that should be mapped. Many of the introduced layers are Java annotation based. They use meta information placed in an annotation on the concrete object or field for mapping. In case of an ontology and a RDF serialization this meta information reflects the particular relationships and declarations coming from the ontology concept this class represents. This information can be derived from the ontology and should be placed in the MoOn to support the implementation based on the persistence layer. This information is strongly aligned to a concrete implementation and thus it tailors the implementation independent model to a concrete programming language and frameworks used in the intended implementation. For every supported persistence layer and programming language we need an new extension wrapping this information.

4.2 Design and Specification of the MoOn

Based on the requirements defined above we will develope the meta-model for the MoOn. We introduce our MoOn meta-model as an UML Class Diagram based meta-model, extended by the OWL UML Profile from the ODM. Additional to this we use ODMs set of mapping conventions for OWL but as described we modified some of them. This builds up the basis for our MoOn capable for the representation of Ontologies in UML2. An overview on the mappings of OWL constructs to UML2 could be found in the Table 5.1 on Page 61. Despite from the syntax and semantic of the model regarding the representation of ontologies, we will focus the integration of the additional informations defined in 4.1.3 into an MoOn. Some of the information could be integrated with standard UML2 constructs, for some others we have to define additional extensions to be able to place them into the model. Others could be modeled by defining a modeling conventions or a specified model structure. For the first ones we give integration instructions. For those we need additional extensions for, we define them. And for the last group we define modeling conventions or a basic structure of our model.

ODP Information: The information about the pattern based structure of the represented ontology is modeled in the MoOn through the special structure of the MoOn. We define a reliable structure for the MoOn. The basic UML element of every MoOn is a model, in this model multiple packages are defined. These packages are stereotyped with ≪owlOntology≫and each of these packages contains one ODP or a non pattern based ontology.

Scope, concept and implementation Information: Informations about the concept scope and the intended API representation could be combined in a new extension to Model. We realize the extension through a new UML profile. This profile is strongly aligned to the Winter API, decribed in 2.7 and 6.2. It describes the concept scope and additional winter related information. This profile defines two stereotypes: ≪WinterLocalObject≫for concepts with local scope and a API representation and ≪winterGlobalObject≫for concepts with global scope and API representation. Concepts without APi representation are not stereotyped in this way.

Cardinality information: For information about the axiomatization of the ontology, through number restrictions and existential quantifiers we do not need an additional extension. We use UML cardinalities to place such information in the MoOn. It is possible to add cardinalities to every kind of association or class property in UML. Cardinalities give use important information about the manner of inter concept relationships, optional or mandatory, in a ODP and about concept properties.

4.3 A full MoOn for our running Example

In this section we present a MoOn for our running example. The Figures 4.1 to 4.4 show the MoOn for our running example. As we know the running example consists of three different patterns, the AnnotationPattern, the DataValuePattern and the DecompositionPattern. Figure 4.1 shows the basic structure of the MoOn model, we can see three different packages one for each pattern. Unfortunately the editor does not visualize stereotypes on models or packages, all packages in the figure are stereotyped with ≪owlOntology≫. Figures 4.2 to 4.4 show the pattern packages and the patterns models.

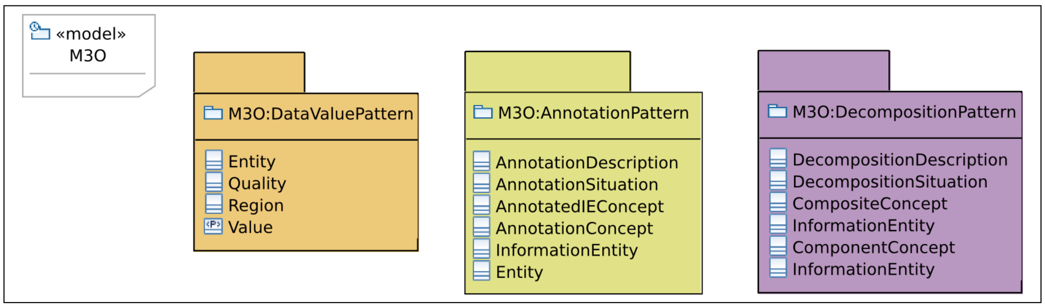


Figure 4.1: The MoOn for our running Example

Figure 4.2 shows the MoOn of the AnnotationPattern. Each class in this package is stereotyped with two different stereotypes. The first one ≪owlClass≫ denotes the UML:Class to represent an owlClass. This stereotype defines a field `URIRef` to define the concept URI of the annotated class, this URI can be derived directly from the `rdf:about` statement of the corresponding `owl:Class` definition in the ontology. The second annotation is Winter and mapping related, it denotes the class to be of global or local scope and allows to define Winter related informations in the model. In the transformation from the MoOn to the OAM, only global classes will get an class representation in the OAM and finally in the ontology API. This would be discussed in detail in Chapter 5. You may also recognize the cardinalities on the associations between the UML classes. The existential and number restrictions from the pattern definition are translated to cardinalities in the model. The associations are stereotyped, in this case with ≪objectProperty≫, this stereotype also defines a field `URIRef` for the predicate URI, this URI could also be derived from the OWL ontology. Additionally the associations are named according to their predicate name in the URI.

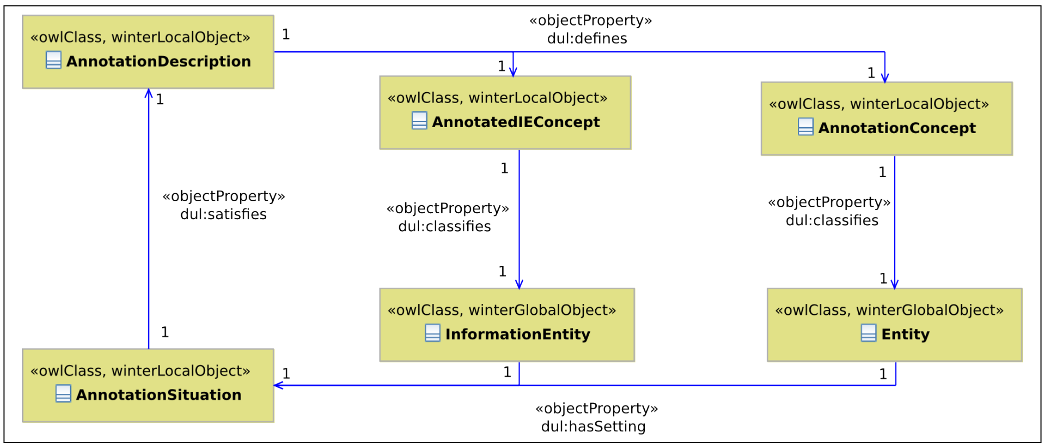


Figure 4.2: The AnnotationPattern Package

Figure 4.3 shows the DecompositionPattern MoOn and Figure 4.4 the DataValuePattern MoOn. Again all the classes and associations are stereotyped in the same way as in the AnnotationPattern MoOn and the cardinality information is present on the associations between the UML classes. In the DataValuePattern we recognize a new stereotype on the last association connecting the `Region` with the `Value`, ≪datatypeProperty≫ and we can also recognize that value is not stereotyped. In this case `Value` is not a `UML:Class` it is a UML:Primitive representing a literal in the ontology.

4.4 Summary

To define the concrete meta-model of our UML MoOn, we will sum up all the statements and observations made in the previous discussions. As basis we use the UML2 Class diagram in combination with the OWL profile from the ODM. With such a model we are able to model nearly all concepts defined with OWL DL and with the stereotypes provided by the profile we could denote the OWL origin for each UML2 entities. We constraint the basic structure of a MoOn model to be a `UML:Model` containing a collection of `UML:Packages`. Each of those `UML:Packages` contains the model for a non

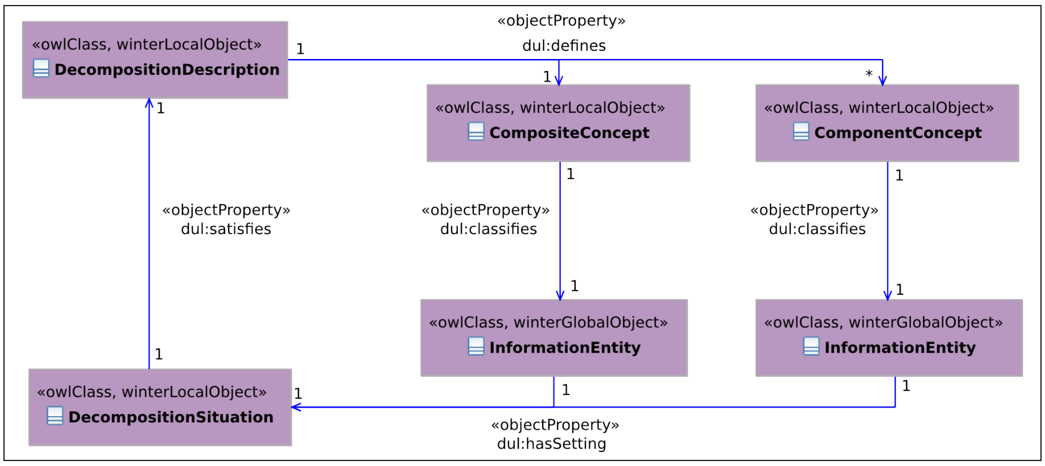


Figure 4.3: The DecompositionPattern Package

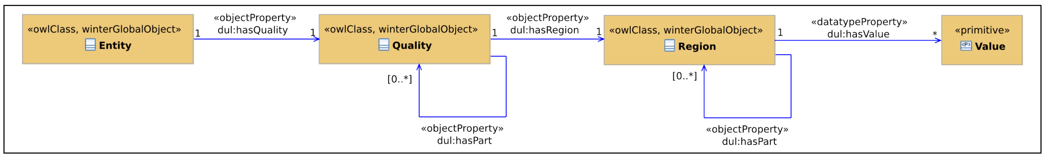


Figure 4.4: The DataValuePattern Package

pattern based ontology or a single ODP. Cardinality information from the ontology will be placed in the MoOn using the cardinality syntax of UML2. Additionally we define extensions to the MoOn for concept scope information and to tailor it to a concrete implementation. Because of the strong dependency of an extension to the concrete implementation we discuss such an implementation related extension in a later chapter in combination with a concrete ontology API implementation model. The extension regarding the scope of single concept is realized through a new UML2 profile. This profile defines two stereotypes ≪global≫ and ≪local≫. Those stereotypes could be used to mark the single concepts with their scope. Another extension, also realized by an UML2 profile, concerns the relevance of single concepts in the API generation process, as described in Section 4.1.3. An initial automated markup with these stereotypes could base on the analysis of the ontology structure. The concept scope we could be analyzed through observing the occurrence of the particular concept in the different patterns. The relevance of named concept could be figured out regarding the ontology before and after the replacement of the anonymous concepts.

5 Ontology APIs and the Ontology API Model (OAM)

In the previous section, we defined the MoOn, a model for ontologies tailored to the task of model driven ontology API generation. The generation process starts with the MoOn and ends in an object-oriented ontology API. Generating code from an ontology is not a new idea. We introduced some approaches in the related work Section 2.7. In some aspects our approach builds on the previously introduced projects. For example our prototype uses Winter as underlying persistence framework for the generated APIs and we make use of the ontology-to-API mappings defined in [?]. In some other aspects our approach goes a different way. Owing to multiple reasons discussed in the previous sections, we decide to start with a model for ontologies the MoOn and transform the MoOn to the Ontology API Model (OAM) and not directly generating an API from it. We introduce the reasons for this decisions in this section and discuss advantages coming with this intermediate model in the next section. Despite from the ontology data aligned CRUD behavior, we discussed in 3, such an API should provide the user a friendly and understandable interface to use the API in an pattern-based application. In this chapter we will discuss strategies to build/generate APIs that work with ontology based data. We will focus on the decisions behind the generation/mapping process that leads from the MoOn to the OAM and discuss general strategies to build ontology APIs. Building an APi for ontologies always means to bridge the fundamental differences inheriting from the different natures of DL based knowledge representation and object oriented systems like those mentioned in [16].

5.1 Mapping Complex Constructors to an API

In Section 2.5.1 and 2.5.3 we described different constructors for concepts and in Section 3.3 we described their influence on the CRUD operations and problems arising with API representations of such concepts in combination with object persistence and CRUD. We introduced simple mapping schemas for DL based constructors, concepts and their representations in the MoOn. We will use these mapping schemas to transform concept representations from the MoOn to their object oriented models in the OAM. We describe the strategies behind these mappings. We show that mapping such constructors to an API model or a concrete implementation is not trivial, not unambiguous and even sometimes not necessary or valuable. Building class representations for complex concepts could easily lead to side effects in the CRUD behavior. Especially when regarding a couple of sequentially performed operations as described in the discussion on equivalence in

Section 3.3.2. At the end, as we denoted in Section 3.3, the important thing is that the serialization is valid according to the ontology.
Analogous to our discussions in Section 3.3, where we have already analyzed DL constructors according to their influence on CRUD, our focus lies again on the DL constructors. In this chapter we continue our discussion on mapping single constructors or combinations of them to an ontology API or API model. Regarding the API representation of concepts we could denote that such an representation should always serve a special purpose. The main purpose of an ontology API in general is to abstract parts of the Knowledge Base in the software and provide persistence functionalities for these abstractions. Issues regarding the persistence, we discussed in Section 3 and we gave initial hints regarding our question for this chapter: How to abstract objects from a Knowledge Base? In the following we discuss the single DL constructors, if it is useful representing them and ways to represent them in an object-oriented ontology API. Additionally we introduce strategies for combinations of multiple constructors. Based on our observations on DLs we define the mappings for OWL expressions and their representations in MoOn in the Table 5.1 in a the following subsection.

1. **Equivalence:** For a concept definition including equivalence there are two possible solutions to map it an API representation. If we declared two named concepts to be equivalent it might be useful to have two different representations in the object-oriented API. On the other hand when the equivalence is only used in a concept definition involving multiple constructors, like in $A \equiv B \sqcap C$, the representation of both concepts A and the anonymous $B \sqcap C$ is not necessary. In such a case, most of the time we only want the named concept on the left side of the equivalence operator to have an API representation. Through the transformation of the ontologies to a form without anonymous concepts multiple named concepts with no direct class representation might arise. In such a case it makes sense to remember the named concept before the ontology is decomposed like mentioned in Section 3.3. Additionally the user has to define the MoOn carefully to avoid unnecessary classes.

2. **Union:** If an API is mapping the concepts to corresponding classes representations, we could simulate the union through a common supertype. This approximates the semantic of the union in a object-oriented environment. Regarding the example of a union in Section 3.3, such an approximation allows us to cast a instance of an class representing $father$ or *mother* to the type of the class representing *parent*. This makes sense in environments where it should be possible to declare or reference instances of classes who are combined to define such a `union-class`, as a members of such `Union-Class`. E.g., referencing a father instance from a parent field.

3. **Intersection:** In Chapter 3.3 we denoted `intersection-class` should be approximated in the object-oriented world by a subclass inheriting from all classes to be combined. In programming languages without multiple inheritance we have to use language specific concepts to inherit from multiple classes or use work-arounds like inheritance chains. In Java for example we have the opportunity to use interfaces to

declare the concept representations our `intersection class` could inherit from. Referring again to your example from Section 3.3 we would expect two classes/interfaces for *human* and *female*. Based on these we could define a class/interface representation for *women* that inherits from the two classes or interfaces. In the case of interfaces we define default implementations for such interfaces.

4. **Complement:** In the standard programming language it is not possible to define a class in a way the complement operator allows it for concepts. We do not need a strong restriction to define the class extension, like we need it for concepts. In DL every individual that fits the restrictions of a concept is part of its extension. In contrast to this, in the object-oriented world only what is constructed from a class is part of the extension. The complement constructor is important in the definition of concepts and their individual sets, but in a object-oriented environment all classes and thus their extensions are disjoint unless stated otherwise. This would make all other classes to be the complement of a particular class. It is possible to abstract this generality in DLs trough common supertypes but this would be very verbose and does not make sense.

5. **Number Restrictions:** It is easily possible to bring number restrictions to the API model. But implementing them in the API needs much more work. First of all a number restrictions gives us a hint on how to represent such restricted property in a class. A `maxCardinality` of one leads us to a single value field, if the `maxCardinality` is greater that one we will need a collection for the property in the declaring class. In our case we do not use collections we use sets because each individual is unique. But number restrictions gives us more information about the intended behavior of the implementation. The `minCardinality` lets us know if a property is mandatory or optional. All properties with a `minCardinality` above null are mandatory. For `minCardinality` above 1 we need control functions counting such properties and check if the restrictions are fulfilled. The could be a listener method allocated to the concrete field counting the fields size and alert if the number constraint is violated. Or a method that counts if a create and delete operations is performed the particular field, to ensure the validity and completeness of the declaring class. Counting could also be necessary if we have maximal value restrictions, we have to ensure that they are fulfilled.

6. **Existential quantifier:** Existential quantifiers like $\forall$ or $\exists$ are used in DL to restrict the extension of a class to individuals fulfilling special prerequisites. In an object-oriented environment we define classes and all instances fulfilling the requirements. So existential quantifiers only help us in designing the object-oriented concept representations in the ontology API but we do not need special entities in the API to represent them. We could derive cardinality information about properties from existential quantifiers as mentioned in Section 3.3.

7. **Value Restriction:** A value restriction limits, as the name says, the value a particular property can achieve. To reflect such behavior in an API we need

special types or methods checking the validity of the value of the corresponding field. We need some kind of listener methods observing the value on create and change operations to ensure its validity according to the value. In case of strings it might make sense to ensure this with the definition of enumerations.

8. **Disjointness:** In the object-oriented world classes are inherently disjoint unless stated otherwise. If necessary, it is possible to implement common super or sub-classes to model such a behavior. But generally there is no need for an API representation of disjointness.

5.1.1 Mapping OWL

Several OWL expressions have no direct equivalent in the software world, especially in the common object oriented languages. To be able to build APIs from ontologies we must bridge this gap between the object-oriented-world an the logic based ontology-world. In the following table we present OWL expressions as URI references and their equivalent in the UML ontology model according to the ODM version 1.0. In the third row of the table we define the API representations for the single OWL constructs. For complex constructs refer to the related discussion below the table.

Table 5.1: OWL expression to API mapping

OWL EXPRESSION	UML ONTOLOGY MODEL (ODM)	UML API MODEL	COMMENT
owl:AllDifferent	≪allDifferent≫ *UML::Constraint*	see Discussion #1.	Applies only between Individuals
owl:AllValuesFrom	*UML::GeneralizationSet* and property redefinition, A.10	see Discussion #2.	—
owl:Annotation	≪annotation≫ *UML::Element*	UML Comment or Stereotype	Not in API Model, maybe Documentation see Discussion #16..
owl:AnnotationProperty	≪annotationProperty≫ *UML::Class UML::Association*	UML Comment or Stereotype see Discussion #16.	Not in API Model, maybe Documentation
owl:backwardCompatibleWith	≪backwardCompatibleWith≫ *UML::Constraint*	see Discussion #3.	Not in API Model
owl:cardinality	apply multiplicities on *UML::Properties* or *UML::Association*	s. Ontology Model	for inherited properties redefine

Continued on next page

Table5.1 – continued from previous page

OWL expression	UML Ontology Model (ODM)	UML API Model	Comment
owl:Class	≪owlClass≫	Class	Stereotype subclass of ≪rdfs:Class≫
owl:complementOf	≪complementOf≫ *UML::Constraint* s. **??**	see Discussion #4.	—
owl:DatatypeProperty	≪DatatypeProperty≫ *UML::Association* or *UML::Property*	*UML::PrimitiveType*	Stereotype
owl:differentFrom	≪differentFrom≫ *UML::Constraint*	see Discussion #1.	Applies only between Individuals
owl:disjointWith	≪disjointWith≫ *UML::Constraint* s. A.12	see Discussion #5.	—
owl:distinctMembers	—	see Discussion #6.	Not in Ontology and API Model
owl:equivalentClass	≪equivalentClass≫ *UML::Constraint* s. A.15	see Discussion #7.	Applied to OWLClass
owl:equivalentProperty	≪equivalentProperty≫ *UML::Constraint* between two *UML::Class*	see Discussion #7.	≪rdfProperty≫, ≪owlProperty≫, ≪objectProperty≫, ≪datatypeProperty≫
owl:FunctionalProperty	≪isFunctional≫ on *UML::Property*, *UML:Association*	see Discussion #8.	Applies to properties or associations
owl:hasValue	*UML::Property* redefinition	see Discussion #12.	—
owl:imports	*UML::Property* of the stereotype it describes	see Discussion #10.	Not in API Model
owl:incompatibleWith	≪incompatibleWith≫ *UML::Constraint*	see Discussion #3.	Not in API Model
owl:intersectionOf	≪intersectionOf≫on *UML::Constraint* s. A.16	stereotyped subclass of all intersected classes	faulty see A.17 for fix

Continued on next page

Table5.1 – continued from previous page

OWL expression	UML Ontology Model (ODM)	UML API Model	Comment
owl:InverseFunctionalProperty	≪isInverseFunctional≫ on *UML::Property* or *UML::Association*	see Discussion #8.	
owl:inverseOf	≪inverseOf≫ *UML::Association*	see Discussion #8.	—
owl:maxCardinality	apply multiplicities on *UML::Properties* or *UML::Association*	s.Discussion about Cardinalities	inherited properties redefine
owl:maxQualifiedCardinality	apply multiplicities on *UML::Properties* or *UML::Association*	s.Discussion about Cardinalities	inherited properties redefine
owl:minCardinality	apply multiplicities on *UML::Properties* or *UML::Association*	s.Discussion about Cardinalities	inherited properties redefine
owl:minQualifiedCardinality	apply multiplicities on *UML::Properties* or *UML::Association*	s.Discussion about Cardinalities	inherited properties redefine
owl:Nothing	one UML:Class for owl:Nothing	see Discussion #11.	Not in API Model
owl:ObjectProperty	≪objectProperty≫ *UML::Property* or *UML::Association*	Property	Stereotype subclass of ≪rdfProperty≫
owl:oneOf	*UML::Enumeration* over *UML::Class*	see Dicussion #8.	—
owl:Ontology	≪ontology≫ *UML::Package*	see Discussion #13.	Not in API Model
owl:OntologyProperty	≪ontologyProperty≫ *UML::Association* *UML::Class*	see Discussion #13.	Not in API Model
owl:priorVersion	≪priorVersion≫ *UML::Constraint*	see Discussion #14.	Not in API Model
owl:qualifiedCardinality	apply multiplicities on *UML::Properties* or *UML::Association*	s.Discussion about Cardinalities	inherited properties redefine
owl:Restriction	≪owlRestriction≫ on a ≪anonymous≫ *UML::Class*	own Stereotype	Special case: Property restr., restr. class is supertype of containing class

Continued on next page

Table5.1 – continued from previous page

OWL expression	UML Ontology Model (ODM)	UML API Model	Comment
owl:sameAs	≪sameAs≫ *UML::Constraint*	see Dicussion #15.	Applies only to instances in OWL DL
owl:someValuesFrom	*UML::Property* redefinition	see Dicussion #2.	like: owl:allValuesFrom, owl:hasValue
owl:SymetricProperty	≪isSymetric≫ on *UML::Property*	see Dicussion #8.	—
owl:Thing	one UML:Class for owl:Thing	see Discussion #11.	Not in API Model
owl:TransitiveProperty	≪isTransitive≫ on *UML::Property*	see Dicussion #8.	—
owl:unionOf	*UML::GeneralizationSet* with isCovering = true	see Discussion on union	similar to owl:intersec-tionOf
owl:versionInfo	≪versionInfo≫ String in ≪rdfDocument≫ or ≪owlOntology≫	see Discussion #16.	Applies to *UML::Package* Not in API Model.

Discussion

Here we discuss the different OWL constructs mentioned in the table above. For a closer look on the semantics of the single OWL constructs refer to [20, 34]. In cases where an API representation makes sense we discuss possible API representations. In cases there an API representation makes no sense we explain why.

1. The `owl:allDifferent` and `owl:differentFrom` mechanisms provide the opposite effect from `sameAS`. The `owl:differentFrom` mechanism could be understood as a not `sameAs` and thus like a $\neg \equiv$ statement in a DL. With `owl:differentFrom` we can denote in opposite to `sameAs` that an instance is different to another instance. With `owl:allDifferent` we can denote that individuals in a list (`owl:distinctMembers`) are all different from each other. Due to the fact that these two constructs operate on instances it makes no sense to map them to the class definitions in an API.

2. The `owl:allValuesFrom` and the `owl:someValuesFrom` restricts the type of the elements that make up a property. Just like to the $\forall$ operator in DL the `owl:allValueFrom` restriction forces that for every instance of the class that has an instance of the restricted property, the values of the property are members of the class indicted by the clause. Just like the $\exists$ operator `owl:someValuesFrom` forces at least one value to be of the defined type. Stronger than `allValuesFrom`, it enforces a min cardinality of one for this property.

In the object oriented environment the type of a poperty is restricted to the type in the property definition. A restriction made in a concept with `owl:AllValuesFrom` or `owl:someValuesFrom` gives us additional information about the specification of properties in the class representation for the concept. The `owl:someValuesFrom` statement gives us additional cardinality informations for our class representation. As example for `owl:AllValuesFrom`, lets take a look at following Listing 5.1 the wine ontology used as example in the OWLGUIDE [34]:

Listing 5.1: `owl:allValuesFrom` in the wine ontology

```
<owl:Class rdf:ID="&food;PotableLiquid">
</owl:Class>

<owl:Class rdf:ID="PotableLiquidMaker">
        ...
        <rdfs:subClassOf>
                <owl:restriction >
                        <owl:onProperty rdf:ID="#makesLiquid">
                        <owl:allValuesFrom rdf:resource="#PotableLiquid">
                </owl:restriction >
        </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Winery">
        <rdfs:subClassOf rdf:resource="PotableLiquidMaker" />
        ...
                <rdfs:subClassOf>
                <owl:restriction >
                        <owl:onProperty rdf:ID="#makesLiquid">
                        <owl:allValuesFrom rdf:resource=#Wine>
                </owl:restriction >
        </rdfs:subClassOf>
</owl:Class>


<owl:Class rdf:ID="Wine">
        <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />
        ...
        <rdfs:subClassOf>
                <owl:Restriction >
                        <owl:onProperty rdf:resource="#hasMaker" />
                        <owl:someValuesFrom rdf:resource="#Winery" />
                </owl:Restriction >
        </rdfs:subClassOf>
        ..
</owl:Class>
```

This example could be mapped to a class representation in defining classes for `PotableLiquidMaker` and `PotableLiquid` as superclasses for the whole category. The classes for `Winery` and `Wine` should be defined as subclasses from the category classes. The `hasMaker` and `makesLiquid` property are mapped as fields of the class representation. The OWL property names are used as names for the fields (properties) in the UML representation. These UML properties are typed by the referencing concept representations. HasMaker is of type Winery and makesLiquid of type Wine. Here we can see that the strong type safety in object-oriented environments forces us to restrict the type of the field `makesLiquid` in the `Winery` class to Wine. Acording to the ontology a Winery only has to produce at least one liquid of type wine. It is possible to model this through an additional liquid supertype in the API. But in this case we need checking methods ensuring that an winery produces at least one wine. Regarding the cardinality coming with `owl:someValuesFrom` the hasMaker could be mandatory to the Wine class. A Wine object must have at least one hasMaker property of type Winery referring to a Winery object.

3. The `owl:backwardCompatibleWith` and `owl:incompatibleWith` denote campatibility or incompatibility to another ontology. Both indicate that the containing ontology is a later version of the referenced ontology, in case of `owl:backwardCompatible` they are compatible in case of `owl:imcompatibleWith` they are not backward compatible. In an API representation of an ontology compatibility information of the underlying ontology is of no concern. It might be an interesting additional information that could be places in the documentation of such an API. But it has no direct influence on the API generation from OWL and on the functionality of the API itself. It might have influence when trying to access data based on an older version of the ontology the API was created with.

4. The `owl:complementOf` construct selects all individuals in the domain not belonging to a certain class. The `complementOf` construct is just like the complement operator $\neg$ in DL. See the discussion of the complement operator in the DL discussion above.

5. The `owl:disjointWith` constructor allows us to define disjoint sets of classes. It guarantees that an individual is only member of one of those classes. For APi mapping see the discussion on disjointness above.

6. The `owl:distinctMembers` can only be used with `owl:AllDifferent`. It is used as begin and end tag for the list of pairwise different individuals. It makes no sense to define a single to-API-mapping for this constructor. See the discussion of `owl:AllDifferent`.

7. To denote equivalence between classes or properties OWL defines `owl:equivalentClass` and `owl:equivalentProperty`. This correponds to the equivalence operator $\equiv$ in DL. In an API we suggest to model only one class representation for equivalent classes, otherwise you could easily run into the described CRUD problems. In some cases it might make sense to implement classes for each of the equivalent owl classes. For an ontology API with multiple equivalent classes we need reasoning services to preserve validity when performing CRUD operations. Additional to this it is recommended to implement a common superclass for equivalent classes to be able to use both classes in properties of other classes. For additional information about mapping such a construct to an API refer to our discussion on DL equivalence.

8. With the `owl:FunctionalProperty`, `owl:inverseOf`, `owl:SymmetricProperty`, `owl:TransitivePrope` and the `owl:InverseFunctionalProperty` construct we are able to further specify properties. With property characteristics we can enhance reasoning about the specified property.
   The `owl:FunctionalProperty` operator gives us detailed information about the cardinalities of the property and the relationship behind. A functional property is unique, thus it has a upper bound `owl:maxCardinality` of one. Such a property could be mapped to a single field in an API class.

The `owl:inverseOf` statement between two properties denoted that the one property is the inverse of the other. For example Listing 5.2 shows us the use of `inverseOf` in the wine ontology.

Listing 5.2: `owl:inverseOf` in the wine ontology

```
<owl:ObjectProperty rdf:ID="hasMaker">
  <rdf:type rdf:resource="&owl;FunctionalProperty" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="producesWine">
  <owl:inverseOf rdf:resource="#hasMaker" />
</owl:ObjectProperty>
```

This denoted every `Wine` to have a maker, which is by definition a `Winery`. Each `Winery` produces the set of wines that identify it as maker. In an API a wine class would have one field for the hasMaker property of type winery and the winery class would have a collection field for the producesWine property of the type wine.
The `owl:SymmetricProperty` denotes a property to be symmetric. Symmetric in this context means that the property is its own inverse. In an API representation this means that if an class X has an symmetric property field of type Y, then class Y has to have an symmetric property field of type X.
The `owl:TransitiveProperty` denotes a particular property to be transitive. In a chain of such a properties, every property has a direct relationship to all following. It is possible to map this to an API by defining transitive property fields as collection and adding all objects in the transitive chain or through iterative getter methods on transitive fields descending through the object and giving back a sorted structure. But this is more complicated to implement and we would suggest to use reasoning services to access objects referenced through transitivity. For example in an extra getter method for transitive property fields using reasoning services to get back all corresponding instances.
The `owl:InverseFunctionalProperty` is just syntactic sugar and represents a combination of `owl:FunctioalProperty` and `inverseOf`.

9. The `owl:hasValue` construct restricts a particular property to be of a particular value. This allows us to define classes based on the existence of particular property values. For an to-API-mapping of a class definition based on a `owl:hasValue` construct, this provides useful information about the properties of the API class. Especially it could give us informations about additional useful specializations of the types of the property. Listing 5.3 shows us the use of `hasValue` in the wine ontology.

Listing 5.3: `owl:hasValue` in the wine ontology

```
<owl:Class rdf:ID="Burgundy">
  ...
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasSugar" />
      <owl:hasValue rdf:resource="#Dry" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This declares that all Burgundy wines are dry. In an API this could be useful to map it to an subclass of `wine` where the `hasSugar` property is dry by default.

10. The `owl:imports` statement can be used to import the entire set of assertions provided by the imported ontology into the importing one. Such an import is important when building our MoOn because all the classes needed should be present in the MoOn. The OAM is generated from the MoOn and thus for the OAM or API generation the imports in the ontology should be present in the MoOn.

11. Every individual in our Ontology is member of `owl:Thing` and thus every named class in our ontology definition is subclass of `owl:Thing`. The `owl:Nothing` class is the complement to `owl:Thing` and thus it is the empty class with no members by default. In the OAM or an API it is possible to produces such inheritance structure, especially in Java where every class is subclass of `Object`. But that leads to no advantages at all, it may even lead to disadvantages regarding the flexibility of the API. We propose to not modelling `owl:Thing` in an API representation. For `owl:Nothing` there is no proper way to represent such a construct in the OAM or API.

12. With `owl:oneOf` OWL provides the opportunity to specify a class via a direct enumeration of its members. In UML2 or an object oriented programming language instances are created from classes. A class definition is a named schema of properties and methods. Thus it is not possible to define a classes via a set of instances. We still have the opportunity to approximate such a behavior in the OAM or API by using enumerations. In Listing 5.4 we can see the use of the `owl:oneOf` statement to denote wine color to be one of #White, #Rose and #Red.

Listing 5.4: `owl:oneOf` in the wine ontology

```
<owl:Class rdf:ID="WineColor">
  <rdfs:subClassOf rdf:resource="#WineDescriptor"/>
  <owl:oneOf rdf:parseType="Collection">
    <WineColor rdf:about="#White" />
    <WineColor rdf:about="#Rose" />
    <WineColor rdf:about="#Red" />
  </owl:oneOf>
</owl:Class>
```

This could be mapped to an enumeration winecolor with the three strings, White, Rose and Red. But this approximation only applies if we have a primitive type property. When we are able to define the possible values using a enumeration. In case of a property referencing a complex type we have to subclass from this type for every preset we want to distinguish in the enumeration.

13. We assume that, in an ODP based ontology, every pattern comes in its own ontology. Thus the `owl:Ontology` statement and especially the structure serves us to distinguish between the different patterns. This has a direct influence on the API by naming the different pattern classes with the names from the pattern declaration. `owl:OntologyProperty` is the class for all additional information on ontologies. For instance an `owl:imports` is an instance of `owl:OntologyProperty`. There is no need of the `owl:OntologyProperty` information in the OAM or the API.

14. A `owl:priorVersion` statement identifies the specified ontology to be a prior version of the containing ontology. Like compatibility information such version information about ontologies is of no interesst in ontology APIs. It may make sense to include such an information in the documentation of an API.

15. In OWL DL, with `owl:sameAs` we are able to denote equivalence between individuals. OWL has no unique naming assumption, so two different names in an OWL statement can refer to the same individual. So `owl:sameAs` is only used in the individual describtion, the A-Box but we use the schema describtion as starting point of our transformations. It makes no sense formulate mapping conventions for `owl:sameAs`.

16. The `owl:Annotation` and the `owl:AnnotationProperty` statements in OWL1 are constructs without any semantic in the ontology. Information provided by such annotation could be placed in the OAM or API documentation but has no influence on the intended behavior. A `owl:versionInfo` statement is an instance of `owl:AnnotationProperty`.

All mappings for OWL constructs defined here are optional. The decision what should be mapped and what should not, lies in the hands of the user. For example the user could decide to skip the `owl:imports` construct in the mapping process. It is easy to exclude particular mappings by just not modeling them in the MoOn. Consequently OWL constructs not present in the MoOn could not be mapped to the OAM or the API. When we decide not to map single constructs we have to keep in mind that the essential thing is a valid serialization. We have to ensure not to leave out constructs mandatory for a valid serialization.

5.2 Mapping Models for Ontology-to-API mapping

An API working on ontology-based data-set should provide CRUD behavior as mentioned in Section 3. The concrete implementation of such CRUD operations depends strongly on the chosen structure of the API and the underlying object persistence layer. Most of the triple persistence layers and the automated ontology API generators, like those introduced in 2.7, refer to a persistence model strongly aligned to the RDB persistence models known for years. These models map each ontology concept to a matching class on API side. Such a class would serialize to a single statement like :

```
IndividualIDURI <rdf:type>        ConceptIDURI.
```

We described this behavior in Section 3.3. Generally this concept-to-class mapping fits the requirements of ontology based object persistence. But such a mapping makes no use of useful information coming with ODP based ontologies, like the pattern based structure. Using simple concept-to-class mapping we have no representation of such information in the API. But as denoted below pattern related information might be useful for CRUD and for other concerns. Lacking such information makes the implementation of adequate CRUD behavior more difficult. In a structure resulting from concept-to-class mapping we need an additional layer that cares of completeness and validity and that abstracts the lost pattern information from the ontology. In the following we will discuss the concept-to-class mapping and the disadvantages coming with it and in a following section we propose a mapping model that solves these problems.

5.2.1 Concept to Class mapping

One of the general problems, regarding CRUD, coming with the use of an concept-to-class mapping is that inter-concept-relationships are encoded in inter-class-relationships in the resulting class structure. With using such an mapping we loose all information about the pattern structure, that could help us to distinguish between the different services this ontology provides for us. We will end up in one general view on a configuration of the whole ontology. To preserve the pattern character in the API and to abstract services requiring multiple CRUD operations, like the instantiation of an whole ODP, we need one or multiple additional layers on top of the classes representing our ontology entities.
So building an application on top of an ontology API generated by a process that maps each OWL class to an API class forces us to implement multiple control mechanisms. We need these mechanisms to ensure the validity and completeness of an extension of such an API structure. In the end these are many tasks for just one layer, so maybe it makes sense to use multiple layers, one for crud control one for service abstraction e.t.c. All these layers need information about the ontology structure. Not a good way to encapsulate the ontology structure from the application and a straight way into an application with multiple instances of structure information, each control layer holds parts of the ontology structure. So we can sum up the problems coming with concept-to-class mapping as follows:

- ontological relationships encoded in object relationships.
- one view on the instance data of the whole ontology, we lost the pattern information.
- multiple structures or one with multiple concerns to provide services and ensure validity.
- each control structures must have ontology structure knowledge, multiple structures → multiple instances of structure information

For example the `AnnotationPattern` mapped with concept-to-class mapping to an API representation will result in six classes. One class for each concept in the pattern and each of the classes would have a property field for each outgoing relationship defined in the pattern. So the class representing the `AnnotationDescription` would have 2 fields of type `AnnotatedConcept` and `AnnotationConcept` for the outgoing `dul:defines` relationships. For classes that could be used in multiple patterns this would mean that they have to declare fields for the relationships of every pattern they could probably play a

role in.
We solve all those problems coming with concept-to-class mapping with our new approach of pattern- to-class introduced in the next section.

5.2.2 Pattern to Class mapping

The general difference of our approach to the `class`-to-class mapping is what we model the inter-object-relationships in a pattern class. Therefore we call this approach `pattern`-to-class mapping. But what does that mean? In our approach every ontology concept is also encoded in an API representation, but in difference to the concept-to-class approach, in our mapping model each of this API representation does not know anything about its relationships defined in the declaring pattern. These previously implicit relationships become explicit through the implementation of the ODP structure in a pattern classes. We implement one extra class for each pattern. Such a pattern class represents all of the inner relationships of the pattern, so we killed two birds with one stone.
The problems occurring with implicit relationships and the lost pattern information in ontology wide class structures. Additional to this, such an pattern class could provide methods abstracting services, like whole pattern instantiation, mapping or deletion, and functionality for CRUD control. Like validity and completeness checking methods or methods that could be parameterized for different intended CRUD behavior. So a control layer, upon this, no longer needs information about the underlying ontology structure. It just has to know which pattern classes refers to which object. This helps us to separate the ontology from the application. Another big point in this separation is that, in ontologies and especially in ODPs we often encounter concepts with nearly no significance to the user or application. For instance most the concepts with only local scope, like defined in 3.1. Often these concepts are only represented through two URIs, the identifier and the concept URI of the class, therefore it is not even necessary to provide a specific classes in the API, for such concepts. It is quite enough to represent them through two wrapped URIs in the pattern class. Through that we are able to hide them from the application and user. Summed up the advantages of such a mapping model will be:

- explicit relationships
- pattern information in the API
- service abstraction
- interface to CRUD control layer
- encapsulation of classes with no significance to the application/user

In case of pattern-to-concept mapping the whole pattern would map to an patter class representation encoding all inner pattern relationsships. The concepts used in the pattern are represented through stand alone classes without any knowledge about their inner pattern relationships. To express such relationship in the API a instance of the

`AnnotationPattern` class would reference all involved instances of the class representations of the concepts in the `AnnotationPattern`.

5.2.3 Concept/Class scope and visibility

An observation made in multiple ontology APIs,e.g., API for the M3O or the eventmodel, we developed was that some concept representations in the ontology API just consists of two URIs as mentioned above. One URI as unique identifier of the individual and one URI for the type provided by the instantiated concept. In Section 3.1, we distinguished concepts in ODPs into two sets, those with ODP wide scope (local) and those with ontology wide scope (global). In the ontology APIs we implemented we additionally observed that most of the classes with local scope in the pattern are of no interest for the user of an concrete API. Regarding the AnnotationPattern, concepts with local scope are those defining the contextualized view. A user of an API working on an ontology based data set, lets say an annotation tool for multimedia data, is not interested in the definition of the contextualized view. An user operates on the entities with global scope, e.g., in case of annotation the image and the annotation. He wants to annotate and thus create a contextualized view - instantiate a whole pattern - and not instantiating the single concept representations with local scope. As you can see most of the class representations of concepts with local scope are of no interest for the user and thus should be hided. We achieve this in combination with the pattern-to-class mapping with implementing the representations for concepts with local scope as properties of the declaring pattern class. Such pattern class should define two fields for each local concept in the ODP. One field for the type and one for the unique identifier. So we do not have first level class representations for such concepts only two different URI fields in the class representation of the declaring pattern. These fields would be allocated on pattern creation time. It depends on the concrete implementation if they should be created separately and passed to the pattern constructor or generated automatically in the pattern constructor. This approach provides possibilities to ensure completeness regarding the local scope concept representations on create time. In such an API is would not be possible to instantiate single instance of local scope classes. And when instantiating whole pattern instances it is ensured that all necessary information is available.

5.3 The Ontology API Model (OAM)

In this Section we will define and discuss our API model. As an intermediate step of the API generation process we decided to introduce the Ontology API Model (OAM) based on the UML2 Class Diagram meta-model and several extensions. These extension are implementation related and we will introduce one for an OAM aligned to Winter in this section. This intermediate model enables the user to influence the generation on an API level. A concrete OAM model is strongly tailored to an concrete implementation. In the OAM we are able to declare specific information needed for the use of an concrete persistence API like Winter (see Section 2.7) in the ontology API. Initially the OAM represents the API structure generated from the ontology representation, the MoOn. We

use the mapping conventions defined in Table 5.1 and the following discussion to generate the OAM from the MoOn. In the previous sections we observed different mapping models, we distinguished between concept-to-class and pattern-to-class mapping. In our approach we provide pattern representations in the API and so such pattern information is also present in the OAM. As a persistence layer supporting pattern representations we tailored our OAM to Winter and include Winter related information into the OAM. This information supports the generation of an ontology API using Winter. Winter uses Java and because of the single inheritance restriction regarding classes in Java we use interfaces as first level representatives for concepts. Because of their nature it is not possible to instantiate from an interface in Java. We also provide class representations implementing the corresponding interfaces. Defining both enables us to be flexible in the code generation and not strongly tied to Java. With the use of interfaces we are able to approximate multiple inheritance in Java.
Based on these mappings defined in this section, we now define a OAM for our running example. The OAM defined here will be for a Java implementation based on the Winter persistence layer described in Section 2.7. But with little modifications of the mapping and/or the extensions it is easily possible to generate a OAM tailored to another language and/or persistence API.

5.3.1 A OAM for Java and Winter

We introduced Winter in 2.7 as Java based persistence layer for RDF data. But Winter is far more than that. With its new mapping concept Winter actively supports complex ontologies and especially ODP based ontologies. Apart from simple object-to-statement mapping Winter supports object-to-multiple_statements mapping and complex object mappings. As in standard object-to-statement mapping every object defines its own object id an URI. Based on this id and the type URI, declared by the concept, each object could be mapped to a RDF triple like:

```
UniqueObjectURI <rdf:type>        ConceptURI.
```

As mentioned in the related work section 2.7, Winter provides functionalities to map single object to multiple statements. This helps us in the implementation of pattern classes, like those discussed in in the previous sections.
As an Example for our explanations we use our running example but this time we focus on an API representation for the example. The figures referenced here show an API model for the three patterns used in our running example. The model is strongly tailored to the needs of our example. Especially regarding the types of some properties in the pattern classes and the corresponding global concept representations. In a real model we would refer to the types defined in the pattern declarations like in the M3O in Section 2.4.1. We used this typing to show the correspondence to our running example. Figure 5.1 shows an UML2 Class Diagram of an API representation in Java using Winter.

As we can see the OAM is organized in three different packages the interface package, the implementation package and the pattern package. Figure 5.2 shows the interface

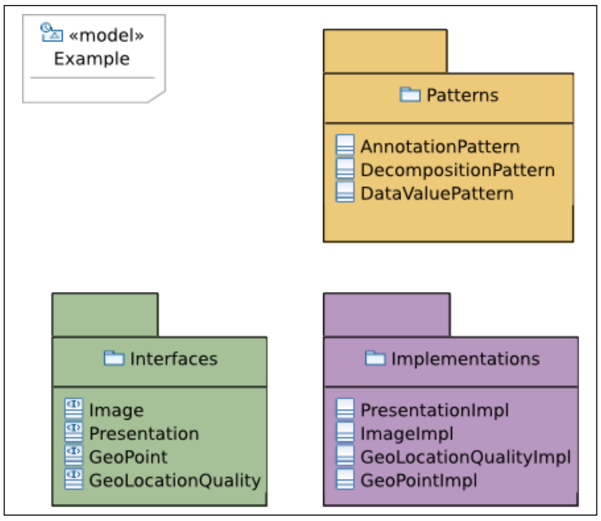


Figure 5.1: The Ontology API Model for our running Example

package. In this package we find interface representations for all concepts with global scope. Figure 5.3 shows the implementation package. In the implementation package we define default implementations for all interfaces in the interface package.

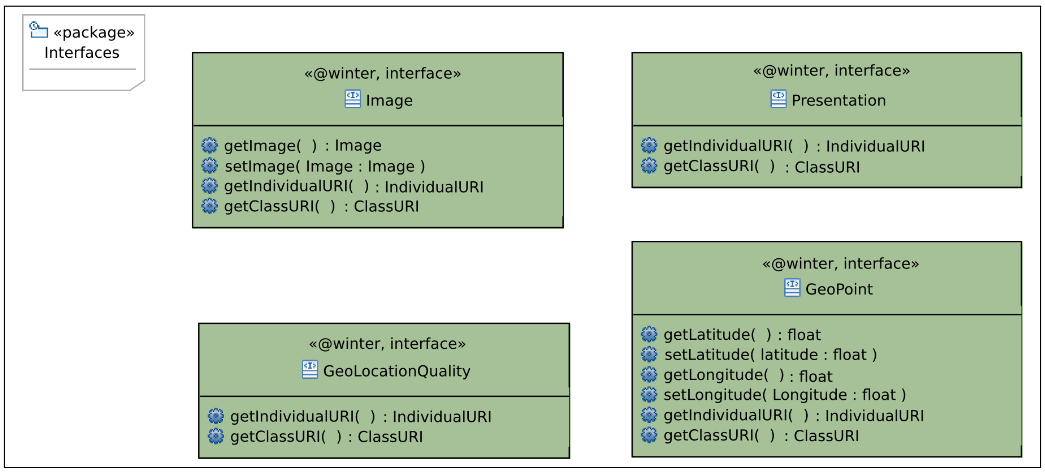

«package»
Interfaces
«@winter, interface»
Image
getImage( ) : Image
setImage(Image : Image)
getIndividualURI( ) : IndividualURI
getClassURI( ) : ClassURI
«@winter, interface»
Presentation
getIndividualURI( ) : IndividualURI
getClassURI( ) : ClassURI
«@winter, interface»
GeoPoint
getLatitude( ) : float
setLatitude(latitude : float)
getLongitude( ) : float
setLongitude(Longitude : float)
getIndividualURI( ) : IndividualURI
getClassURI( ) : ClassURI
«@winter, interface»
GeoLocationQuality
getIndividualURI( ) : IndividualURI
getClassURI( ) : ClassURI

Figure 5.2: The Interfaces Package

In the last Figure 5.4 we see the pattern package. In this package we defined the pattern classes. If we compare this and the implementation package to the running example we notice that we do not have direct implementations for all the concepts with local scope. Regarding the observations made in Section 5.2.3 this becomes clear. All the concepts with local scope are represented as properties of the declaring pattern classes. If we take a look at the `AnnotationPattern` class in Figure 5.4 we can see that this class has the two fields representing the concepts with global scope, GeoPoint and Image.

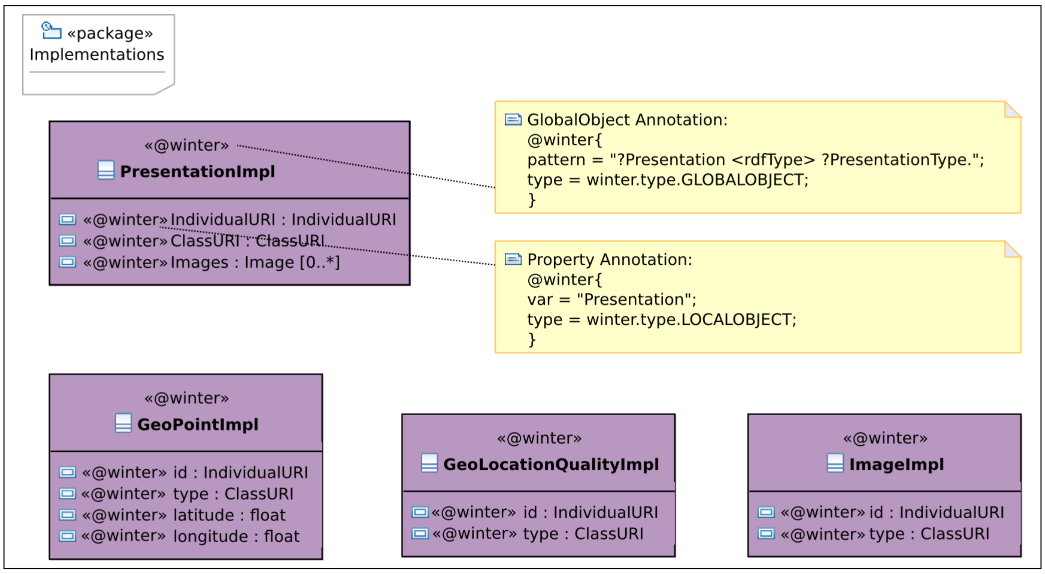
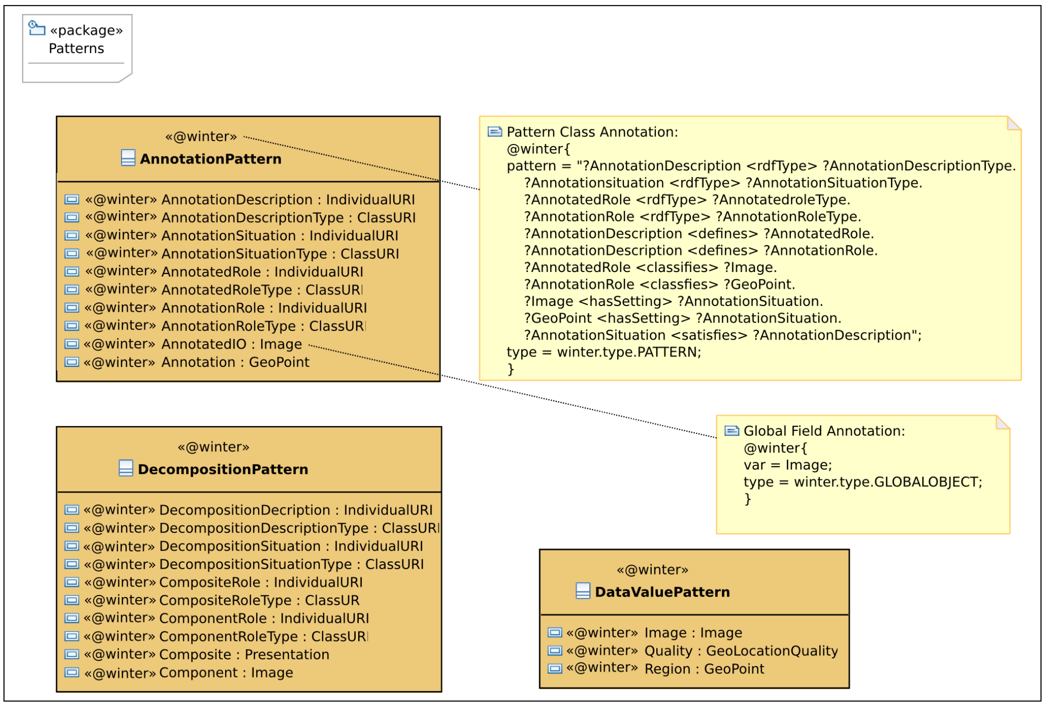

«package» Implementations
«@winter» PresentationImpl
«@winter» IndividualURI : IndividualURI
«@winter» ClassURI : ClassURI
«@winter» Images : Image [0..*]
GlobalObject Annotation:
@winter{
pattern = "?Presentation <rdfType> ?PresentationType.";
type = winter.type.GLOBALOBJECT;
}
Property Annotation:
@winter{
var = "Presentation";
type = winter.type.LOCALOBJECT;
}
«@winter» GeoPointImpl
«@winter» id : IndividualURI
«@winter» type : ClassURI
«@winter» latitude : float
«@winter» longitude : float
«@winter» GeoLocationQualityImpl
«@winter» id : IndividualURI
«@winter» type : ClassURI
«@winter» ImageImpl
«@winter» id : IndividualURI
«@winter» type : ClassURI

Figure 5.3: The Implementation Package

Additional to this we find pairs of fields representing all local scope concepts. One field for the type and the other for the identifier. The type and identifier URIs are realized as two different java.net.URIs, the `ClassURI` and the `IndividualURI`.

Unfortunately it is not possible to inherit from java.net.URI so the two classes just wrap the URIs and implement an interface corresponding to their type. Classes of global scope also implement these interfaces declaring access methods for their identifier and type.

All model entities also implement interfaces coming with Winter. Those interfaces declare methods used by Winter to interact with those classes to provide mapping functionalities. As mentioned Winter bases on Java annotations, providing the meta-information needed for mapping for each class. In the OAM we introduce a UML2 Profile providing the `@winter` stereotype for the Java annotation used in the Winter API. This annotation stereotype is usable on interfaces, classes and fields. In the annotation stereotype we can declare all variables of the Winter annotation, the mapping pattern, a variable name, the object type and two variable maps to declare inter pattern mapping. Depending on what is annotated the annotations schema changes. As you can see, the mapping pattern consists of one or multiple statements, each statement is a subject-predicate-object triple. Winter uses SPARQL syntax for these statements. In such a statement the predicate is fix, subject and object are variables. The type declaration declares a general object type Winter uses. Winter knows five different general types, PATTERN, GLOBALOBJECT, LOCALOBJECT, LITERAL and MAPPING. Further informations about Winter and a detailed discussion of the Winter annotation and the single variables in it could be found in Section 6.2. We attached one corresponding annotation as UML comment to one class and one property in each figure. As mentioned such annotation is realized through a profile declaring a stereotype for the annotation. We used the comment form because


«package»
Patterns
«@winter»
AnnotationPattern
«@winter» AnnotationDescription : IndividualURI
«@winter» AnnotationDescriptionType : ClassURI
«@winter» AnnotationSituation : IndividualURI
«@winter» AnnotationSituationType : ClassURI
«@winter» AnnotatedRole : IndividualURI
«@winter» AnnotatedRoleType : ClassURI
«@winter» AnnotationRole : IndividualURI
«@winter» AnnotationRoleType : ClassURI
«@winter» AnnotatedIO : Image
«@winter» Annotation : GeoPoint
«@winter»
DecompositionPattern
«@winter» DecompositionDecription : IndividualURI
«@winter» DecompositionDescriptionType : ClassURI
«@winter» DecompositionSituation : IndividualURI
«@winter» DecompositionSituationType : ClassURI
«@winter» CompositeRole : IndividualURI
«@winter» CompositeRoleType : ClassURI
«@winter» ComponentRole : IndividualURI
«@winter» ComponentRoleType : ClassURI
«@winter» Composite : Presentation
«@winter» Component : Image
«@winter»
DataValuePattern
«@winter» Image : Image
«@winter» Quality : GeoLocationQuality
«@winter» Region : GeoPoint

```
Pattern Class Annotation:
@winter{
pattern = "?AnnotationDescription <rdfType> ?AnnotationDescriptionType.
   ?Annotationsituation <rdfType> ?AnnotationSituationType.
   ?AnnotatedRole <rdfType> ?AnnotatedroleType.
   ?AnnotationRole <rdfType> ?AnnotationRoleType.
   ?AnnotationDescription <defines> ?AnnotatedRole.
   ?AnnotationDescription <defines> ?AnnotationRole.
   ?AnnotatedRole <classifies> ?Image.
   ?AnnotationRole <classfies> ?GeoPoint.
   ?Image <hasSetting> ?AnnotationSituation.
   ?GeoPoint <hasSetting> ?AnnotationSituation.
   ?AnnotationSituation <satisfies> ?AnnotationDescription";
type = winter.type.PATTERN;
}
```

```
Global Field Annotation:
@winter{
var = Image;
type = winter.type.GLOBALOBJECT;
}
```

Figure 5.4: The Pattern Package

the used UML2 Editor does not visualize the whole information in the stereotype. We attached the comments only to single entities in the figures but as you can see all classes are stereotyped. We attached a comment to an arbitrary class and property in each figure to make clear how to stereotype in the OAM.

5.3.2 API Customizations in the OAM

Apart from the language independence, one of the main advantages coming with the use of an intermediate API model like the OAM is the opportunity to customize the the API model before code generation. Within such customization it is possible to import foreign API classes and enrich them with semantic information. We could customize our OAM for the running example in a way that the image class is realized as subclass of an arbitrary image class implementation from another API. In case of Java such an API could be for example AWT[1]. When the image class in our OAM would be declared as subclass of java.awt.image, this subclass implements the image interfaces defined in the interface package of our OAM. Apart from that it implements all Winter related interfaces. We suggest to encapsulate such concrete implementations in an extra package in the OAM and later in the API. With this mechanism we are able to generate customized classes inheriting functionality from widely used implementations and providing functionalities for our domain, RDF persistence. Figure 5.5 shows such a scenario for an image class.

[1]abstract Window Toolkit a GUI API for Java `http://en.wikibooks.org/wiki/Java_Swings/AWT`

In this figure you can see that the `InformationRealization` class inherits from an arbitrary image class (this could come from any foreign java library) and implements the `InformationRealization` interface. The new class `InformationRealization`retains all functionalities from the image class it inherits from and gains semantic data persistence with implementing the methods in the interface.
But this is not the end of what is possible with customization. For images for instance it makes sense to store the semantic information in a triple store and to store the image itself in a RDB with the object URI as primary key. This requires additional object persistence to RDBs. This additional persistence could also be modeled in the OAM and a customized code generation could generate full featured APIs from such a model.

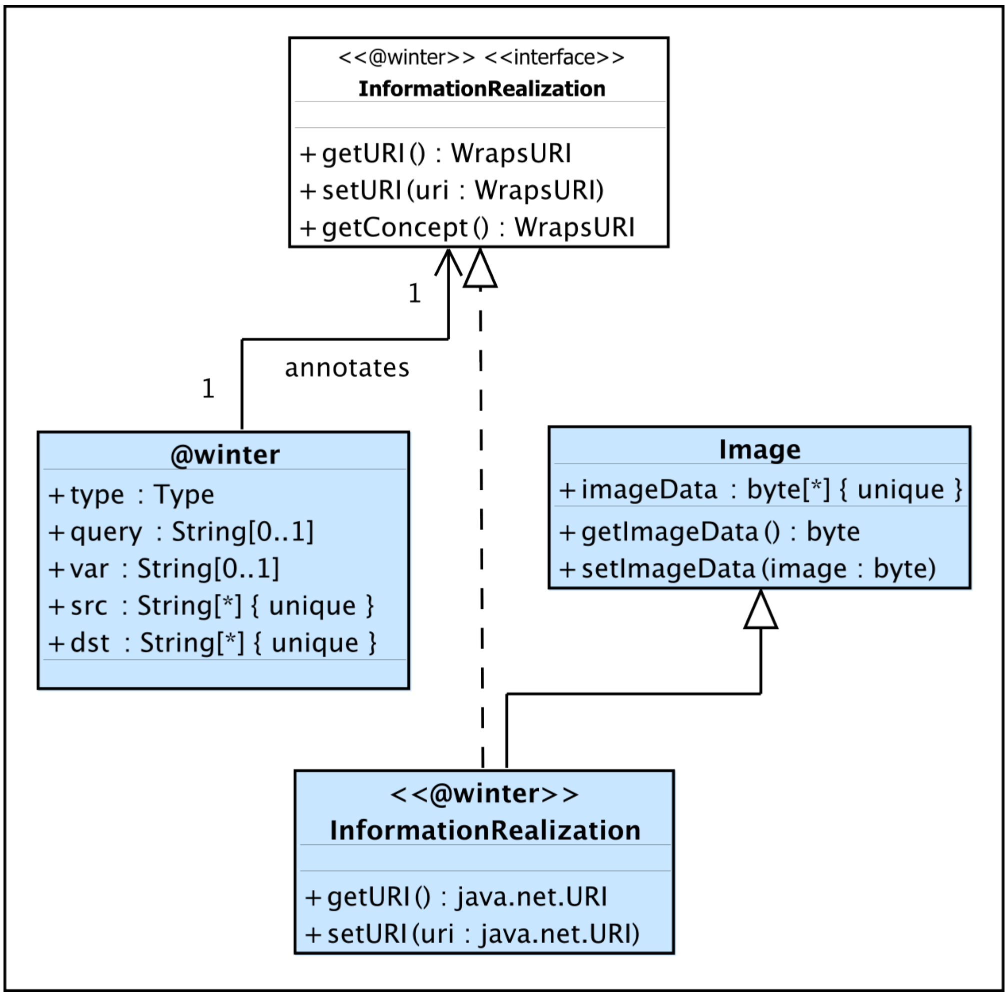


Figure 5.5: A Model of an customized Image Class

6 Implementation

We implemented a prototype of our system as a set of plugins based the Eclipse Modeling Framework (EMF)[1]. Here we introduction all implementation related technologies and discuss the single parts of our implementation. The EMF and the Graphical Modeling Framework (GMF), discussed in Section 6.1 build the basis of our implementation. EMF provides the UML2 implementation for our two models and the transformation process operates on the models in EMF. Additional to this we give a detailed introduction to Winter in Section 6.2. We discuss its features and how it works in detail because our prototype implementation uses Winter as persistence layer underlying the generated APIs. These Winter related information is important for the concrete transformations and models in our prototype. This discussion of Winter becomes important in Section 6.3, where we define the concrete models and transformations in our prototype. Additionally to the model manipulation facilities EMF also provides JET (Java Emitter Templates) a JSP (Java Server Pages) based framework for code generation. In the last Section 6.4 we describe how to customize the code generation step to generate a Winter based ontology API from the OAM.

6.1 The Eclipse Modeling Framework and the Graphical Modeling Framework

With the EMF, Eclipse provides a powerful and flexible basis for application development through the pragmatic modeling and code generation facilities. The EMF combines features from several different modeling projects based on Eclipse. EMF is the top level projects of those sub projects around modeling. In EMF a model is specified in ECore a modeling language based on XML Metadata Interchange (XMI) [21]. EMF provides facilities for code generation, graphical diagramming, model transformations, model validation and search, just to name a few. EMF provides tools, editors and adapter classes to support viewing and command-based editing of such models. On top of ECore EMF supports multiple modeling languages like UML. The UML2 project inside of the EMF provides a full UML2 meta-model implementation based on the ECore implementation in the EMF. The ECore and the UML mata-model both base on the EMOF (Essential Meta-Object Facility) [22] standard of the OMG. EMF provides import methods for ,e.g., UML based models in formats other that the XMI based format EMF uses,e.g., IBM RationalRose. So it is possible to use multiple editors to modify models. The GMF (Graphical Modeling Framework)[2] provides basic graphical editors in

[1]The EMF Project website `http://www.eclipse.org/modeling/emf/?project=emf`last visit 03.2010
[2]The GMF Project website `http://www.eclipse.org/modeling/gmf/` last visit 03.2010

Eclipse for EMF models. Based on the GMF it is possible to build customized modeling environments tailored to a specific domain. In the concrete implementation we used the basic GMF editors but in the future it is recommended to supply customized editors especially design for this generation process. This would increase the usability of our approach and speed up the generation process. EMF provides two UML2 profiles for RDF and OWL, thereby the OWL profile bases on the RDF profile. The OWL profile is an implementation of the ODM OWL profile based on the UML2 meta-model in the EMF. This profile consists of all OWL stereotypes defined in the ODM.
We use this UML2 implementation to work with our models and we use RDF and OWL profile and the stereotypes to denote the OWL nature of the UML elements in our MoOn.

6.2 Winter

In this section we will give a short introduction to Winter and how it works. We already mentioned Winter a few times and we know that it is a annotation based object-to-statement persistence API in JAVA. In the last chapter we discussed the Winter annotation and introduced the winter types. Apart from the annotation Winter needs classes to implement some specific methods. These method build up the core functionality so that Winter is able to work with the class. To be able to access those methods in arbitrary Java classes Winter defines different interfaces, those classes have to implement. In the pink area in Figure 6.1 you can see all winter related interfaces. The RDFSerializable interface defines basic methods to register the Winter mappers in a concrete class and to access some winter related variables in the class. All classes using Winter functionalities have to implement this interface. The IdentifiedByURI and the HasConcept interface define access methods to the ConceptURI field and to the IndividualURI field. These fields are implemented by classes representing global concepts. The WrapsURI interfaces defines general access methods to wrapped URIs. In the Winter two different wrapped URIs are defined, the IndividualURI unique for every object and the ConceptURI unique for every class.

6.2.1 The Winter Annotation

Additionally Winter provides the annotation already mentioned, applicable to fields and classes.Apart from other fields the annotation declares the Winter type of the annotated field or class. Winter knows five different internal types, they are:

- **PATTERN** Annotations with type PATTERN should only be assigned to classes. They denote that this class is a ODP representation.
- **GLOBALOBJECT** Annotations with type GLOBALOBJECT could be applied to classes or fields. A class annotated with GLOBALOBJECT represents an ontology concept with global scope and thus with its own class representation. Fields annotated in this way refer to an object of a GLOBALOBJECT annotated class.

- **LOCALOBJECT** Annotations with type LOCALOBJECT could only be assigned to fields. They denote the field to contain a reference to an concept with local scope representation. A concept with local scope is not represented through a dedicated class, such a concept is represented through two wrapped URIs. The ConceptURI and the IndividualURI. A LOCALOBJECT annotated field could refer to a ConceptURI or an IndividualURI. In general these fields are always declared in pairs.
- **MAPPING** A annotation with type MAPPING applies only to fields. Such a field refers to an PATTERN annotated class.
- **LITERAL** A annotation with type LITERAL applies only to fields. These fields contain values mapped to literals in the serialization.

Additional to this and depending on the Winter type multiple other declarations could be made in the annotation. These declarations are used by Winter to serialize and deserialize annotated objects. For this concern the Winter annotation defines different fields. The following list gives an overview over the annotations fields and their function.

- **type** The type field declares the annotated to be of one of the five Winter types described above..
- **pattern** This field contains a statement pattern in SPARQL. This Pattern is used by Winter for the mapping. The pattern field applies to annotations of type PATTERN, GLOBALOBJECT, LOCALOBJECT and LITERAL. If a class is annotated Winter matches all variables names declared in the SPARQL statements against the var field in the annotated fields declared in the class. Class annotations always need an SPARQL pattern. In case of an field annotation the SPARQL pattern is optional. If the variable name in the annotation of the field (the var field) is present in the SPARQL Pattern of the declaring class we do not need a SPARQL pattern in the fields annotation. This case indicates fro Winter that the field is mandatory to the class. This has direct influence on the mapping behavior of Winter. The field has to be set when the object should be mapped, otherwise winter won't map the object. Otherwise then the variable name in the annotation of the field is not present in the SPARQL pattern of the declaring class annotation the field is optional to Winter. We need a SPARQL pattern in the annotation of the field. If optional fields are not set only they won't be mapped but this does not influence the mapping of the whole object. Only the single field is affected. If the field is set it would be mapped according to the SPARQL pattern in its own annotation. Variables other that the fields annotation variable will be substituted from the corresponding fields in the object.
- **var** This field of the annotation defines the variable name in the SPARQL statements to substitute with the content of this field. The var field only applies in field annotations and only in LOCALOBJECT, GLOBALOBJECT and LITERAL typed annotations.

- **src/dst** The src field is strongly related to the dst field. Both are collections of var names like in the var field. They should have the same length and it makes no sense to define one without the other. The src and dst field represent a variable name to variable name mapping. Each of the sequential field in one of the collections has a corresponding field in the other collection. It is used to map variable names between the declaring class and a field referencing a PATTERN annotated class. The src contains the variable names of the declaring class and the dst field those of the referenced class. The values in the referenced class are are replaced through those of the declaring. With this functionality it is possible to declare patterns like the ProvenanceInformationPattern, see Section 2.4

6.2.2 Winter related Interfaces and classes

As denoted Winter defines four different interfaces to be implemented by classes using Winters mapping facilities. These interfaces declare general methods used by Winter to access the URIs and other variables in serialization and deserialization.

- **RDFSerializable** The RDFSerializable interfaces defines methods to register the different Winter mappers to a class. Additionally it defines getter/setter methods for Winter related fields recommended to a class. Every class, global and pattern, using Winter functionalities has to implement these interfaces.
- **HasConcept** The HasConcept interface a the `getConcept()` function to access the concept URI in a global class.
- **IdentifiedByURI** The IdentifiedByURI interface defines get and set methods for the IndividualURI of a global class.
- **WrapsURI** The WrapsURI interface is used in the definition of the ConceptURI and the IndividualURI.

Additional we defined two classes representing the wrapped URIs used by Winter, the ConceptURI and the IndividualURI. These URIs are used by global concept classes and in the declaration of local concept fields. Both classes realize the WrapsURI interface.

- **ConceptURI** The ConceptURI class represents a wrapped concept/class URI. Such an URI is the unique identifier of each concept/class.
- **IndividualURI** The IndividualURI class represents a wrapped individual/instance URI. Such an URI is the unique identifier of each individual/instance.

6.2.3 Mapping Objects to Statements with Winter

In a concrete mapping Winter builds up a SPARQL query from the class or field annotation. While mapping a whole class Winter builds this query from the pattern field in the class annotation. Depending on the performed operation, serialization or deserialization, Winter substitutes all or some variables in the query with values from the corresponding class fields. In case of serialization Winter could build RDF statements from this binded query and add them to the triple store. In case of deserialization Winter performs the query to get the requested values back. If we refer to the Annotation pattern in our running example this mechanism could be understood easily. Mapping the Annotation pattern instance described, results in these RDF statements.

Listing 6.1: `AnnotationPattern` RDF statements

```
eaad              <rdfType>         AnnotationDescription.
eaac              <rdfType>         AnnotatedConcept.
image-2           <rdfType>         Image.
glp-1             <rdfType>         EXIFGeoParameter.

geo-location-1    <rdfType>         GeoPoint.

eaas              <rdfType>         AnnotationSituation.

eaas              <dul:defines>     eaac.
eaad              <dul:defines>     glp-1.
eaac              <dul:classifies>           image-2.

glp-1             <dul:classifies>           geo-location-1.
geo-location-1    <geo:long>        "40,76"^^xsd:decimal.
geo-location-1    <geo:lat>         "-73,99"^^xsd:decimal.
geo-location-1    <hasSetting>      eaas.

image-2           <hasSetting>      eaas.
eaas              <satisfies>       eaad.
```

These statements are build from the pattern in the annotation of the Annotation pattern class, shown in 5.4 and Listing 6.2. For mapping Winter will substitute all variables in the SPARQL pattern through the corresponding fields of the declaring object. As example for a field with refering to a global scope object look at lines 33-37 of Listing 6.2, the declaration of the image field in the annotation pattern class. Local scope concepts are represented through pair of fields declaring the ID and type. For the AnnotationDescription you can find these field declarations in Listing 6.2 in lines 17-29. As you can see the values in the var fields of the annotations are referring to the corresponding variables in the SPARQL pattern. Optional fields come with their own SPARQL pattern in the annotation. If such a field is not empty it is mapped subsequently when the declaring object is mapped.

Listing 6.2: `AnnotationPattern` class annotation

```
1  @winter{
2  pattern = "?AnnotationDescription <rdfType> ?AnnotationDescriptionType.
3             ?AnnotationSituation <rdfType> ?AnnotationSituationType.
4             ?AnnotatedRole <rdfType> ?AnnotatedRoleType.
```

```
            ?AnnotationRole <rdfType> ?AnnotationRoleType.
            ?AnnotationDescription <dul:defines> ?AnnotatedRole.
            ?AnnotationDescription <dul:defines> ?AnnotationRole.
            ?AnnotatedRole <dul:classifies> ?Image.
            ?AnnotationRole <dul:classifies> ?GeoPoint.
            ?Image <dul:hasSetting> ?AnnotationSituation.
            ?GeoPoint <dul:hasSetting> ?AnnotationSituation.
            ?AnnotationSituation <dul:satisfies> ?AnnotationDescribtion";
type = winter.type.PATTERN;
}
public class AnnotationPattern implements RDFSerializable{

        @winter{
        var = "AnnotationDescription";
        type = winter.type.LOCALOBJECT;
        }
        private IndividualURI annotationDescription =
                "http://www.example.de/AnnotationDescription#1";

        @winter{
        var = "AnnotationDescriptionType";
        type = winter.type.LOCALOBJECT;
        }
        private ConceptURI annotationDescriptionType =
                "http://www.example.de/AnnotationDescriptionType#1";
                        .
                        .
                        .
        @winter{
        var = "Image";
        type = winter.type.GLOBALOBJECT;
        }
        private Image image;
                        .
                        .
                        .
}
```

You might recognize that the type declarations statements of the `?Image` (`?Image <rdfType> ?ImageType`) and the `?Geopoint` (`?GeoPoint <rdfType> ?GeoPointType`) are not present in the SPARQL pattern of the annotation but in the statements of the serialization. These two concepts are of global scope and thus are represented through their own classes. Both classes are annotated and in this annotation the type declaration is defined. A annotation on such a class would look like this.

Listing 6.3: `Image` class annotation

```
@winter{
pattern = "?Image <rdfType> ?ImageType";
type = winter.type.GLOBALOBJECT;
}
private class Image implements HasConcept, IdentifiedByURI, RDFSerializable{

        @winter{
        var = "Image";
        type = winter.type.LOCALOBJECT;
        }
        private IndividualURI id =
                "http://www.example.de/Image#1";

        @winter{
        var = "ImageType";
        type = winter.type.LOCALOBJECT;
        }
        private ConceptURI concept =
                "http://www.example.de/ImageType#1";
        .
        .
        .
}
```

For object manipulation all getter/setter methods for annotated fields refer directly to the triple store using Winter functionalities. So performing a get on an annotated field always results in a query against the triple store and a set results in the replacement of one or multiple statements in the store. Listing 6.4 shows the getter/setter methods for the local field annotationSituation in the AnnotationPattern class.

Listing 6.4: Getter/Setter for `AnnotationDescription`

```
@winter{
public IndividualURI getAnnotationDescription() {
  try {
   if(readmapper != null)readmapper.updateOBJECTField
    (AnnotationPattern.class.getDeclaredField("annotationDescription"), this)
  } catch (SecurityException e) {
   ...
  }
  return annotationDescription;
 }

 public void setAnnotationDescription(IndividualURI annotationDescription) {
  if(writemapper != null){
   try {
    Field field =
     AnnotationPattern.class.getDeclaredField("annotationDescription");
    if (annotationDescription != null){
     Object newObj = annotationDescription;
     writemapper.replaceObject
      (field.getAnnotation(winter.class), this, newObj);
     }else{
      writemapper.deleteObject(field.getAnnotation(winter.class), ... );
     }
   } catch (NoSuchFieldException e) {
   ...
   }
  }
  this.annotationDescription = annotationDescription;
 }
```

6.3 The Mapping from MoOn to OAM

For the model transformation in our prototype we decided not to use the model transformation language comming with EMF, namely the Atlas Transformation Language (ATL)[3] provided by the EMF. We decided to implement our transformation on the basis of the adapter classes for UML2[4] and ECore[5]. Because with this APIs we are more flexible regarding the upcoming code generation process. Based on these APIs we developed an iterative transformation process between MoOn and OAM. The transformation combines multiple iterations over the MoOn to transform it to the OAM. These iterations are :

1. In a first iteration we generate the SPARQL patterns for every ODP and every named global concept in the MoOn. These patterns are used in the Winter annotation of the classes and fields in the OAM and later in the generated API.
2. In a second step we filter all global concepts and generate interface and/or class representations for them in the OAM. We can easily identify them by the mentioned MoOn stereotype `winterGlobalObject`.
3. In the last step we generate class representations for the single patterns in the MoOn. As mentioned every single pattern resides in its own `owl:ontology` stereotyped

[3]The ATL Project inside Eclipse `http://www.eclipse.org/m2m/atl/` last visit 05.03.2010
[4]UML2 API Javadoc `http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/topic/org.eclipse.uml2.doc/references/javadoc/org/eclipse/uml2/package-summary.html`
[5]ECore API Javadoc `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/ecore/package-summary.html`

package in the MoOn. So we build a class representation for each of these pattern packages.

Additional to the profiles for RDF/OWL we also provide profiles and models for some of the Winter related features. We provide an extra profile realizing a stereotype for the Winter annotation. All the generated classes are stereotyped with the Winter annotation stereotype as long as they need Winters mapping functionalities. Winter defines several interfaces implemented by classes using Winter features. We defined models for those Winter interfaces. The generated models in the OAM implement the corresponding Winter interfaces. All generated classes and interfaces are organized the different package like described in Section 5.3.1.

6.3.1 The Winter Profile

To encapsulate the Java annotation needed by Winter for the mapping functionalities we defined a new profile, the Winter profile. In the Winter profile we define a stereotype for the Java annotation of Winter. As mentioned above this annotation consists of 5 different fields:

- The pattern field, a String field that can contain the statement pattern in SPARQL
- The var field, a String field that can contain the variable name from the query pattern.
- The src field, a String collection containing variable names from the SPARQL pattern for inter pattern matching in combination with the dst field.
- The dst field, in combination with the src field this field builds a name to name map for variable matching in inter pattern matching.
- The type fields, denotes the annotated as one of the five Winter types.

According to the concrete needs of the class/interface or field the stereotype should be attached and the appropriate fields of the annotation should be filled. For example, if we annotate a pattern class, like the annotation Pattern in in Figure 5.4, the annotation will consist of a pattern and a type declaration. All generated classes and fields in the OAM are stereotyped automatically with a stereotype instance generated specially for this class or field. In this stereotype instance we use if intended the corresponding SPARQL pattern generated in the first step of the mapping process.

6.3.2 Winter Models

To simplify code generation we defined models for the interfaces and URI classes defined by Winter. The interfaces models are realized by all classes using Winter, in the OAM. We need these models in the code generation step. The URI models are used to declare the types of local concept fields in global and pattern classes in the OAM. You can see this in Figure 5.3 and Figure 5.4 unfortunately the editor does not visualize the interface implementations. In Figure 6.1 we visualize the interfaces implementation structure in the OAM using the `InformationRealisation` pattern as example. In the pink area the figure the classes and in the and in the light green area the interfaces they implement.

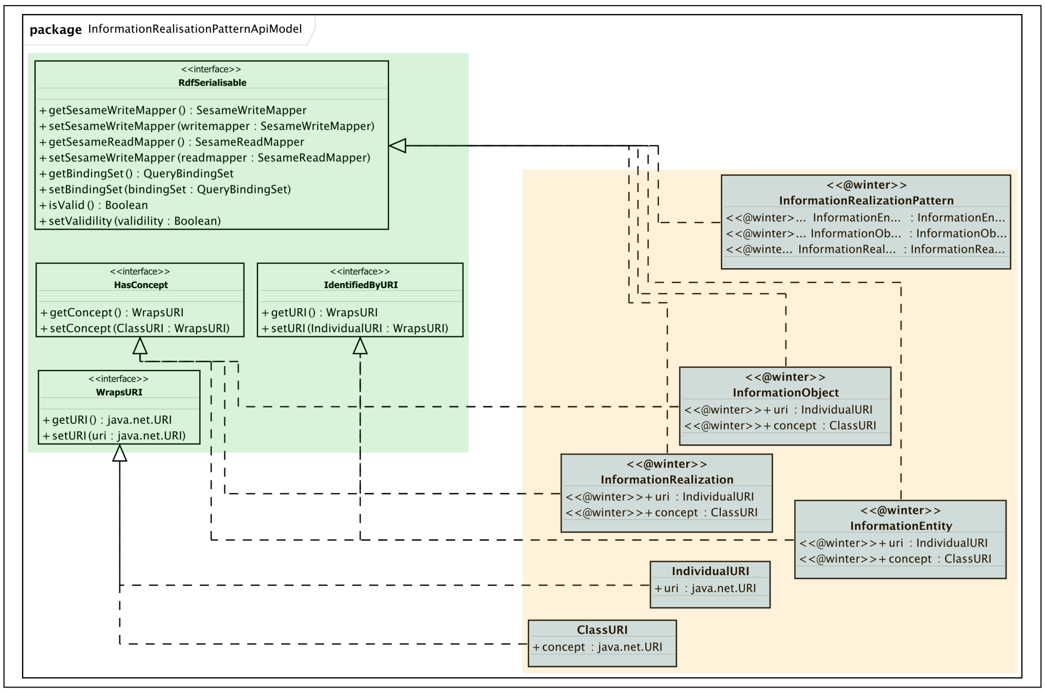

package InformationRealisationPatternApiModel
<<interface>>
RdfSerialisable
+getSesameWriteMapper() : SesameWriteMapper
+setSesameWriteMapper(writemapper : SesameWriteMapper)
+getSesameReadMapper() : SesameReadMapper
+setSesameWriteMapper(readmapper : SesameReadMapper)
+getBindingSet() : QueryBindingSet
+setBindingSet(bindingSet : QueryBindingSet)
+isValid() : Boolean
+setValidility(validility : Boolean)
<<interface>>
HasConcept
+getConcept() : WrapsURI
+setConcept(ClassURI : WrapsURI)
<<interface>>
IdentifiedByURI
+getURI() : WrapsURI
+setURI(IndividualURI : WrapsURI)
<<interface>>
WrapsURI
+getURI() : java.net.URI
+setURI(uri : java.net.URI)
<<@winter>>
InformationRealizationPattern
<<@winter>... InformationEn... : InformationEn...
<<@winter>... InformationOb... : InformationOb...
<<@winte... InformationReal... : InformationRea...
<<@winter>>
InformationObject
<<@winter>>+uri : IndividualURI
<<@winter>>+concept : ClassURI
<<@winter>>
InformationRealization
<<@winter>>+uri : IndividualURI
<<@winter>>+concept : ClassURI
<<@winter>>
InformationEntity
<<@winter>>+uri : IndividualURI
<<@winter>>+concept : ClassURI
IndividualURI
+uri : java.net.URI
ClassURI
+concept : java.net.URI

Figure 6.1: The Information Realization API Model

6.3.3 The Mapping for the running Example

In the previous chapter we described the OAM for our running example. Based on the described of the MoOn and the OAM we describe the mapping between the two models in this section. We refer to the concrete models used in our running example. Section 4.3 describes a MoOn for our running example. We start from this MoOn and discuss how we end up in the OAM for the running example. As decribed above the transformation process is iterative process with actually three iterations. In the first iteration we analyze the packages in the model and generate SPARQL patterns for the patterns and classes. We only have to generate SPARQL patterns for the classes with global scope. We can easily identify them by the applied stereotype ≪winterGlobalObject≫. A SPARQL pattern for global class usually consists of an type declaration of the class and the declarative statements for the mandatory properties of the particular class. In the case of a whole pattern the SPARQL pattern represents all mandatory relationships in the pattern. If we take a look at the annotation pattern, such a SPARQL pattern will look like the one defined in Figure 5.4. We analyze the outgoing association for every class in the pattern. If the association is mandatory and has a minimal cardinality above null, the relationship presented by this association has to be in the SPARQL pattern for

the whole pattern. In such a case we would add a statement in the form subject (the class the association goes out), predicate (attached to the association) and object (the class associated) to the SPARQL pattern. In the Annotation Pattern in our running example In the second step we filter all global concepts from all patterns, in this case `Image`, `Presentation`, `Quality` and `Region`. We build class representations for all these concepts. All of these classes implement the related interfaces as defined in Section 6.3.2 and Figure 6.1. As you can see in Figure 5.3 all those classes are annotated. According to the definitions made in Section 6.2 an annotation on a global class definition will consist of a type declaration, here GLOBALOBJECT and the SPARQL pattern, generated in the previous step.
In the last step we generate class models for our patterns. Similar to the global object classes the pattern classes are annotated. In this annotation of type PATTERN we use the SPARQL pattern generated in the first step. A pattern class defines fields for the global objects referenced in the pattern, additionally it also defines pairs of fields for all local objects in the pattern representation in the MoOn. All generated fields, that should be mapped, are annotated. Mandatory fields are only annotated with type and variable name, optional field are also annotated with a SPARQL pattern.
After this last step we have created an OAM like in Figures 5.1 to 5.4 from the MoOn presented in Figures 4.1 to 4.4.

6.4 Code generation and Java Emitter Templates

The last step of our work-flow is the code generation. We decided to use JET (Java Emitter Template) for code generation. JET uses a template technology which is very closely related to the Syntax of Java Server Pages (JSPs). The JET engine in the EMF can be used to generated any kind of output based on a model, in our case we want to generate Java code from the OAM. According to the Jet FAQ [1] it is not a good practice to generate code directly from the UML2 model. It is recommended to implement a intermediate model tuned to the structure of the output. For the best practice it is recommended to:

- Separate the concerns of abstract code representation for the user from the concerns of code generation.
- Create an intermediate model specific to code generation, that represents concepts needed to generate the intended code artifacts.
- Build a Jet transformation from the intermediate model to code.
- If necessary the intermediate model and the to-code transformation can be managed by wizards or other UI tools.

The internal language of our UML2 models is XMI, but the model structure is very complex and thus the XMI file is overloaded with information useless for our code generation process. We decided to transform the UML2 model in a simplified XML representation holding all information necessary for the code generation. The Jet templates using this intermediate model as input and generate Java code from it. So for the AnnotationPattern class this XML declaration for could look like in this listing.

Listing 6.5: AnnotationPattern Class XML Declaration
```
<app class="AnnotationPattern">
  <ann isann="True"
    type="winter.type.PATTERN"
    pattern="?AnnotationDescription <rdfType> ?AnnotationDescriptionType.
    ?AnnotationSituation <rdfType> ?AnnotationSituationType.
    ?AnnotatedRole <rdfType> ?AnnotatedRoleType.
    ?AnnotationRole <rdfType> ?AnnotationRoleType.
    ?AnnotationDescription <dul:defines> ?AnnotatedRole.
    ?AnnotationDescription <dul:defines> ?AnnotationRole.
    ?AnnotatedRole <dul:classifies> ?Image.
    ?AnnotationRole <dul:classifies> ?GeoPoint.
    ?Image <dul:hasSetting> ?AnnotationSituation.
    ?GeoPoint <dul:hasSetting> ?AnnotationSituation.
    ?AnnotationSituation <dul:satisfies> ?AnnotationDescribtion"
  />

  <property name="AnnotationDescription" type="IndividualURI">
    <ann type="winter.type.LOCALOBJECT" var="AnnotationDescription" />
  </property>
  <property name="AnnotationDescriptionType" type="ConceptURI">
    <ann type="winter.type.LOCALOBJECT" var="AnnotationDescriptionType" />
  </property>
    .
    .
    .
  <property name="Image" type="m3o.interfaces.Image">
    <ann type="winter.type.GLOBALOBJECT" var="Image" />
  </property>
</app>
```
With the Jet template in the next listings we are able to generate Java code from XML declaration of pattern classes.. The part of the template shown in Listing 6.6 generates a class header with the name given in the declaration. The next part of the template in Listing 6.7 iterates over all the properties defined in the generation an generates the field declarations from the properties. If the fields are annotated it adds Winter annotations to the field declaration. The next part in Listing 6.8 generates the constructor for the class, in this case a constructor that expects all defines properties as arguments. In the last Listing **??** you can see the part responsible for the getter setter methods.
In the first statement of this listing 6.6 we check if the concrete class in the XML declaration should be annotated or not. For this concern we check the `isann` variable

in the declaration in line 3. If the concrete class should be annotated the `@winter` annotations would be created. After that line 10 creates the class header.

Listing 6.6: Jet Template Header for Class generation

```
1  <c:iterate select=”/app/class” var=”c”>
2    <c:choose select=$p/@isann>
3      <c:when test=”’True’”> @winter{
4        pattern = ”<c:get select=”$p/@pattern” />” ,
5        type = <c:get select=”$c/@type” />,
6        var = ”<c:get select=”$c/@var” />” ,
7        src = ”<c:get select=”$c/@src”>” ,
8        dst = ”<c:get select=”$c/@dst”>”}
9      </c:when>
10   </c.choose>
11    class <c:get select=”/app/@class” /> implements RDFSerializable {
```

In this listing we iterate over all properties in the XML class declaration, with the `c:iterate` statement in Line 11. We generate field declarations for each property. If the property is annotated we add an `@winter` annotation to the field declaration. Line 22 generates the concrete field declaration from the current property in the XML class description.

Listing 6.7: Jet Template Part for Field generation

```
12   <c:iterate select=”/app/property” var=”p”>
13     <c:choose select=$p/@isann>
14       <c:when test=”’True’”> @winter{
15         pattern = ”<c:get select=”$p/@pattern” />” ,
16         type = <c:get select=”$p/@type” />,
17         var = ”<c:get select=”$p/@var” />” ,
18         src = ”<c:get select=”$p/@src”>” ,
19         dst = ”<c:get select=”$p/@dst”>”
20       </c:when>
21     </c.choose>
22      private <c:get select=”$p/@type” /> <c:get select=”$p/@name” />;
23   </c:iterate>
```

This listing generates the constructor for the class. Line 24 generates the header of the constructor. The following lines 25-28 iterate over all properties and add them as arguments to the constructor. In lines 30-32 the body of the constructor is created by iterating over all properties again and add the corresponding assignments.

Listing 6.8: Jet Template Part for Constructor generation

```
24    public <c:get select=”/app/@class” />(
25   <c:iterate select=”/app/property” var=”p”>
26     <c:get select=”/$p/@type” />
27      <c:get select=”/$p/@name” />,
28    </c:iterate>
29    ){
30    <c:iterate select=”/app/property” var=”p”>
31      this.<c:get select=”$p/@name” /> = <c:get select=”$p/@name” />;
32    </c:iterate>
33    }
```

In this last listing we generate getter/setter methods. Line 34-39 generates the setter, lines 34-36 the header of the setter method and line 37-39 the assignment in the body of the setter method.Lines 41-44 are for the the getter method. As you can see line 41 and 42 are for the header of the method and line 43 the body with the return statement.

Listing 6.9: Jet Template Part for Getter/Setter generation

```
34     public void set<c:get select=\”camelCase($p/@name)” />(
35    <c:get select=”$p/@type” />
36    <c:get select=”$p/@name” />) {
37      this.<c:get select=”$p/@name”/> =
38     <c:get select=”$p/@name” />;
39    }
40
41     public <c:get select=”$p/@type”>
42     get<c:get select=\”camelCase($p/@name)” />(){
43     return <c:get select=”$p/@name” />;
44    }
45  }
```

We use different Jet templates for the different class and interface types. One of the pattern classes, one for the implementation and one for the interface definitions. The concrete generation process is evoked from a Eclipse plugin. Initially this plugin generates the different XML class declarations for all the classes present in the OAM. Then the plugin provides a wizard driven generation process that gives the user the possibility to generate all classes based on the XML class declarations. It is intended to generate interfaces, pattern classes and concrete implementations to different java packages.

7 Conclusion and further work

7.1 Conclusion

Our goal was to develop an model-driven API generation process to support the implementation of ontology APIs from the schema description in pattern based ontology. To fulfil this and to enable the process to support ODPs we analyse ontology/ODP related object persistence in Section 3 and the influence of ontology/ODP structures and technologies on the intended API implementation. For the generation process we decided to introduce two intermediate models, the MoOn and the OAM, these two models are UML2 based.
The motivation to introduce the MoOn as starting point for our generation process was to give the user a visualization and easy to manipulate view on his ontology. The Moon enables the user to specialize and customize the ontology or single ODPs in order to adapt them to the intended domain. Less axiomatized ontologies could lead to problems in the API generation because the object.oriented world is strongly axiomatization. The MoOn also enables us to implements APIs from less axiomatized ontology. We discuss this in Section 4.
In next Section 5, we describe the motivation and specification behind the OAM. The OAM is an extended UML2 Class Diagram used to visualize, customize and manipulate the structure of the API to generate. The OAM serves us as programming language and persistence layer independent model of the API to generate. Additional to this we discuss the design of ontology APIs in general and define mappings from the MoOn to the OAM. From the OAM we are able to generate the intended API in multiple programming languages. We are able to specialize the OAM to support a specific programming language. Additionally the OAM supports the customization of the API. With this customization we are able to support the special needs of specific persistence layer. Additionally we can specify single classes to derive from classes in other programming libraries.
The last Section 6 describes the implementation of a prototype of our system. We introduce relevant technologies and describe the implementation related decisions.

7.2 Further work

Our approach could benefit from the ongoing research in the multiple fields of software and ontology engineering used. In the fields of ontology engineering and especially the field of ODP based ontologies we could benefit from research on:

- A stronger formalization of the definition of ODPs. Based on an analysis of popular ontologies and patterns the idea of ODPs should be generalized and best practice for multiple ontology design problems and domains should be developed. This leads us directly to the next point.

- An ODP library for common use-cases and domains. Such a library should provide general ODPs and documentations regarding the specialization and customization for multiple design problems. It would be desirable to provide multiple domain independent ODPS. For popular domains it might make sense to provide specialization too.
- To-API mapping conventions for selected very popular ODPs, e.g., D&S based patterns. These conventions should represent the best practice in mapping such ODPs.
- Guided pattern specialization for library ODPs. This could help the untrained user in specialize a ontology suitable to his domain from general pattern he could find in a ODP library.
- Pattern origin recognition in pattern based ontologies to automatically support such to-API mapping conventions. If we are able to recognize specific patterns or their origin in a pattern based ontology, we are able to generate suggestions about a possible API structure to the user. Additional to this we would be able to support guided pattern specialization.
- Pattern recognition in non-pattern-based ontologies to support more structured and cleaner APIs for such ontologies. To recognize patterns in a non-pattern based ontology, helps us generating useful APIs for such ontologies. We would be able to abstract the diffent services that an API should provide from the ontology. Additionally we could patternize an ontology.
- Mappings for OWL constructs not covered in this work. It might be useful to support constructs like `owl:imports` or `owl:incompatibleWith` in the API generation process or even in an ontology API.
-

The research in many of the fields mentioned here is ongoing. For example, under the NeOn[1] project for example a website `http://ontologydesignpatterns.org/wiki/Main_Page` was started to collect ontologies and ODPs to build up an ontology/ODP library on the web. In the field of software engineering useful further research could care about:

- Software Design Patterns for non relational database persistence, especially triple based data. As in RDBs, where from the best practice in object persistence several patterns arose, like ORM. We suggest to develop patterns for triple persistence, especially regarding ontological data. Such a pattern should be able to deal with the problems described in our discussions about object persistence.

[1]The NeOn Project Website `http://www.neon-project.org/nw/Welcome_to_the_NeOn_Project`

- Combined Patterns to support semantic data to-triple and data to RDB mapping. Often, due to performance reasons it might be very useful to store semantic data and relational data in different stores. So it might be useful to have a combined pattern supporting triple persistence for semantical data and relational persistence for the rest. For example if we have a image class with semantic data we want the semantic in the triple store and the image in a relational data base.

- Research on best practice regarding ontology API design. Formal models for efficient ontology APIs and their use could help in developing such API in the future.

The implementation of our approach could also be improved in multiple ways.

- Develope plugins for ontology editors like Protégé 4[2] or the NeOn Toolkit[3].

- A UML2 modeling toolkit especially aligned to the visual ontology language used in the MoOn could significantly speed up the API generation process and could help users not so common to ontology engineering in developing or customizing their ontologies.

- Wizards that help the user through the generation process and support user-driven decisions. These wizards should guide users not familar with ontologies through the generation process and especially support him in ontology related questions. The implementation of such wizards could benefit from the ongoing research in the fields of ODP and especially ODP recognition.

- Mappings and generators for multiple programming languages and multiple persistence layers. This would make our application more flexible and useful.

- Automated model generation from code to simplify the integration of existing foreign APIs in the OAM. If we are able to generate an class/interface model adequate to customize the OAM directly from the class declaration in the concrete programming language, this would simplify and speed up the process of API customization.

Conceivably, in the future a developer with the need of representing semantic data in his application could visit such an ontology/ODP library in the net or his development tool includes such a library. There he gets some suitable ontologies or ODPs for his task and domain. Then he modifies them a bit with a visual and easy to use ontology development toolkit or editor. Finally he generates an API suitable for his application. But such a work-flow needs strong formalization of the underlying technologies to avoid misunderstandings, especially if the user is untrained in ontology related technologies.

[2]The Protég'e Project Website `http://protege.stanford.edu/`
[3]NeOn Toolkit Project Website `http://neon-toolkit.org/wiki/Main_Page`

A Appendix

A.1 The M3O Patterns

We want to give the reader a more concrete look on the M3O. This should give you an idea what we mean with our explanations of patterns and where they come from. Especially this should help you to understand how the declarations we made in 2.2 and the abstract patterns we introduced in 2.3.1 are translated to concrete ODPs. In the graphical representation of the patterns below all the classes coming from other ontologies, like DUL and are not specialized for their use in the pattern are white, only the blue classes are newly derived for the M3O or Event-Model-F and usually inherit from general DUL concepts.

A.1.1 The Provenance Information Pattern

The **Provenance Information Pattern** (PIP) was not defined in the original proposal of M3O [30], it is rather an result of the continuous evolution which these ontologies are subjected to. As you easily can see in figure A.1, taken from the original proposal of the M3O, the provenance information is modeled inside of the pattern. Due to the fact that this is common to many of the patterns, we decided to demerge this provenance information structure to an independent pattern. The Provenance Information Pattern bases on the D&S Pattern shown in 2.3.1. This pattern could be attached to other D&S based patterns. Via appending the Provenance Information Pattern we achieve an detailed representation of provenance information about the contextualized view of the enriched pattern. Such provenance information could be a description of the method used to create the view or the creator. As in the D&S Pattern, the Provenance Information Pattern classifies method and creator the by the concepts. In the Provenance Information Pattern the role and the concept are defined by the Description. This Description could be unique to the Provenance Information Pattern or can come with the enriched pattern depending on the use of the Provenance Information Pattern. The method itself and the entities have their settings in the Situation. Depending on the concrete realization, like in the case of the Description, the Situation in the PIP and the enriched pattern are equivalent or not. Now that we have defined the Provenance Information Pattern, shown in A.2, we have two different scenarios on how to tailor it to an other Pattern. On the one hand provenance information could be understood as additional information depending on the conceptualized view, on the relations of the set of individuals. Under this assumption the Provenance Information Pattern always refers to a specific other Pattern, which is enriched by it. In this case there is no possibility of

an stand-alone existence of a Provenance Information Pattern and the `Description` and Situation in the Provenance Information Pattern are congruent with the `Description` and `Situation` of the enriched Pattern, see figure A.3.
On the other hand the provenance information can be modeled in an independent pattern and attached to the pattern to enrich by using the mechanism described in the Interpretation Pattern. Thereby the Provenance Information Pattern brings its own conceptualized view there only the `ProvenanceSituation` of the Pattern `isObjectIncludedin` in the `Situation` of the enriched Pattern. There might be cases there this kind of modeling makes sense and there is an ongoing discussion on this issue in our workgroup. In most cases we want the dependency between the Provenance Information Pattern and the conceptualized view of the enriched pattern, like it is shown the first case in A.4. It has to be said that the attachment of the independent Provenance Information Pattern to the Annotation Pattern, like in A.4 does not makes real sense and should only be seen as an example of the underlying mechanism.

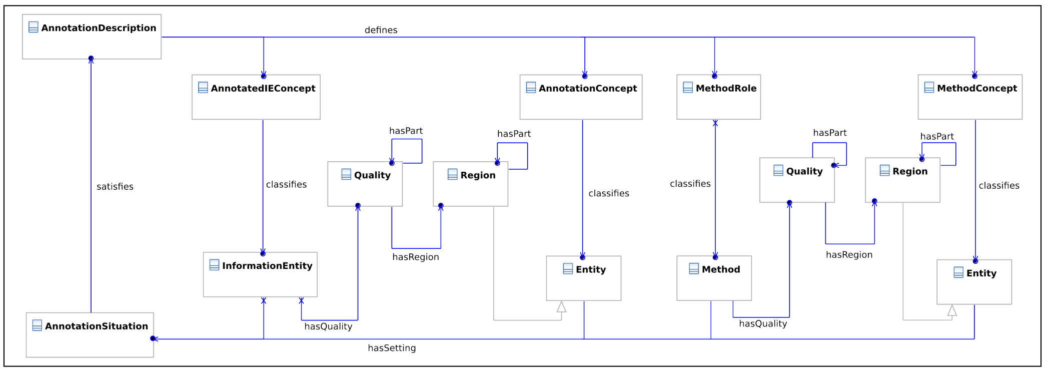


Figure A.1: The original Annotation Pattern

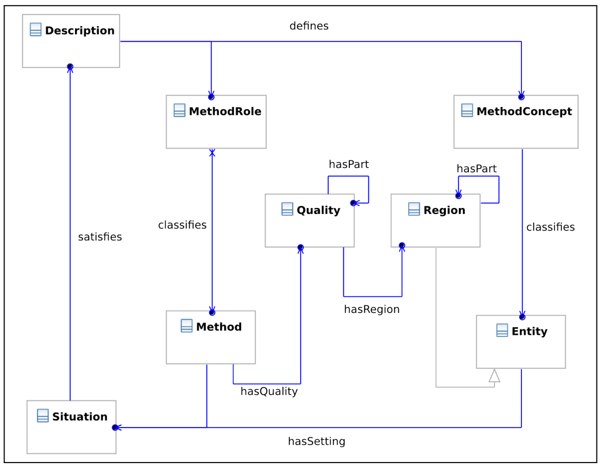


Figure A.2: The Provenance Information Pattern

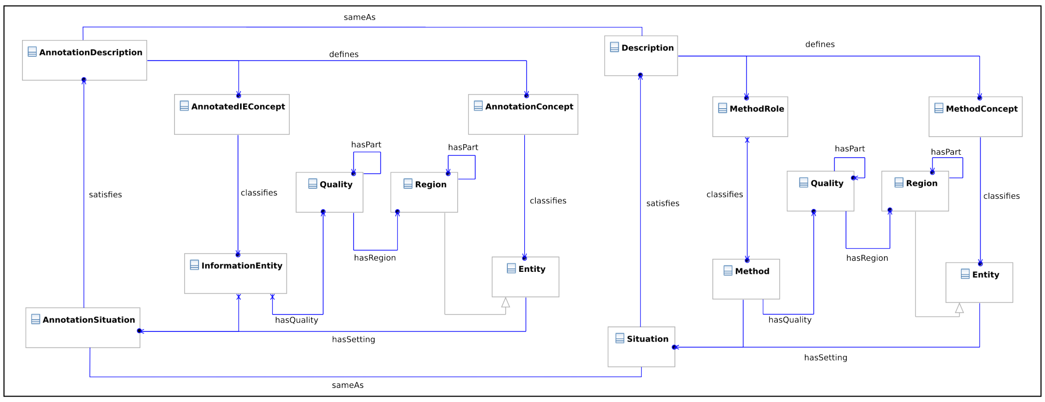


Figure A.3: The Annotation Pattern enriched with dependent Provenance Information

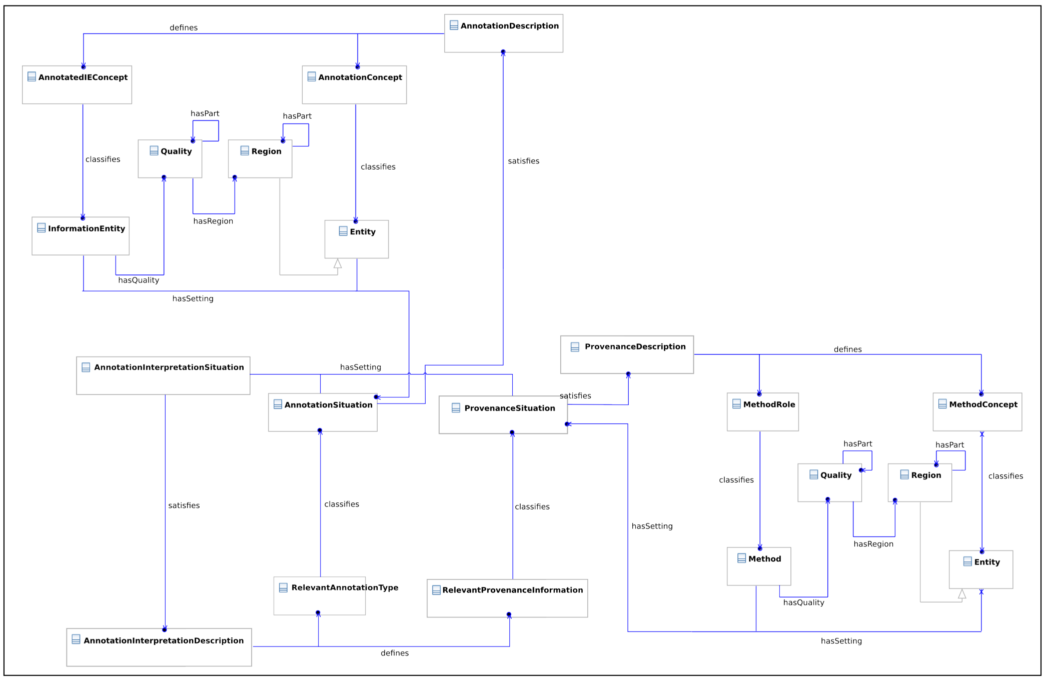


Figure A.4: The Annotation Pattern enriched with independent Provenance Information

A.1.2 The Annotation Pattern

The **Annotation Pattern** allows us to assign any annotation to an information object considering the context. It is specialized from the D&S Pattern, see Figure 2.2, and consists of an `AnnotationDescription` and a `AnnotationSituation`, whereas the `Description` defines at least one `AnnotatedInformationEntityConcepts` that classifies the `InformationEntity` to be annotated. The annotation itself is realized by an `Entity` that is classified by an `AnnotationConcept`, also defined by the `Description`. Both `InformationEntity` and the metadata have their settings (hasSetting) in the `AnnotationSituation` and they are sticked together through the Information Realization Pattern, shown in figure 2.5. This part of the pattern works as a stand-alone Annotation Pattern without provenance information, because it already gives us a conceptualized view on annotations. The relation between the `InformationEntity` and the meta–data modeled by the use of the Information Realization Pattern is also optional and the pattern could be used without this additional information. Furthermore as mentioned it is possible to enrich the pattern with provenance information by adding the Provenance Information Pattern. In this case both `Description` and `Situation` of the Annotation Pattern and of the Provenance Information Pattern are congruent. You can see the original Annotation Pattern figure, like it was proposed in [30], in A.1 and the pattern modified under the assumption of an existing Provenance Information Pattern in A.5. [31]

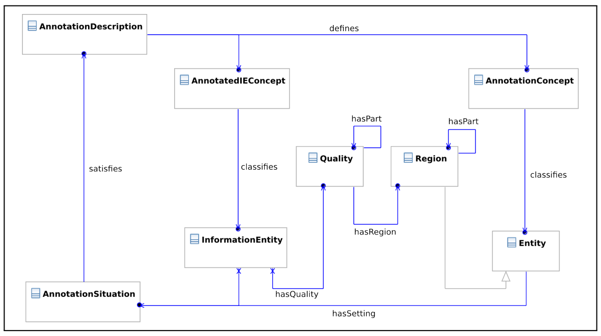


Figure A.5: The Annotation Pattern

A.1.3 The Decomposition Pattern

The **Decomposition Pattern**, like the Annotation Pattern is derived from the D&S Pattern. The Annotation Pattern and the Decomposition Pattern have a pretty similar layout. The pattern consists of a `DecompositionDescription` defining exactly one `CompositeConcept` and at leastone `ComponentConcept`. The `ComponentConcept` classifies the whole and the `CompositeConcept` classify the parts, whereas both are represented by `InformationObjects`. Both objects have their settings in the `Situation` as well as in the Annotation Pattern. Provenance information and the use of the Information

Realization Pattern can be added in the same way as it was added in the Annotation Pattern. The figure A.6 shows the modified Decomposition Pattern without provenance information according to our discussion in the previous section. [31]

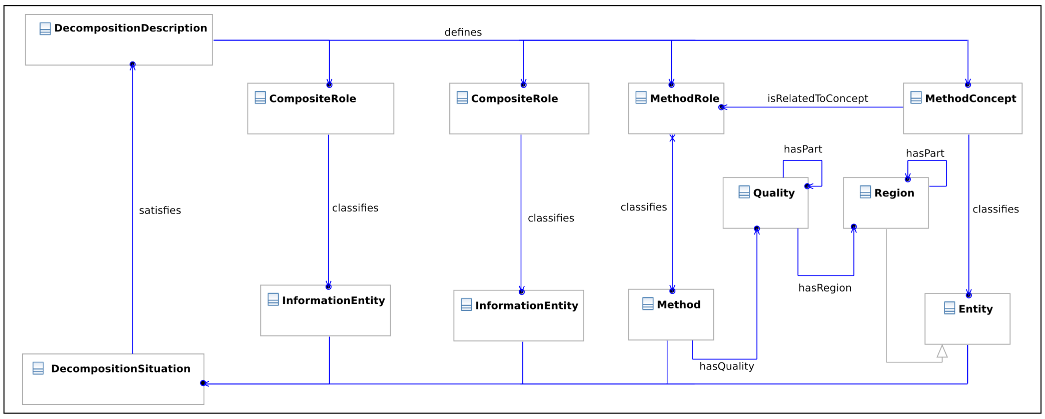


Figure A.6: The Decomposition Pattern in the M3O

A.2 The Eventmodel F Patterns

The following Patterns are all taken from the Event-Model-F [33]. Most of the definitions in Event-Model-F [33] have been specialized from the generic classes defined in DUL, e.g. Event, Object, Abstract, Quality, etc. With the concrete implementation of the Decomposition Pattern and by comparing it to the M3O implementation of the same pattern, we will show in an exemplary manner different appearances of similar patterns. These different forms are owed to the fact, that the concrete characteristics of one and the same pattern can differ in different domains, or to the evolution in pattern design specially when the patterns are designed by the same group of persons. This shows again that there is not unique solution in the design of Ontologies and Ontology Patterns, even if the domain is the same and the design process is fully underpinned with state of the art theory. There have always choices to be made for which there is no clear answer available in theory.

A.2.1 The Decomposition Pattern

The **Decomposition Pattern** in the Event-Model-F, shown in figure A.7 deals with the (de)composition of a event. This decomposition may appear as temporal, spatiotemporal or as spatial decomposition. As the nomenclature indicates temporal decompositions divides by time and spacial by space, spatiotemporal decompositions divides the given event by both time and space. This Pattern uses the mechanism of reification through a specialization of the D&S Pattern. It defines a `EventDecompositionDescription` and a `EventDecompositionSituation` that satisfies the `Description`. The event itself is classified by a EventRole in which the event may play neither the composite role (the whole) or the component role (the part).A concrete component EventRole has a parameter set by an EventCompositionConstrain, that can be spatial, temporal or spatiotemporal, depending on the concrete appearance of the decomposition. These EventCompositionConstrains are defined by the DecompositionDescription and parameterize the Region in the Data Value Pattern of the concrete Event. The event and its concrete regions are all included in the DecompositionSituation. [33]

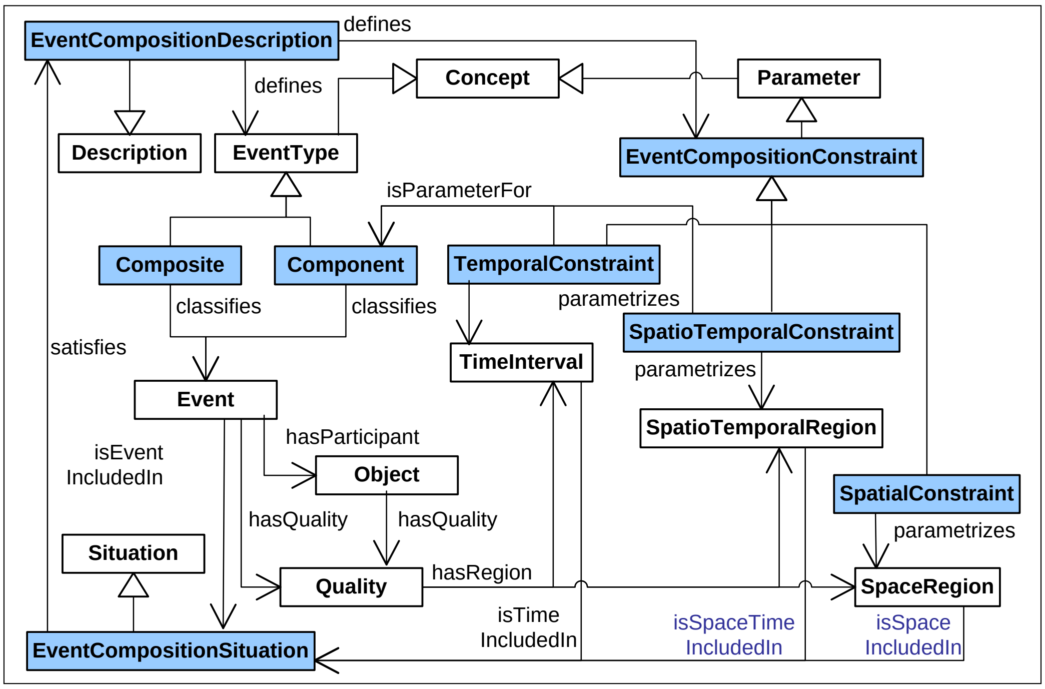


Figure A.7: The Decomposition Pattern in the Event-Model-F

A.2.2 The Participation Pattern

The **Participation Pattern** deals with the aspect of the constitutional declaration of events by giving the objects, e.g. persons, participating in such a event. In Figure A.8 we can see that this participation is modeled through a specialization of the D&S Pattern. Therefor the pattern defines a ParticipationDescription that is satisfied by a ParticipationSituation. The concrete Situation includes the event and the objects participating in this event. The Description defines the concepts of event and participant, as in the D&S Pattern. Different types of events or objects participating, are represented as instances of the related concepts. With this mechanism it is possible to model the instantiation relationship between a concrete Event and its *super concept*, e.g. if we talk about the hurricane katrina only in this way the relationship can be reified. The object itself can be described more precisely using a domain Ontology or with the Data Value Pattern of DUL presented before. [33]

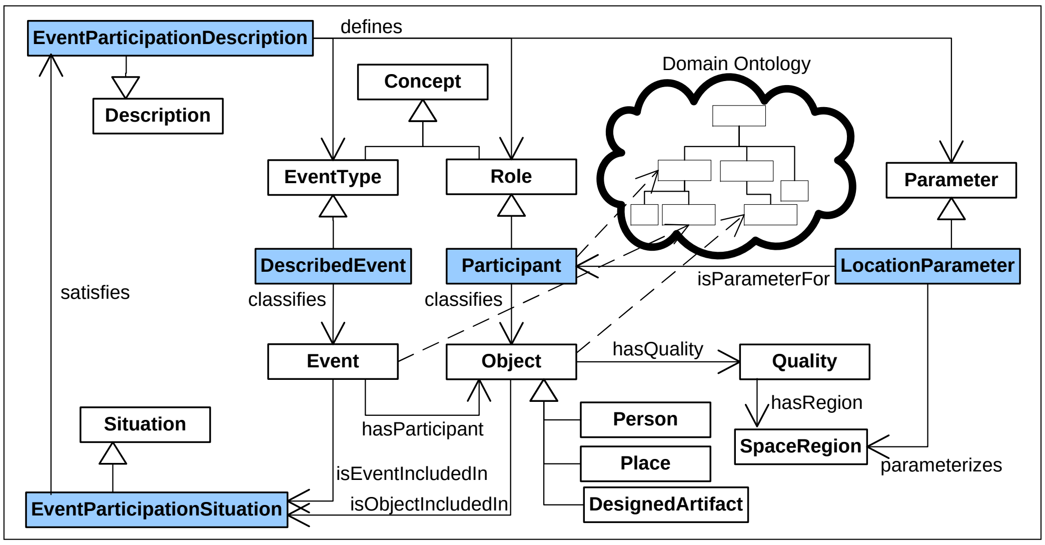


Figure A.8: The Participation Pattern

A.2.3 The Interpretation Pattern

Due to the fundamental design decision of identification and clear separation of aspects of events, different context-dependent views on an event can be described using the Interpretation Pattern. Each instance models a single, independent view (*interpretation*) on an event by joining the different Patterns relevant in the context of the *interpretation*. This pattern is based on the D&S Pattern, thus it defines a `EventInterpretationDescription` and an `EventInterpretationSituation`. This `Description` defines a interpretant, the entity that specifies how the event is interpreted, and a `RelevantSituation` as a `Situation` that satisfies the Patterns `Description` relevant in this interpretation.

A.3 OWL to UML Mappings

A.3.1 Figures

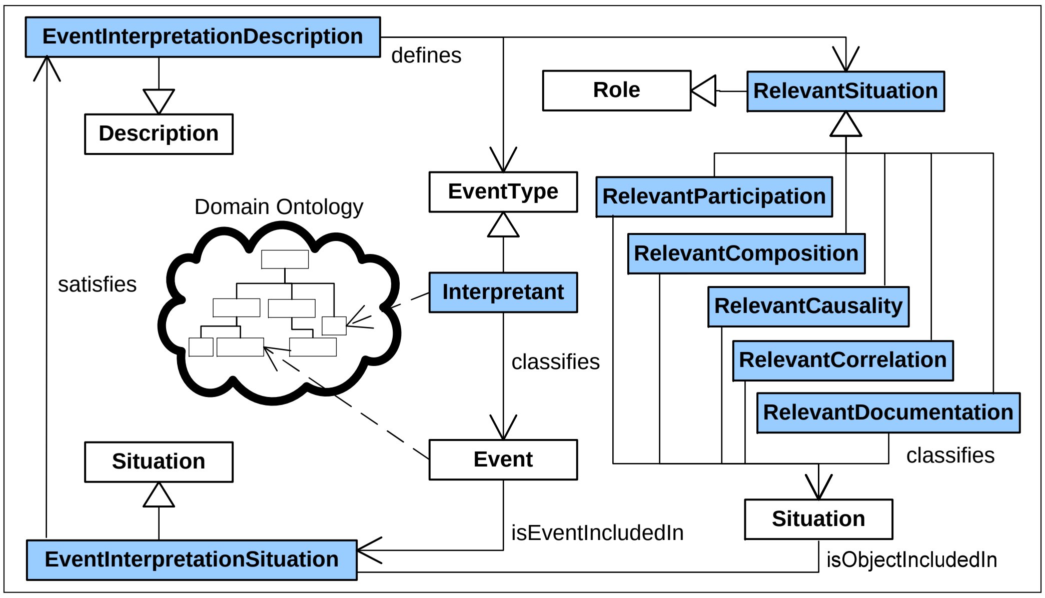


Figure A.9: The Interpretation Pattern

Figure A.10: `owl:AllValuesFrom` in UML

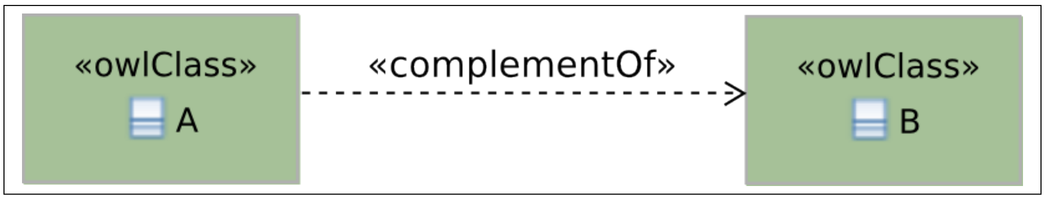


Figure A.11: `owl:appcomplementOf` model in UML

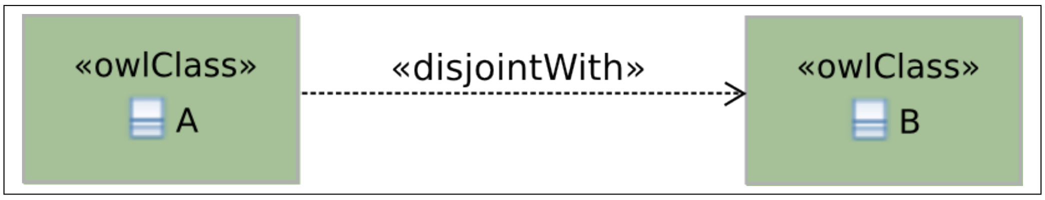


Figure A.12: `owl:disjointWith` in UML

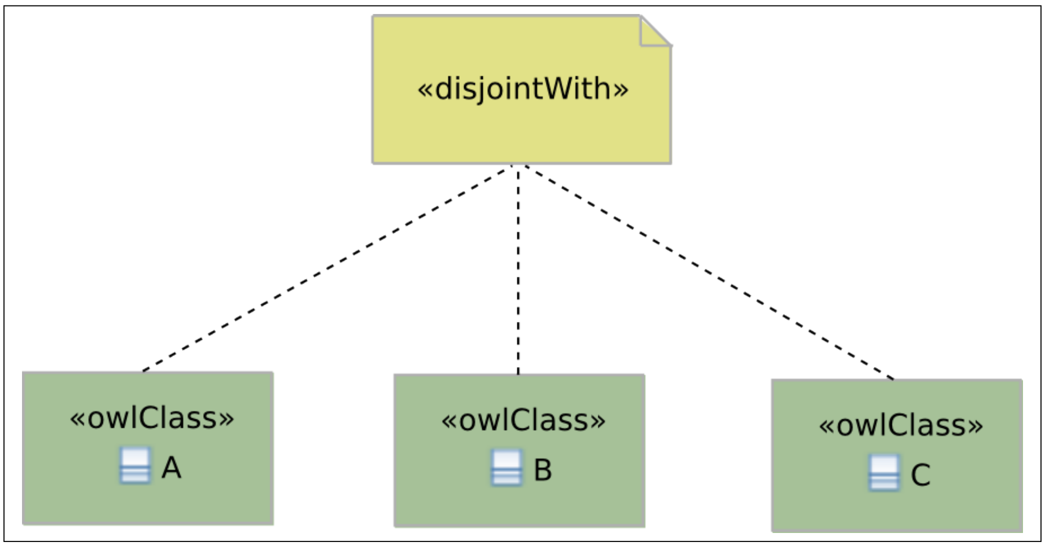


Figure A.13: Using `owl:disjointWith` between multiple Classes in UML

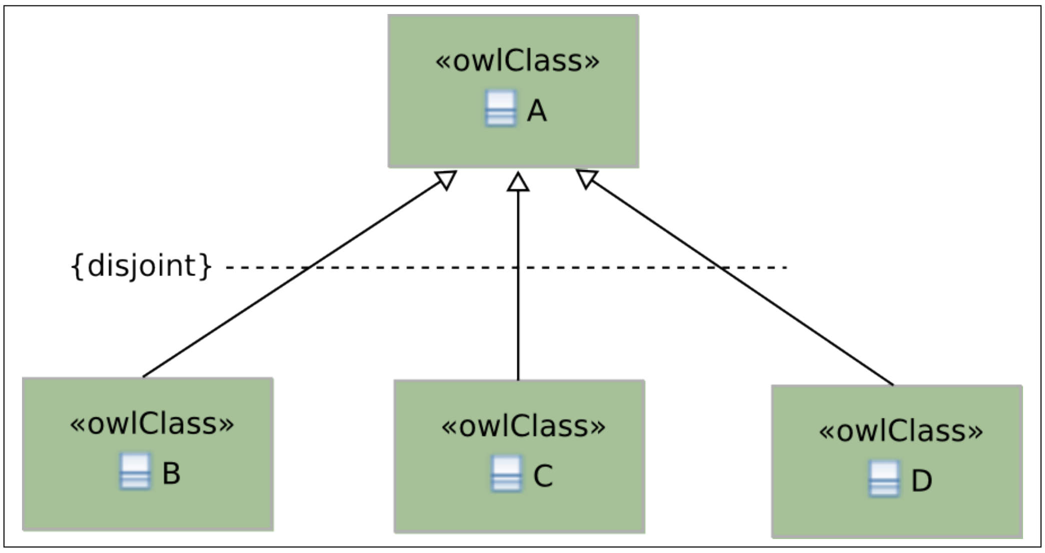


Figure A.14: Using `owl:disjointWith` with Common Supertype in UML

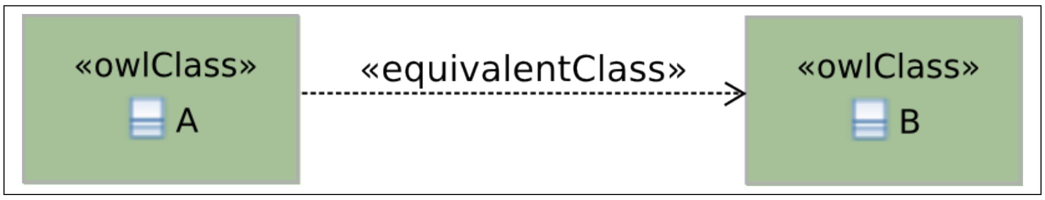


Figure A.15: `owl:equivalentClass` in UML

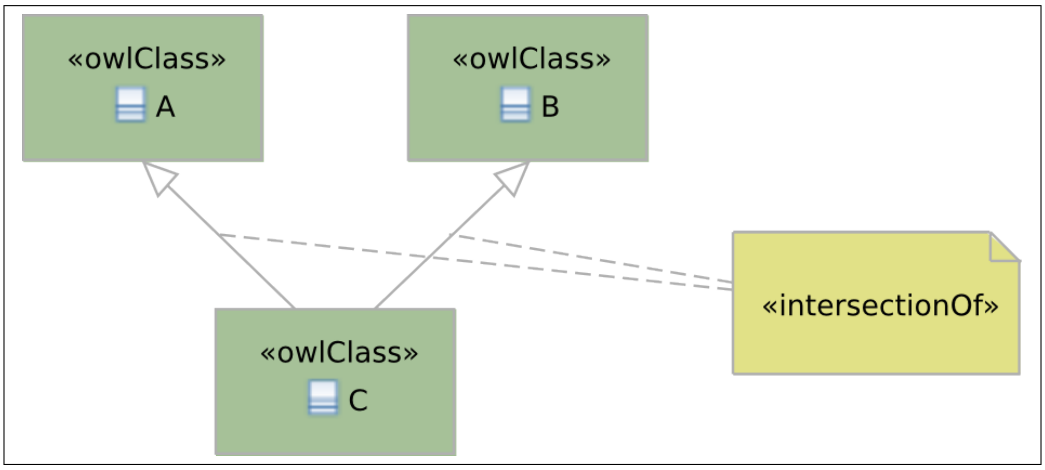


Figure A.16: ODM current Version `owl:intersectionOf` in UML

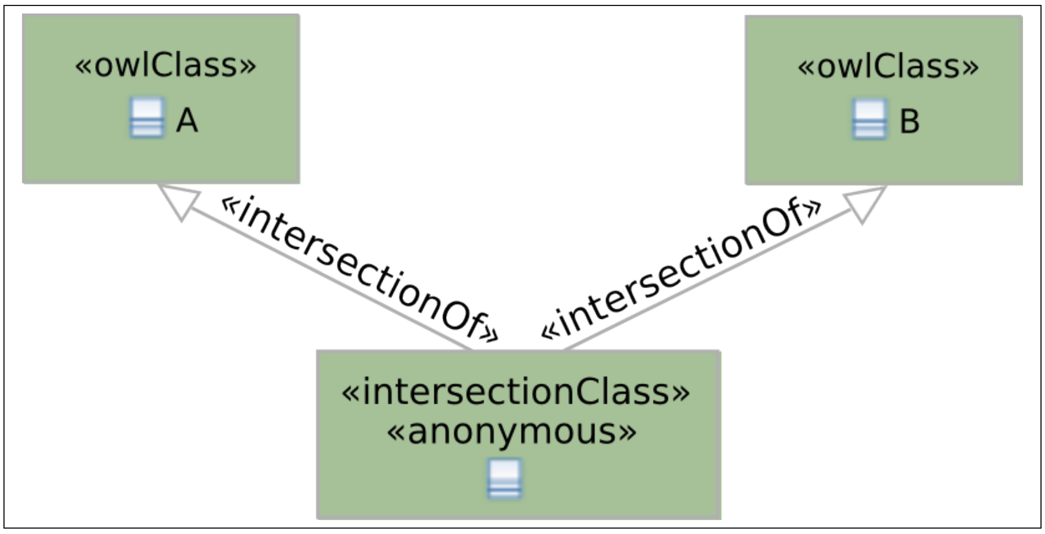


Figure A.17: ODM next Version `owl:intersectionOf` in UML

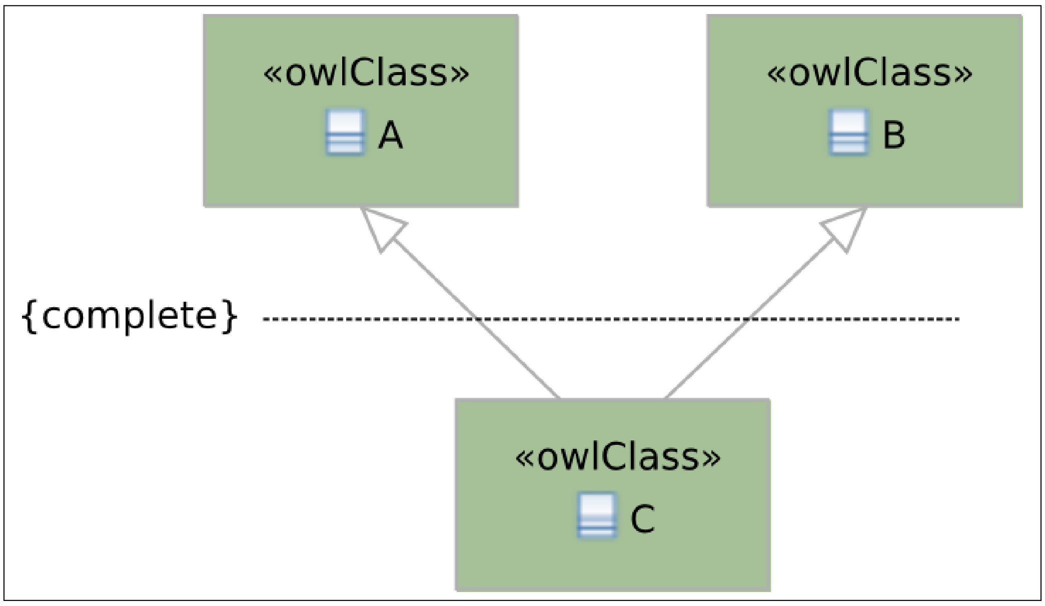


Figure A.18: ODM current Version `owl:unionOf` in UML

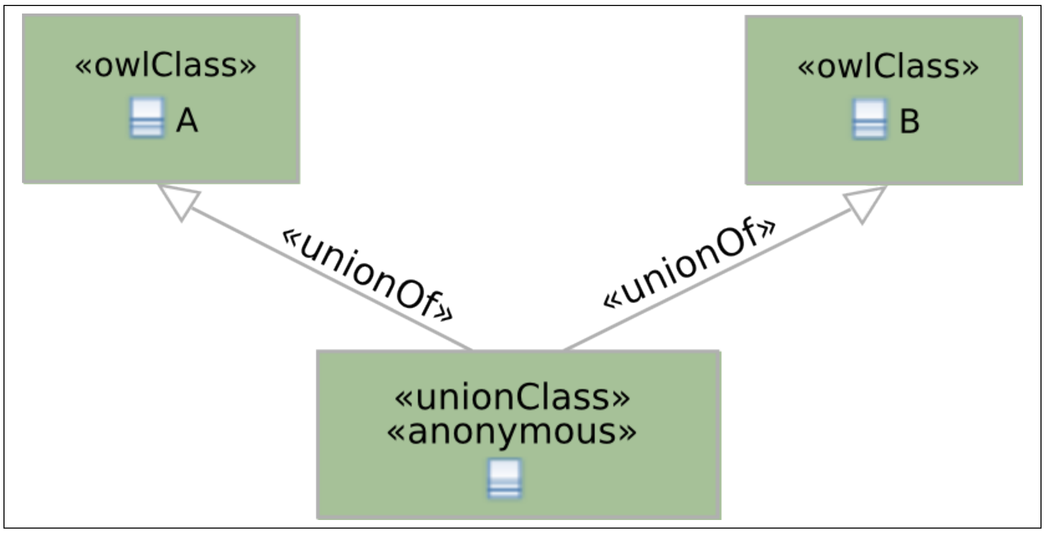


Figure A.19: ODM next Version `owl:unionOf` in UML

Bibliography

[1] M2T JET-FAQ. `http://wiki.eclipse.org/M2T-JET-FAQ`, 2010.

[2] Valentina Presutti Aldo Gangemi. *Ontology Design Patterns*, chapter Ontology Engineering Part II, pages 221– 243. In Staab and Studer [36], second edition, 2009.

[3] Thomas Franz Ansgar Scherp, Carsten Saathoff and Steffen Staab. Designing beautiful core ontologies. In *Applied Ontology 3*. IOS Press, 2009.

[4] Grigoris Antoniou and Frank van Harmelen. *Web Ontology Language: OWL*, chapter Ontology Representation Languages, pages 91–110. In Staab and Studer [36], second edition, 2009.

[5] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[6] Eva Blomqvist and Kurt Sandkuhl. K.: Patterns in ontology engineering: Classification of ontology patterns. In *In: Proc. of ICEIS2005*, 2005.

[7] Stefano Borgo and Claudio Masolo. *Foundational Choises in DOLCE*, chapter Ontologies, pages 361–381. In Staab and Studer [36], second edition, 2009.

[8] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, February 2004. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[9] Saartje Brockmans, Peter Haase, Pascal Hitzler, and Rudi Studer. A metamodel and UML profile for Rule-Extended OWL DL Ontologies. In *III European semantic Web conference ESWC 2006*, June 2006-06.

[10] James Bruck and Kenn Hussey. Customizing UML: Which Technique is Right for You? `http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html`, June 2008.

[11] Ian Horrocks Franz Baader and Ulrike Sattler. *Description Logics*, chapter Ontology Representation Languages, pages 21–43. In Staab and Studer [36], second edition, 2009.

[12] A. Gangemi. Ontology design patterns for semantic web content. In *M. Musen et al. (eds.): Proceedings of the Fourth International Semantic Web Conference*. Berlin, Springer, 2005.

[13] A. Gangemi and P. Mika. Understanding the semantic web through descriptions and situations. In *International Conference on Ontologies, Databases and Application of Semantics (ODBASE 2003)*, Catania, (Italy), November 2003.

[14] Aldo Gangemi. Dolce+dns ultralite (dul). `http://ontologydesignpatterns.org/wiki/Ontology:DOLCE%2BDnS_Ultralite`, 2009.

[15] Norman L. Geisler, editor. *Baker Encyclopedia of Christian Apologetics.* Baker Books, Grand Rapids Michigan, 1999.

[16] L. Hart and P. Emery. OWL Full and UML 2.0 Compared. `http://uk.builder.com/whitepapers/0and39026692and60093347p-39001028qand00.htm`, Acessado em Outubro de 2004.

[17] Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A. Padget. Automatic Mapping of OWL Ontologies into Java. In Frank Maurer and Günther Ruhe, editors, *SEKE*, pages 98–103, 2004.

[18] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari, and L. Schneider. WonderWeb deliverable D17. the WonderWeb library of foundational ontologies and the DOLCE ontology. Technical report, ISTC-CNR, 2002.

[19] Diana Maynard, Adam Funk, and Wim Peters. Using lexico-syntactic ontology design patterns for ontology creation and population. In Francois Scharffe Eva Blomqvist, Kurt Sandkuhl and Vojtech Svatek, editors, *WOP '09:Proceedings of the Workshop on Ontology Patterns*, 2009.

[20] Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Mike Smith. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax.* W3C, `http://www.w3.org/TR/2009/CR-owl2-syntax-20090611`, june 2009.

[21] Object Management Group, Framingham, Massachusetts. *MOF 2.0/XMI Mapping Specification, v2.1.1*, 2007.

[22] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Core Specification. Technical Report formal/06-01-01, OMG, 2001. OMG Available Specification.

[23] OMG. *Ontology Definition Metamodel Version 1.0.* Object Modeling Group, May 2009.

[24] Fernando Silva Parreiras, Carsten Saathoff, Tobias Walter, Thomas Franz, and Steffen Staab. ‘a gogo: Automatic generation of ontology apis. In *IEEE Int. Conference on Semantic Computing*. IEEE Press, 2009.

[25] C. S. Pierce et al. *Writings of Charles S. Peirce: Achronological Edition Volume 1. 1857-1866*. Indiana University Press, 1981.

[26] Shelley Powers. *Practical RDF: Solving Problems with the Resource Description Framework*. O'Reilly, Beijing, 2003.

[27] Valentina Presutti and Aldo Gangemi. Content ontology design patterns as practical building blocks for web ontologies. In *ER '08: Proceedings of the 27th International Conference on Conceptual Modeling*, pages 128–141, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF W3C Recommendation 15 January 2008*. W3C, `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`, Feb 2008.

[29] Carsten Saathof. How many axioms would you like? `http://kodemaniak.de/?p=40`, 3 2010.

[30] Carsten Saathoff, Stefan Scheglmann, and Simon Schenk. Winter : Mapping rdf to pojos revisited. In *Poster and Demo Session, ESWC 2009, Heraklion, Greece, May 31 - June 3*, Heraklion, Greece, 2009.

[31] Carsten Saathoff and Ansgar Scherp. M3o: The multimedia metadata ontology. In *Proceedings of the Workshop on Semantic Multimedia Database Technologies, 10th International Workshop of the Multimedia Metadata Community (SeMuDaTe 2009)*, Graz, Austria, 2009.

[32] SC34/WG3. *ISO/IEC 13250-1 Topic Maps – Overview*, june 2008.

[33] Ansgar Scherp, Thomas Franz, Carsten Saathoff, and Steffen Staab. F–a model of events based on the foundational ontology dolce+dns ultralight. In *K-CAP '09: Proceedings of the fifth international conference on Knowledge capture*, pages 137–144, New York, NY, USA, 2009. ACM.

[34] Michael K. Smith, Michael Smith, Chris Welty, and Deborah L. McGuinness. *OWL Web Ontology Language Guide W3C Recommendation 10 February 2004*. W3C, `http://www.w3.org/TR/2004/REC-owl-guide-20040210/`, February 2004.

[35] John F. Sowa. *Conceptual Graph Standard*. 2001. Proposed draft.

[36] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, second edition, 2009.

[37] R. Studer, R. Benjamins, and D. Fensel. Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering*, 25:161–197, 1998.

[38] Denny Vrandečić 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, October 2006. Springer.