Declarative Representation of Programming Access to Ontologies

Stefan Scheglmann, Ansgar Scherp, and Steffen Staab

Institute for Web Science and Technologies
University of Koblenz-Landau, Germany
{schegi,scherp,staab}@uni-koblenz.de

Abstract. Using ontologies in software applications is a challenging task due to the chasm between the logics-based world of ontologies and the object-oriented world of software applications. The logics-based representation emphasizes the meaning of concepts and properties, i.e., their semantics. The modeler in the object-oriented paradigm also takes into account the pragmatics, i.e., how the classes are used, by whom, and why. To enable a comprehensive use of logics-based representations in object-oriented software systems, a seamless integration of the two paradigms is needed. However, the pragmatic issues of using logic-based knowledge in object-oriented software applications has yet not been considered sufficiently. Rather, the pragmatic issues that arise in using an ontology, e.g., which classes to instantiate in which order, remains a task to be carefully considered by the application developer. In this paper, we present a declarative representation for designing and applying programming access to ontologies. Based on this declarative representation, we have build OntoMDE, a model-driven engineering toolkit that we have applied to several example ontologies with different Characteristics.

1 Introduction

One of the most challenging issues in implementing Semantic Web applications is that they are built using two different technologies: object-oriented programming for the application logic and ontologies for the knowledge representation. Object-oriented programming provides for maintainability, reuseability and robustness in the implementation of complex software systems. Ontologies provide powerful means for knowledge representation and reasoning and are useful for various application domains. For accessing ontological knowledge from object-oriented software systems, there are solutions like ActiveRDF [8] and Jastor[1]. Most of these frameworks make use of the structural similarities of both paradigms, e.g., similar inheritance mechanisms and utilize simple solutions known from the field of object-relational mapping. But with the use of these existing tools some problems cannot be solved: Typically, the structural similarities lead to a one-to-one mapping between ontology concepts, properties and individuals and object-oriented classes, fields and objects, respectively. This leads to a data-centric

[1] http://jastor.sourceforge.net/ last visit June 24, 2011.

E. Simperl et al. (Eds.): ESWC 2012, LNCS 7295, pp. 659–673, 2012.

object-oriented representation of the ontology which ignores the responsibility-driven [17] nature of object-orientation. It is up to the API developer to provide additional object-oriented layers which allow the use of the generated class representations. In addition, not all concepts and relations that must be defined in the ontology are useful in the object-oriented model. Again it is up to the API developer to provide proper encapsulations to hide such concepts from the application developer. Since this additional programming effort of the API developer relies on the one-to-one class representations of a specific ontology, changes in the ontology easily end up in excessive adaptation work of the API. In addition, as the experiences in the WeKnowIt-project show, new requirements and changes in the ontology may imply tedious and complex updates of the programming access to the logics-based representation. What is needed is a tool that comprehensively supports API developers in designing pragmatic programming access to ontological knowledge.

In this paper, we present a declarative representation for pragmatic access to ontological structures that supports the developer in building programmatic access to ontologies. We present OntoMDE, a Model-Driven Engineering toolkit for the generation of programming access to ontologies that is based on these declarative representations. OntoMDE supports the developer in building APIs adapted to concrete application needs. We define our problem and introduce a scenario and running example in the following section. In Section 3, we define the requirements for developing programming access to ontologies. Based on these requirements, we introduce our approach in Section 4. We have applied our approach at the examples of selected ontologies presented in Section 5. In Section 6, we discuss the related work, before we conclude the paper.

2 Scenario, Example and Problem

First, we present a scenario to motivate our work. Subsequently, an example ontology is introduced to demonstrate the problems of today's API generation tools conducting a one-to-one mapping. We compare the API resulting from the use of existing tools with an API that would be more natural to have in a purely object-oriented model.

2.1 Scenario: An Ontological Multimedia Annotation Framework

Jim works for a multimedia company and is responsible for the integration of knowledge-base access in an object-oriented media annotation framework. The media annotation framework should support the user in annotating multimedia content such as images or video clips. Jim shall use an ontology for representing annotated media as well as the multimedia annotations. He has not been involved in the design of the ontologies. His task is to define the programming interfaces to access and update the knowledge-base seamlessly from the application. He has to consider that further specializations toward domain-specific annotations could result in changes of the implementation.

2.2 Example: Ontology-Based Modeling of Multimedia Metadata

Figure 1(a) shows an excerpt of the ontology used by Jim to model the multimedia metadata. The example is based on the Multimedia Metadata Ontology (M3O) [13] for representing annotation, decomposition, and provenance information of multimedia data. It models the annotations of an image with an EXIF[2] geo-point wgs84:Point[3] and a FoaF[4] person foaf:Person as image creator. As we can see from the different namespaces, the m3o:Image, wgs84:Point and foaf:Person concepts and their superconcepts dul:InformationEntity, dul:Object and finally dul:Entity are defined in different ontologies. The inheritance and import relationships are shown in Figure 1*b*, which is needed important for a proper API representation.

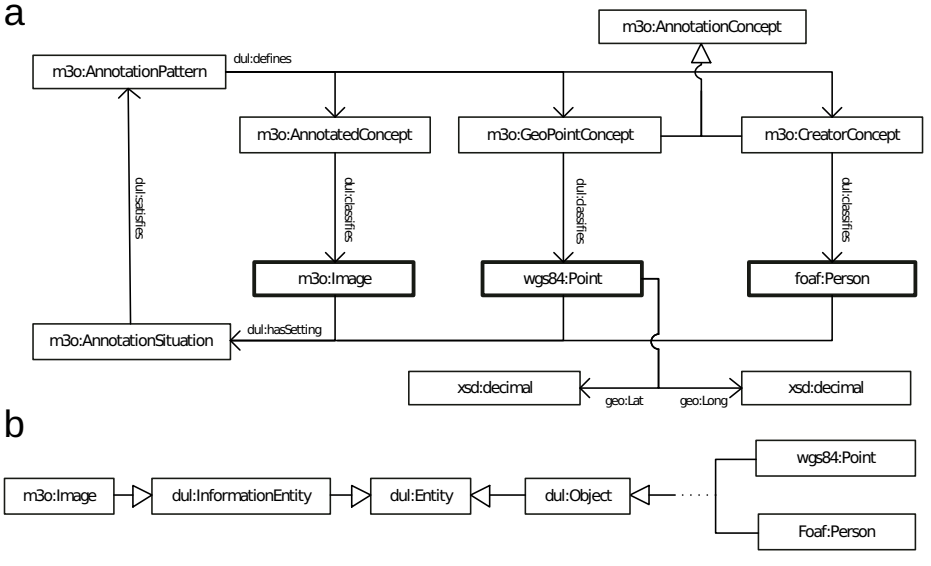

Fig. 1. Annotation of an Image with its Geo-location and Creator

2.3 Issues with APIs Provided by Existing Frameworks

Jim uses a simple ontology API generation framework with a one-to-one mapping like those mentioned in the introduction to generate a programming access to the ontology. Figure 2 shows the generation result for the ontology excerpt presented above using such an existing tool. The framework creates a class representation

[2] `http://www.exif.org/` last visit dec 05, 2011.
[3] Basic Geo (WGS84 lat/long) Vocabulary `http://www.w3.org/2003/01/geo/` provides the namespace, last visit dec 05, June 2011.
[4] `http://www.foaf-project.org/` last visit dec 05, 2011.

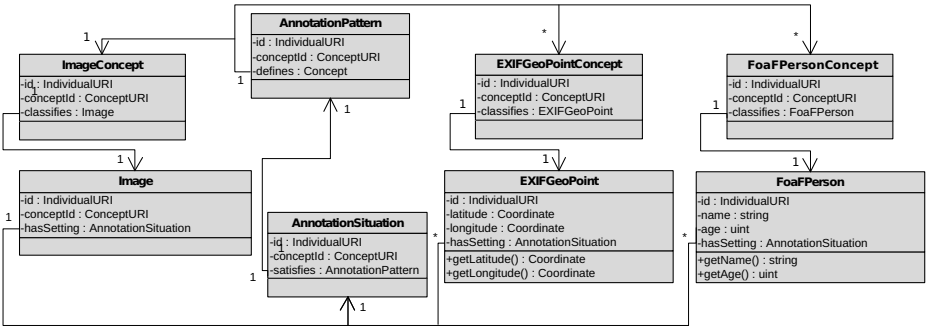

Fig. 2. Naive ontology API implementation generated by existing tools

for each of the concepts defined in the ontology. The relationships between concepts are represented as fields of the domain classes, e.g., the satisfies relationship between the m3o:AnnotationSituation and the m3o:AnnotationPattern concept is represented as satisfies field of type AnnotationPattern in the AnnotationSituation class. The generated class structure gives Jim no information about how to use it, i.e., which classes to instantiate when annotating an image with a geo-point or a creator. In fact one has to instantiate the class representations AnnotationPattern, AnnotationSituation, Image, EXIFGeoPoint, ImageConcept and EXIFGeoPointConcept and fill all the fields representing the relationships, namely defines, classifies, hasSetting and satisfies.

Furthermore not all class representations are of direct concern for Jim's application. Some of these representations provide direct *content* for the application, like the annotated entity — the Image — or the annotation entities — the EXIFGeoPoint and the FoaFPerson. Other classes only provide the structure necessary for a proper knowledge representation. The M3O ontology uses the Description & Situation (D&S) ontology design pattern. Description & Situation is another reification [3] formalism in contrast to the RDF reification[5]. For using D&S as reification formalism one has to add additional resources, the description, situation and the classifying concepts. The class representation for these concepts are of no use for Jim when using the API in his application. For this reason, he decides to encapsulate them from direct access and hide them from an eventual application developer.

2.4 Solution: Reference API for the Example Ontology

Due to the problems arising with the use of simple one-to-one mappings, Jim decides to build a programming interface to the ontology without the use of an API generation framework. Please note that the subsequently described API results from the design decisions made by Jim and represents only one possible model of an API for accessing this ontology. The API model designed by Jim is presented in Figure 3. In addition Figure 4 ahows two further possible

[5] `http://www.w3.org/TR/rdf-mt/#ReifAndCont` last visit dec 10, 2011.

models. All the API models are used in our evaluation in Section 5. Jim first identifies the functionality to be provided by the API, the annotation of images. Jim decides to provide a class for this annotation, the annotation class. In the following, we describe the different designs of the three APIs. **API-1:** He defines the set of concepts and properties involved in this functionality. Jim classifies the concepts in this set according to how they are used in the application and he splits them into two disjoint sets. The first set contains all concepts representing the *content* the application works on. In our terminology, we call them ***content concepts***. We would like to emphasize that in our scenario Jim as an API developer will not have to know about the terminology we use at all; but it is significantly easier in this paper to use our terminology to explain the different decisions he may take when developing the API. For our example Jim chooses the m3o:Image, the wgs84:Point and the foaf:Person to provide the *content*. The other set contains the concepts of *structural* concern for the knowledge representation. Subsequently, we call these concepts ***structure concepts***. For Jim these concepts are m3o:AnnotationPattern, m3o:AnnotationSituation, m3o:AnnotatedConcept, m3o:GeoPointConcept, and m3o-:CreatorConcept and he wants his API to encapsulate and hide class representations of such concepts from the application. In our terminology, we call a set of concepts and relations related to an API class a ***semantic unit*** $SU = (CO, SO, R)$ with CO the set of ***content concepts***, SO the ***structure concepts*** and R the set of relations. For our example, ***semantic units*** are, e.g., the annotation as described above or the geopoint consisting of the wgs84:Point together with its latitude and longitude. Jim wants his API to be prepared for arbitrary multimedia content and new types of annotations. The ontology provides abstract concepts for multimedia content and annotations in its inheritance structure presented in Figure1*b*. But not all concepts from this structure are of interest to the application. Thus Jim decides to use only the least common subsumers, e.g., dul:InfomationObject for annotatable multimedia content and dul:Object for annotations. Jim implements interfaces representing these two concepts.

Jim is now able to design the API. He defines a class for the annotation functionality as shown in Figure 3. In addition, he defines a class for each ***content concept*** the application works on, in this case **Image**, **EXIFGeoPoint** and **FoaFPerson**. These classes implement the interfaces derived from the inheritance structure of the ontology, **InformationEntity** and **Object**. The **InformationEntity** interface has to be realized by classes representing multimedia content, e.g., by the **Image** class. The **Object** interface has to be realized by annotation entities, e.g., the classes **EXIFGeoPoint** and **FoaFPerson**. All these classes and interfaces together with the operations form a so-called ***pragmatic unit***. A ***pragmatic unit*** is a tuple $PU = (C, F, M)$ that contains the classes C, the fields F and the methods M of an object-oriented model and that relates to a specific ***semantic unit*** in the underlying knowledge model.

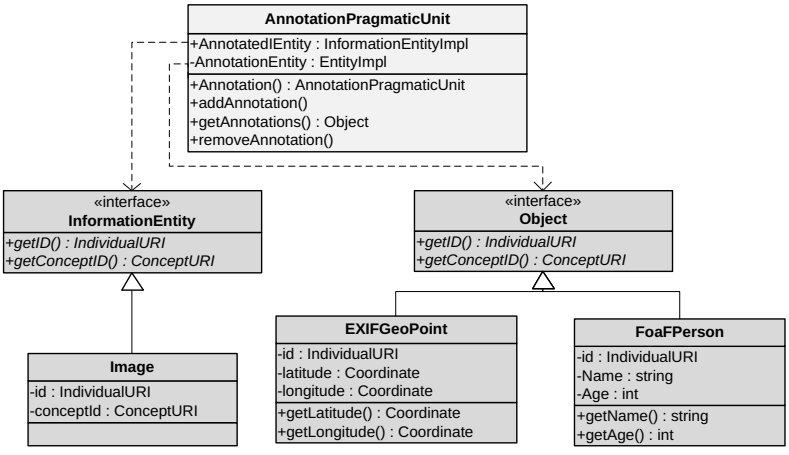

Fig. 3. API for the Running Example developed by Jim

API-2: Another possible model is API 2 shown in Figure 4 , which is a more lightweight API for an image-viewer. The API only consists of three classes, a representation for the m3o:Image, the wgs84:Point and the foaf:Person. The class representation of the annotation ***semantic unit*** is integrated within the m3o:Image ***content concept*** class representation.

API-3: The decisions behind API 3, shown in Figure 4 are basically the same as for API 1 with the difference that the annotation ***semantic unit*** class representation should be identifiable by an URI. For this purpose the annotation ***semantic unit*** class representation is integrated with the AnnotationSituation class representation. In this API model the AnnotationSituation is classified as ***content concept*** and encapsulates the annotation ***semantic unit***.

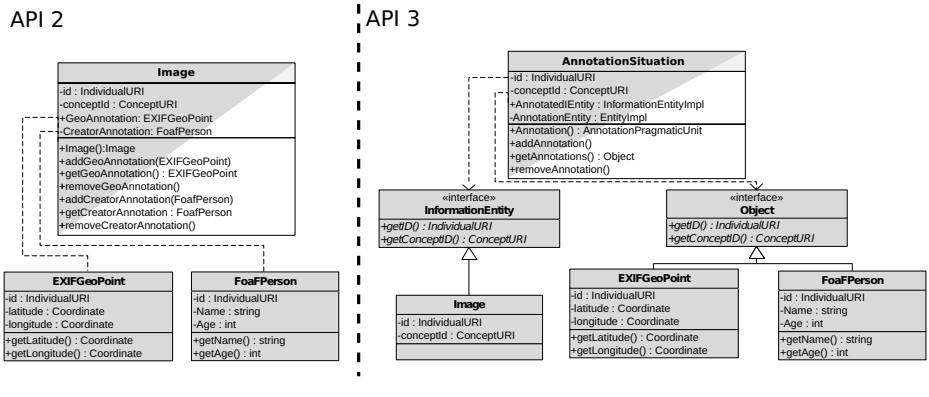

Fig. 4. Alternative APIs for the Running Example Ontology

3 Requirements for Programming Access to Ontologies

We analyze the requirements for the generation of programming access to ontologies. The requirements have been derived from real world implementation efforts made for different projects in our workgroup, e.g., the EU project WeKnowIt[6]. We use the scenario in Section 2 and the implementation of the reference API described in Section 2.4 to motivate the requirements. The requirements are distinguished into two sets of requirements: (1) requirements directly related to the programming access described in Section 3.1, typically in form of an API; (2) requirements related to a process that generates such an API described in Section 3.2.

3.1 Requirements on the Pragmatic Programming Access

(R1) Concept Representations. Programming access to ontologies has to represent the ontology concepts as classes in the object-oriented software system similar to Data Access Objects[7] (DAOs), ActiveRecords or Data Mapper (both [1]) in the world of relational databases. Frameworks like those presented in the related work usually map each ontology concept to an object-oriented class representation and map the concept's properties to fields of this class. For our example, such a mapping is shown in Figure 2.

(R2) Encapsulation. Not all concepts of the ontology are of concern for an application developer. In Section 2.4, Jim identifies several concepts providing the *content* for his application, the ***content concepts***. The rest of the concepts are classified as ***structure concepts***. These ***structure concepts*** are only of concern for the proper knowledge representation. A programming access should provide for encapsulating concepts that are not interesting for an application developer.

(R3) Mapping of Inheritance Structures. There are differences between the inheritance structure of an API and of an ontology. In object-orientation, a class can inherit both data (attributes) and behavior (methods) from an ancestor class. Furthermore some object-oriented languages do not support multiple inheritance, e.g. Java. For generating programming access to ontologies, we need information how to generate a lean and useful inheritance structure from the ontology for the API.

(R4) Pragmatic Units. APIs provide a programming interface for their responsibility, e.g., the annotation of images like the API from our example in Section 2.3. Such a programming interface supports methods to perform operations, like in our example adding, removing or manipulating annotations and images. Performing such operations in programming access to ontologies often results in the manipulation of multiple ontology entities and thus multiple concept-class

[6] `http://www.weknowit.eu/` last visit dec 5, 2011.

[7] DAOs as Core J2EE Pattern `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`

representations. Our API should provide classes to support the application developer in performing these operations in an easy and well encapsulated way.

(R5) Method Behavior. APIs provide methods to access or manipulate API entities or to query for entity properties. In some cases, it might be necessary to fall back to reasoning on the ontology [10] to be able to answer queries. For example querying for all instances of a specific concept could be such a question. A method for such a query performed on the Java representation could guarantee soundness but never completeness. The same also applies for consistency preservation. In some cases, the API could restrict its behavior in a way that it ensures the consistency of the represented knowledge. We expect the API to either inform the calling method or throw an exception that the requested action would affect the consistency of the represented knowledge. Sometimes, it is not possible or practical for complexity reasons to restrict the API behavior. In this case the API cannot ensure the consistency. Currently, we focus on cases where restrictions or query answering on the API are possible, e.g., qualified number restrictions on properties. A reasoner integration to ensure validity of operations remains for future work.

3.2 Requirements on the Process for Generating Programming Access to Ontologies

(R6) Customizing generated APIs. The output of the API gerneration process is strongly driven by the developer and the context of the target application. For instance, in Section 2.4 we have demonstrated how three different APIs might have been defined for a given ontology, reflecting different needs of the target applications. The generation process has to support the developer in controlling and customizing the output. From our observation, we know that concept classification and assignment to ***semantic units*** is mostly uniform for various application scenarios but choice of ***pragmatic units*** and their arrangement can vary strongly from case to case. The import of ontologies and the intended inheritance structure in the API can also vary for different application scenarios.

(R7) Legacy APIs integration. The API developer should be able to integrate legacy APIs. Let us assume Jim uses the image class of the AWT API[8]. To use this image class, Jim has to integrate it with the ontology API and provide ontology access functionalities for this class.

(R8) Import. The generation process has to deal with import instructions in the ontologies. A generation process has to manage all imports and decide which are important for the API generation process.

(R9) Deanonymization of Concepts. Ontologies allow for anonymous concepts in complex class expressions in OWL or blank nodes in RDFS. However, there are no anonymous classes in object orientation. For this reason, we only allow named concepts in ontologies and need to de-anonymize anonymous concepts first, if necessary.

[8] `http://download.oracle.com/javase/1.4.2/docs/api/java/awt/Image.html`

4 Programming Access to Ontologies with OntoMDE

In order to alleviate application developers from building the pragmatics of accessing Semantic Web knowledge in object-oriented applications, we present OntoMDE a Model-driven Engineering (MDE) approach for the generation of programming access APIs from an input ontology. The OntoMDE framework guides the developer through the semi-automatic generation process. Figure 5 depicts the whole process with its two intermediate models, the MoOn and the OAM. OntoMDE provides tools to support the user in adding declarative information about the pragmatic programming access to the intermediate models.

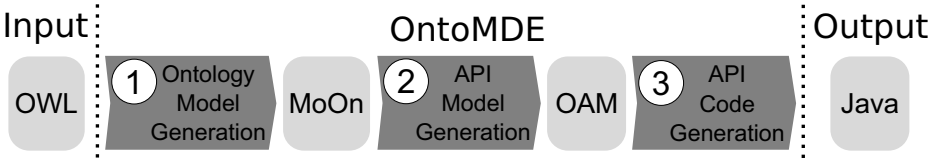

Fig. 5. The API Generation Process

In the first step, the Model of Ontologies (MoOn) is used to represent crucial properties of the target API as properties of the ontology in a declarative manner. In MoOn, concepts are classified as either being ***content concepts*** or ***structure concepts***. ***Semantic units*** are defined and one can adapt parts of the ontology's inheritance structure to the API. Figure 6 shows the ***semantic unit*** **annotation** from our running example in the MoOn-based representation.

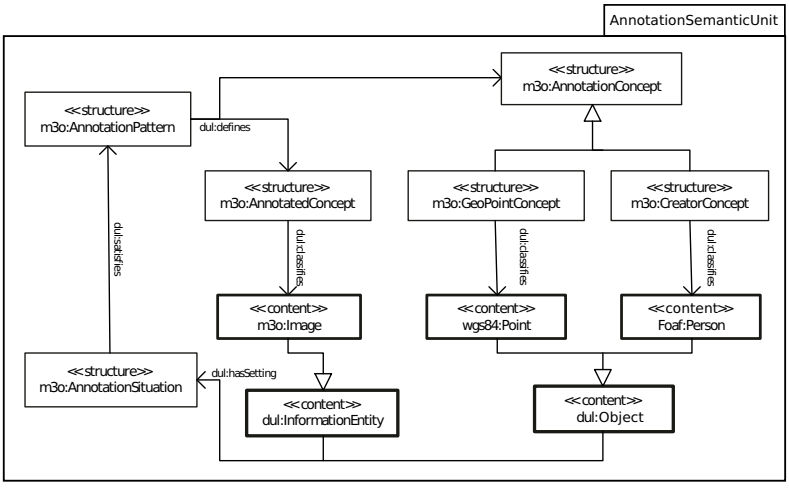

Fig. 6. The Annotation ***Semantic Unit*** in the MoOn

In the second step, the MoOn-based representation is transformed to the Ontology API Model (OAM). The OAM provides a declarative representation of API properties that cannot be tied to the structure of the ontology, like legacy API integration or method behavior customization. In addition, the OAM enables to embed information relevant for the code generation process, e.g., to tailor the concrete API to a particular repository backend. Figure 3 shows the OAM for our running example. Finally, the code is generated from the OAM in fully automated manner.

In the following sections, we describe the different transformation steps along the example from Section 2 in more detail and associate the design decision with the requirements from the previous section.

4.1 Step 1: From Ontology T-Box to MoOn

MoOn is based on an adaptation of the ECore Metamodel for OWL2[9]. The transformation of OWL-based ontology entities into a MoOn representation is inspired by the OWL-to-UML mappings described in [6,4,11], see the discussion on mapping models in the related work in Section 6. A MoOn model for an ontology results from two different steps: First, a fully automatic transformation of the ontology in an ECore model. Second, a manual extension of this ECore model with declarative information about the pragmatic programming access.

Transformation from OWL to MoOn: First, we have to represent the ontology in the MoOn. This preparation of the MoOn includes the representation of all relevant concepts, see (R1). For this reason, ontologies distributed over multiple files are accumulated, imports in the ontology are resolved, see (R8) and implicit knowledge of the ontology is materialized using reasoning. After these steps, we substitute anonymous concepts by named concepts (R9). This is easily possible in Description Logics based languages as OWL by just naming all anonymous classes. In a last step, we consider the parts of the inheritance structure that are carried over to the MoOn-based ontology representation (compare Figure 1 and Figure 6). To adapt the inheritance structure in MoOn-based ontology representations to our needs, the proper concepts from the inheritance structure are selected, e.g., by choosing the least common super-concept (R3).

Adding Declarative Information to the MoOn: The next step is to add responsibility-driven information, i.e., information about how to use ontology concepts in context of the applications. The user defines ***semantic units*** and allocate concepts to them, see (R4). For our example, Jim represents the annotation functionality as ***semantic unit*** and allocates all concepts shown in Figure 6 to it. Additionally, the concepts have to be classified into ***structure concept*** and ***content concepts*** (R2). For Jim, m3o:Image, wgs84:Point and foaf:Person are the ***content concepts*** and m3o:AnnotationPattern, m3o:AnnotationDescription, m3o:AnnotatedConcept and m3o:GeoPointConcept are the ***structure concepts***.

[9] MOF-Based Metamodel for OWL2
`http://www.w3.org/2007/OWL/wiki/MOF-Based_Metamodel`

OntoMDE provides for user support in concept assignment and classification tasks. Based on an existing ***semantic unit*** allocation, OntoMDE suggests for concept classification and based on concept classifications OntoMDE can give advices for ***semantic unit*** allocation.

4.2 Step 2: From MoOn to OAM

The OAM uses the syntax of UML2 with profiles[10] in order to represent the target API. The primary purpose of the OAM is to provide declarative representations of additional information used during code generation. For example, information to integrate a particular repository backend (R6) or information about the integration of legacy API classes (R7). Very important is information about the characteristics of properties such as symmetry or transitivity. This is used to support dedicated method behavior (R5) in the ontology API. The API representation in the OAM is generated fully automatically from the MoOn-based ontology representation. In this transformation, class representations for ***content concepts***, ***semantic units*** and interfaces for the inheritance structure are generated, similar to what Jim did in Section 2.4, (R1,R3,R4). Table 1 summarizes the mappings between MoOn entities and the API enities.

Table 1. Overview of Mappings between MoOn and OAM

MoOn based ontology representation	**Ontology API Model (OAM)**
Content concepts	Content classes & class fields
Content individuals	Content objects
Structure concepts	Class attributes
Structure individual	Individual URI and concept URI
Semantic unit	Pragmatic unit class
Concept properties & relations	Encapsulated in Pragmatic unit classes or class fields
Property characteristics	declarative extension in OAM

Step 3: Generating the Code of an API from the OAM

In the last step, we generate code from the API representation in the OAM. This fully automated process is supported by the OntoMDE toolkit using Java Emitter Templates[11] (JET) as code generation framework.

5 Case Studies and Lessons Learned

The primary objective of our case studies is to demonstrate the applicability of our approach. In addition, we want to show the flexibility and adaptability of the approach.

[10] `http://www.omg.org/technology/documents/profile_catalog.htm`

[11] `http://www.eclipse.org/modeling/m2t/?project=jet#jet` last visit dec 5, 2011.

To show the applicability of our approach, we have developed and applied the OntoMDE toolkit to generate APIs from different ontologies. We have selected ontologies with different characteristics in terms of complexity, level of abstraction, degree of formalization, provenance, and domain-specificity. We have used the OntoMDE toolkit to generate APIs for the Pizza[12] and Wine[13] ontologies. As less formal real world ontologies, we have choosen the Ontology for Media Resources (OfMR)[14] of the W3C and the CURIO[15] ontology used in the We-KnowIt project[16]. And last, we have used OntoMDE to generate APIs for the M3O [13], our running example is based on, and the Event-Model-F (EMF) [14].

To demonstrate the flexibility and adaptability, we used OntoMDE to generate different APIs from the same input ontology, from slightly changed versions of the same ontology and to integrate legacy APIs into our ontology access API. We have selected the M3O ontology ,OfMR aligned with the M3O and an EXIF[17] ontology aligned to the M3O as input ontology for this study. As outlined for our example in Section 2.4, we designed different possible APIs for accessing the M3O. Then, we generated these APIs from the M3O ontology by changing the declarative information about programming access on the MoOn and the OAM. To show the integration capabilities of OntoMDE, we use the OAM to integrate legacy APIs for the Image class in the M3O API.

With the first use case, the generation of APIs for the Pizza and Wine ontologies, we have shown that our approach is capable of processing OWL ontologies, (R1,R9). From applying OntoMDE to multiple ontologies with different characteristics, we can conclude that the general idea of distinguishing concepts into ***content concepts*** or ***structure concepts*** is applicable to all tested ontologies. The concrete sets of ***content concepts*** or ***structure concepts*** strongly depends on the characteristics of the ontology. In simple, less formal ontologies most of the concepts are ***content concepts*** of direct concern for the application. Whereas, in complex ontologies with a high level of abstraction and intense use of reification more of the concepts tend to be ***structure concepts***. The organization of concepts in ***semantic units*** is also applicable to all kinds of ontologies. Again, we encounter differences depending on the characteristics of the ontology. Simple ontologies often only allow for few and usually small ***semantic units***. Complex ontologies allow for multiple partially overlapping ***semantic units*** with potentially many concepts.

We have also investigated the flexibility and adaptability of our approach. Regarding the adaptability, we have integrated the java.awt.image package as legacy APIs for representing images into the APIs of our example. Using the OAM, the integration of the generated API and the legacy API could be conducted in a

[12] The pizza ontology `http://www.co-ode.org/ontologies/pizza/2007/02/12/` last visit dec 5, 2011.

[13] `http://www.w3.org/TR/owl-guide/wine.rdf` last visit dec 5, 2011.

[14] `http://www.w3.org/TR/mediaont-10/` last visit dec 5, 2011.

[15] `http://www.weknowit.eu/content/curio_collaborative_user_resource_interaction_ontology` last visit dec 5, 2011.

[16] `http://www.weknowit.eu/` last visit dec 5, 2011.

[17] `http://www.exif.org/specifications.html` last visit dec 5, 2011.

few steps. As mentioned, we have generated different APIs for the ontology from our example. We have also shown that changes of the API model could be accomplished by modifications on the MoOn, such as "choice of pragmatic units" or "choice of content concepts". As you can see, these changes result in different numbers of pragmatic units and generated concept classes. To demonstrate the flexibility regarding the actual RDF-persistence layer used, we have changed the back-end API of the OntoMDE approach. We used our own RDF-persistence layer Winter [12] as well as the RDF-persistence layer Alibaba[18]. This change of the backend could be conducted within a short time of about one hour. This addresses requirements (R5), (R6), and (R7).

6 Related Work

The problem space of object relational impedance mismatch and the set of conceptual and technical difficulties is addressed frequently in literature, e.g. in [5,15,16,2]. Among others, Fowler provides in his book [1] a wide collection of patterns to common object relational mapping problems. Due to the fact that many problems in persistence and code generation for ontologies are similar to problems from the field of relational databases many approaches utilize object-relational strategies for object-triple problems, for example like ActiveRDF [8], a persistence API for RDF adapting the object-relational ActiveRecord pattern from Fowlers book or OTMj[19] a framework that resembles some of Fowlers patterns to the field of object-triple mapping. Most of the other frameworks, like AliBaba, OWL2Java [6], Jastor[20], OntologyBeanGenerator[21], Àgogo [9], and others, use similar techniques adapting object-relational solutions. An overview can be found at Tripresso[22], a project web site on mapping RDF to the object-oriented world. These frameworks use a simple mapping model for transforming each concept of the ontology into a class representation in a specific programming language like Java or Ruby. Properties are mapped to fields. Only Àgogo [9] is a programming-language independent model driven approach for automatically generating ontology APIs. It introduces an intermediate step based on a Domain Specific Language (DSL). This DSL captures domain concepts necessary to map ontologies to object-oriented representations but it does not captures the pragmatics.

The mappings used to generate the MoOn from the OWL ontologies are based on the work done for the Ontology Definition Metamodel (ODM) [4,11]. The Ontology Definition Metamodel [7] is an initiative of the OMG[23] for defining an ontology development platform on top of MDA technologies like UML.

[18] `http://www.openrdf.org/doc/alibaba/2.0-alpha4/` last visit dec 5, 2011.
[19] `https://projects.quasthoffs.de/otm-j` last visit dec 5, 2011.
[20] `http://jastor.sourceforge.net/` last visit dec 5, 2011.
[21] `http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator` last visit dec 5, 2011.
[22] `http://semanticweb.org/wiki/Tripresso` last visit dec 5, 2011.
[23] `http://www.omg.org/` last visit dec 5, 2011.

7 Conclusion

We have presented with MoOn and OAM a declarative representation of properties of ontologies and their entities with regard to their use in applications and application programming interfaces (APIs). On this basis, we have introduced a multi-step model-driven approach to generate APIs from OWL-based ontologies. The approach allows for user-driven customizations to reflect the needs in a specific application context. This distinguishes our approach from other approaches performing a naive one-to-one mapping of the ontology concepts and properties to the API classes and fields, respectively. With our approach, we alleviate the developers from the tedious and time-consuming API development task such that they can concentrate on developing the application's functionalities. The declarative nature of our approach eases reuseability and maintainability of the generated API. In the case of a change of the ontology or the API, most of the time only the declarative representation has to be adapted and a new API could be generated. In our case studies, we applied our approach to several ontologies covering different characteristics in terms of complexity, level of abstraction, degree of formalization, provenance, and domain-specificity. For our future work, we plan to integrate the support for different method behaviors (see R5) and the dynamic extensibility of ontologies. The support of the dynamic extensibility of ontologies strongly depends on the persistence layer used. Another idea is to use the declarative representation in combination with the ontology to prove consistency of the data representation and manipulation in the API regarding the ontology.

Acknowledgements. This research has been co-funded by the EU in FP7 in the SocialSensor project (287975).

References

1. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Amsterdam (2002)
2. Fussell, M.L. (ed.): Foundations of Object Relational Mapping (2007), http://www.database-books.us/databasesystems_0003.php
3. Gangemi, A., Mika, P.: Understanding the Semantic Web through Descriptions and Situations. In: Meersman, R., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 689–706. Springer, Heidelberg (2003)
4. Hart, L., Emery, P.: OWL Full and UML 2.0 Compared (2004), http://uk.builder.com/whitepapers/0and39026692and60093347p-39001028qand00.html
5. Ireland, C., Bowers, D., Newton, M., Waugh, K.: A classification of object-relational impedance mismatch. In: Chen, Q., Cuzzocrea, A., Hara, T., Hunt, E., Popescu, M. (eds.) DBKDA, pp. 36–43. IEEE Computer Society (2009)
6. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic Mapping of OWL Ontologies into Java. In: SEKE (2004)
7. Ontology Definition Metamodel. Object Modeling Group (May 2009), http://www.omg.org/spec/ODM/1.0/PDF

8. Oren, E., Delbru, R., Gerke, S., Haller, A., Decker, S.: Activerdf: object-oriented semantic web programming. In: WWW. ACM (2007)
9. Parreiras, F.S., Saathoff, C., Walter, T., Franz, T., Staab, S.: à gogo: Automatic Generation of Ontology APIs. In: IEEE Int. Conference on Semantic Computing. IEEE Press (2009)
10. Parreiras, F.S., Staab, S., Winter, A.: Improving design patterns by description logics: A use case with abstract factory and strategy. In: Khne, T., Reisig, W., Steimann, F. (eds.) Modellierung. LNI, vol. 127, pp. 89–104. GI (2008)
11. Rahmani, T., Oberle, D., Dahms, M.: An Adjustable Transformation from OWL to Ecore. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 243–257. Springer, Heidelberg (2010)
12. Saathoff, C., Scheglmann, S., Schenk, S.: Winter: Mapping RDF to POJOs revisited. In: Poster and Demo Session, ESWC, Heraklion, Greece (2009)
13. Saathoff, C., Scherp, A.: Unlocking the Semantics of Multimedia Presentations in the Web with the Multimedia Metadata Ontology. In: WWW. ACM (2010)
14. Scherp, A., Franz, T., Saathoff, C., Staab, S.: F–a model of events based on the foundational ontology DOLCE+DnS Ultralight. In: K-CAP 2009. ACM, New York (2009)
15. Ambler Scott, W.: Crossing the object-data divide (March 2000), `http://drdobbs.com/architecture-and-design/184414587`
16. Ambler Scott, W.: The object-relational impedance mismatch (January 2010), `http://www.agiledata.org/essays/impedanceMismatch.html`
17. Wirfs-Brock, R., Wilkerson, B.: Object-Oriented Design: A Responsibility Driven Approach. SIGPLAN Notices (October 1989)