M2O: A Library for Using Ontologies in Software Engineering

Claudia Pop, Dorin Moldovan, Marcel Antal, Dan Valea, Tudor Cioara, Ionut Anghel, Ioan Salomie
Computer Science Department
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
{claudia.pop, dorin.moldovan, marcel.antal, dan.valea, tudor.cioara, ionut.anghel, ioan.salomie}@cs.utcluj.ro

Abstract— **In this paper we propose an extensible framework over Jena and OWL API that maps complex Java data models onto semantic models based on some custom annotations in order to benefit from the advantages of ontologies in software engineering. Furthermore, it facilitates the implementation of basic CRUD operations for the domain classes and objects, also allowing the definition of new custom operations. We have performed tests on the Stanford Wine ontology, obtaining a code complexity reduction of up to 85% compared to the classical approaches using Jena or OWL API without noticeable performance reduction.**

Keywords—ontology, data models, mapping, class library, software engineering

I. Introduction

In the last two decades, the Internet had a spectacular growth especially due to the development of the World Wide Web (WWW or W3). Primarily, WWW was based on simple web pages containing only formatted documents in the HTML standard. However, in a short time, content was added to these pages, such as images, videos, embedded links, and new technologies emerged. A huge leap was represented by the integration of relational databases with the web pages in order to store the continuously increasing amount of online information in a structured way. Even though relational databases were developed about four decades ago, their popularity increased only recently with the WWW. As a consequence, specialized software companies started to develop frameworks to ease the development of web applications. These became more and more complex and their structure became cumbersome. Even if Object Relational Mapping (ORM) frameworks were developed to hide the complexity of the SQL databases in the back-end of the web applications, in order to perform specific or complex operations programmers still have to write SQL queries and to design databases. Thus, the Database Comprehension Problem appears in the case of complex diagrammatic modeling of the real world making the conceptual models too large for a designer to understand and to manage. Furthermore, the un-natural mode of writing queries on these databases, different from the human language, makes the programmers life even more difficult. A possible solution for these problems is offered by semantic domain modeling.

Since the development of artificial intelligence, in 1970s, researchers have recognized that capturing knowledge is the key of creating an intelligent and autonomous system. Furthermore, ontologies were developed as computational models that enabled reasoning. The new concept of ontology emerged in the field of computer science as a new way of representing knowledge closer to humans. Ontology is a graph of knowledge. It addresses some of the most common needs when talking about representing and describing concepts from a domain. Furthermore, among the numerous advantages brought by the ontologies, we begin with their expressivity, due to the resemblance with the human knowledge representation. Some other important characteristics of the ontologies are their extensibility and flexibility to changes. Adding or removing concepts and individuals is easier compared to the classic relational mapping, where tables and relationships between them must be altered. Last but not least, they offer the possibility of reasoning, leading to an enhancement of the information already stored.

For reducing the database comprehension problem in software development it comes naturally to use ontologies as semantic data models. Many frameworks have been developed in order to provide the possibility of accessing, manipulating and querying ontologies. Among these frameworks the most used and notable ones are: Jena [7] and OWL API [4]. Both APIs provide reliable and scalable implementations for accessing and operating on ontologies. Even if they allow modeling the data in a manner close to the natural language and the developer, it is very complicated from the development point of view, having a very slow learning curve, due to their complex model and functionalities. Furthermore, in order to perform basic operations (for example retrieving data from the ontology) complex code needs to be written, specific to each OOP class and the mapping between the ontology properties and OOP classes attributes needs to be done manually.

We propose a library that allows using the ontology in a similar way with the ORM programming technique in software engineering. It provides a very light interface for accessing the ontology and reduces the code complexity, by providing one line methods for performing basic operations (create, update, delete, find). It uses reflection to parse the Java entities, hiding in this way the code complexity needed by APIs like Jena and OWL API to perform operations on ontologies, at the same time, benefitting from the performance and scalability properties offered by these. Furthermore, it is the first open-source tool that offers the functionality of

978-1-4673-8200-7/15/$31.00 ©2015 IEEE

generating semantic ontological model from an object-oriented model design.

The rest of the paper is structured as follows: Section II shows related work, Section III presents the proposed framework for mapping an object oriented domain model to a semantic model, Section IV presents numerical simulation based experiments and results, while Section V concludes the paper and presents the future work.

II. Related Work

In order to integrate semantic models into software applications, special frameworks that manipulate ontologies were designed. One of the most used tools in software engineering that allow working with ontologies are the Web Ontology Language Application Programming Interface (OWL API) , described in [4,5,6] and the Jena Semantic web development framework that can be consulted in [7, 8,9,10,11]. One of the disadvantages of using OWL API is represented by the fact that this framework supports the manipulation of the OWL ontologies only at a specific level of abstraction, different from the Resource Description Framework (RDF) level. This problem does not exist in Jena, which contains an API that can be used for the extraction and for the insertion of data in RDF graphs and OWL ontologies and it provides different ways to query the information such as RDQL for RDF and SPARQL for OWL.

Furthermore, because the frameworks presented above need an identical copy of the semantic domain model in the software domain model, some special libraries started to emerge. They can be classified in three main categories: the first category is represented by the frameworks that are trying to map an object oriented (OOP) model to an existing semantic model, or reverse. However, a major disadvantage of this category is the existence of both models at the beginning of the development. The second category is represented by those frameworks which generate the object oriented code from existing ontologies. And finally, the third category is represented by software tools that generate ontologies from OOP code, thus easing the development of semantically enhanced software applications.

Most of the frameworks from the first category define a mapping between existing Java object oriented models and an existing ontology, having the major disadvantage of model duplication. One of such frameworks is the MOOT framework [12], an approach which allows the transformation of abstract ontological concepts into everyday programming languages. The authors propose the following reasons why such a mapping between ontologies and everyday programming languages is needed: the transformation of the concepts from the ontologies into every-day programming languages would increase the adoption of the ontologies in the solving of many engineering tasks [13] and the large number of similarities between the ontological world and object-oriented world [14] is an inspiration for the researchers to find other solutions to access the data which is represented semantically. The MOOT framework maps some of the components of OWL 2 to the Java programming language. The model is universal as it supports multiple ontological languages and multiple programming languages. Another such approach is presented in [16], where the authors show a method of object-ontology mapping. The paper presents how to map ontologies to object-oriented representations and how to map object-oriented representations to ontologies. One of the most important elements of the architecture for object-ontology mapping is represented by CRUD support. The classes from the ontology correspond to classes from the object-oriented programming language. Furthermore, in [17] it is described how to map OWL individuals to pre-generated Java ontology classes. The mapping process is unidirectional as it shows how to map from OWL to object-oriented concepts only. The other direction of mapping (from object oriented concepts to OWL classes) is not discussed. JOM (Java Ontology Mapper) uses Jena framework and Java programming language. Paper [19] presents Java2OWL, a system that can be used for the synchronization of Java class hierarchies with OWL concept hierarchies, by using some extra annotations in the Java class files. By combining Java and OWL, Java may be used for the computation while OWL can be used for the retrieval of the individuals that correspond to some OWL concept expressions. Java2OWL assumes that there exists a “background ontology”. Java classes that will be mapped to the ontology classes must be annotated.

Frameworks classified in the second category generate Java code from OWL ontologies in order to facilitate development by eliminating model duplication. However, the generated code may be incomplete because of the complex semantic relationships such as multiple inheritances. One framework from the second category that generates Java code from OWL ontologies is he Ontology Bean Generator [18]. It may create Java class source files which illustrate the logical structure of the ontology designed in Protégé. Another such framework is Sapphire [21], a tool that can be used for the generating of Java Runtime artifacts from OWL Ontologies. The Sapphire tool is used to generate byte code for a collection of Java interfaces which correspond to a collection of OWL ontologies. Some of the challenges related to the mapping between OWL and Java are the following ones: the mapping of OWL classes to Java interfaces is a technique for the approximation of the OWL’s multiple inheritance, OWL properties have rich descriptions, the OWL models make open-world assumptions, and OWL classes may be defined as union, intersection or complement of other classes. In the case of Sapphire, the OWL classes are mapped to Java interfaces.

Regarding the third software category, there are some approaches to try to generate ontologies from Java code, but they rely on comments or they try to generate only partial ontologies, not complete domain models identical to the OOP Java model. One such approach is Semlet presented in [15], which is a customizable doclet for ontology extraction. OWL ontologies are extracted from Java libraries by using Javadoc technology (a tool for automatic extraction of documentation) and Jena (a Java framework that can be used for the construction of semantic web applications). Some of the assumptions used in the construction of this framework are the following ones: a class is always a class (a Java class must correspond always to an ontology class), two classes from Java which have the same name but correspond to different packages must have different names in the ontology (this

problem can be solved by using namespaces), access level modifiers (public, protected, and private) indicate which elements participate in the construction of the ontology, interfaces from Java are treated as OWL classes, methods correspond to object properties, fields are translated into either data type properties or object properties and so on. Another framework is presented in [20]. The authors discuss how to generate the ontology from the Java source code. The framework's aim is to extract the methods from a project and to store metadata associated to these methods in the ontology. The metadata is extracted from the code by using QDox code generators, and the information is stored in OWL using the Jena framework. Information which is stored in OWL includes: classes, methods, return type and parameters. QDox may be used for the extraction of classes, interfaces and method definitions from source code, while the metadata extracted by QDox may be stored by using the Jena Framework.

As opposed to the presented state of the art work, the framework proposed by us eases the integration of ontologies into software applications by allowing the generation of a semantically enhanced model from an existing annotated Java model, thus belonging to the third category of frameworks. As result, a complete ontological model from the Java model is generated, thus reducing development time. Furthermore, it generates methods for basic CRUD access of the ontology, decreasing the complexity of the application. Last but not least, for manipulating the ontology it offers support for Jena, OWL API and other library that can be easily integrated because of the extensibility of the design. As far as we know, this is the first framework that offers these complete features of generating ontologies from Java models.

III. The M2O Library

When creating a data model using data from ontology, the process of mapping Java classes to the Ontology classes can be very complex. M2O library is designed to ease the process of integration in Java, providing direct access to the ontology data by mapping directly the data model (Java beans) to the ontology classes. Contrary to the existent libraries presented in section II, M2O purpose is to map the object-oriented model to ontology. It offers the possibility of creating an ontology code-first, or mapping the classes to an existent corresponding ontology and then applying basic operations on the classes of this ontology (create, update, delete, select all individuals, select specific individual). All the available operations are implemented based on existing API's such as Jena and OWL API. The proposed library is a wrapper over these APIs, providing a generalized way of accessing the ontology.

TABLE I. OOP TO ONTOLOGY MAPPING

OOP	***Ontology***
Class	Class
Instance	Individual
Field	Data Property
Annotated fields : @ObjectProperty (*)	Object Property

(*) The annotated field is the range and the current object is the domain

In order to be able to map the object oriented paradigms to ontology, some rules are applied (see TABLE I), which are meant to find the equivalence between the OOP principles and the ontology principles.

A. Arhitecture

The application (M2O library) is implemented using a well modularized architecture. The library architecture is presented in Fig. 1. It was designed to ensure reliability and scalability. Furthermore, due to its low coupled modules, the application can be easily extended, by plugging in other ontology APIs (similarly to the ones already used: Jena and OWL API).

Object Abstraction **-** An abstraction layer is needed between the POJO classes and the ontology APIs. In order to achieve this level of abstraction, intermediate classes have been defined by implementing the *OntologyModel* interface. The ontology object parser, *OntologyEntityReflectionParser* is a utility class responsible to create *OntologyModel* instances based on the Java class received as input. Depending on the operations performed, there are two types of *OntologyModel* objects.

Firstly, the *OntologyClass* contains all the information necessary for creating a class in the ontology. The ontology object parser receives a Java class as input, and extracts all the information regarding the classname, the fields –String, and their type – Object (*Map<String,Object> fields*), the object properties – String, and their range- Object , and stores them in key-value data structures(*Map<String,Object> objectProperties*). A list of *OntologyClass* objects is created from all the ontology classes defined in Java, containing the representation of all the ontology entities which are then passed to the *OntologyAccessManager* for generating code-first ontology.

Secondly, the *OntologyIndividual* contains all the information for creating an individual in ontology. The ontology object parser is responsible for creating an *OntologyIndividual* object, from a Java object received as parameter. The created object contains all the information regarding the java object's classname, fields and their values (*Map<String,List<Object>> fieldsValues*) together with the object properties associated to existing instances from the ontology(*Map<String,List<Range>> objectPropertiesValues*). This *OntologyIndividual* object is then used by the *OntologyAccessManger* to perform operations like: create and update individual in ontology.

TABLE II. OPERATIONS EXPOSED BY M2O

Return type	***Method***
void	create(T entity)
void	update(T entity)
T	findByIdentifier(V identifier)
List<T>	findAll()
void	delete(V identifier)

Ontology Repository - The *OntologyRepository* provides the interface to the M2O library. The basic operations (see

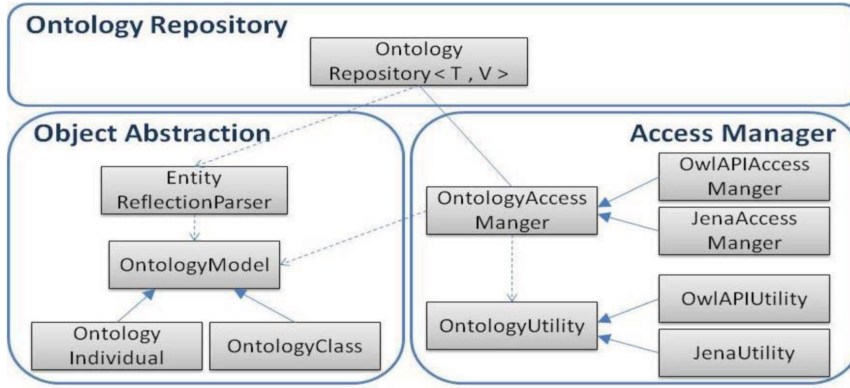

Fig. 1 M2O architeture

TABLE II) are defined at this level. The operations are defined on the generic type T and V. The type T represents the entity and the type V represents the identifier of the individual (the name of the individual, or an ID if entities are also mapped to a database). These types will be specified by the user of the library by extending the *OntologyRepository* class. Each method that receives an entity object as a parameter uses the *OntologyEntityReflectionParser* to obtain the equivalent *OntologyModel* object. This object is then passed to the *OntologyAccessManager* object to perform the actual operation. A creational design pattern (factory design pattern) is used to instantiate the *OntologyAccessManger*. Based on the API specified in the configuration file, the *OntologyAccessManagerFactory* returns a *JenaAccessManger* object or a *OwlAPIAccessManger* object.

Access Manager **-** The *OntologyAccessManger* interface defines the basic operations (see Table III) needed to be implemented by each API's access manger class. Different implementations of access manager are required for each API (Jena and OWL API).

TABLE III. ONTOLOGY ACCESS OPERATIONS

Return type	***Method***
void	addIndividual(OntologyIndividual individual)
void	updateIndividual(OntologyIndividual individual)
T	getIndividual(Class<T> cls, V identifier)
List<T>	getIndividuals(Class<T> class)
void	deleteIndividual(V identifier)
void	insertOntologyClasses()
void	insertOntologyClass(OntologyClass class)

The *OntologyAccessManger* provides a generalized mechanism in order to perform the basic operations, using the previously defined abstract data model (*OntologyClass* and *OntologyIndividual*). For example, by calling the *addIndividual* method on the *JenaAccessManager* object, a *com.hp.hpl.jena.ontology.Individual* object will be created and stored in the ontology, using the fields and the object properties contained in the *OntologyIndividual* class received as parameter. Similarly, *OwlAPIAccessManager* will create an *org.semanticweb.owlapi.model.OWLNamedIndividual* object. This is possible due to the abstract data model used (*OntologyIndividual*). In this way the access manager classes do not need any concrete information regarding the entities that are stored in the ontology, all the operations being executed in a generic way. Each *OntologyAccessManager* implementation class (*JenaAccessManger* and *OwlAPIAccessManager*) is implemented using the Singleton Design Pattern. In this way, a single point of access to the ontology is provided. Some utility classes are defined to provide functionalities like: load ontology, save ontology, save snapshot (the state of the ontology at a given time), etc. The *OntologyUtility* interface provides a contract for these operations. The ontology utility classes (*JenaUtility* and *OwlAPIUtility*) are used by the access manager (*OntologyAccessManger* implementing classes), but can also be used by the library's user in order to save and make snapshots of the ontology.

B. *Custom Defined Annotations*

Similarly with the Hibernate [1] model, the ontology entities will need a way to specify the metadata needed in order to perform the mapping between the Java model and the ontology. The method used for this library is annotation-based mapping metadata. In order to achieve this, the following annotations are defined:

@OntologyEntity - It is a class annotation that will be used for every class that needs to be mapped to the ontology.

@InstanceIdentifier - It is a field annotation specifying the field that will be used to identify individuals. Similarly to the data base approach, this identifier will ensure the uniqueness of the individual and in the same time will allow the user to load the individual based on this identifier.

@ObjectProperty(*value* = *"objProp"*, ***range*** = *Some.class***)**- This field annotation will be used in order to specify an object property of the current class. The "value" field will contain the name of the object property and using the "range" field, the class of the range object will be specified.

@OntologyIgnore- In order to ignore a field, this annotation will be used. As a result, this field will not be considered when mapping the class/instance to the ontology class/individual.

As an example we will use classes from the well-known Wine ontology proposed by Stanford University [3]. The

ontology entity class represented in Fig. 2 is the wine class. It has different object properties like: WineGrape, WineColor, WineBody, WineFlavor, WineSugar, Winery, and Region; which are all represented in the Wine Java class. The @OntologyEntity annotation is exemplified on the first line, as a class annotation on the Wine class. At line 4, the @InstanceIdentifier is exemplified, the wine being identified by its name. The object properties are enumerated (lines 7-22), while the last annotation (line 25) exemplifies an ignored field.

```
1   @OntologyEntity
2   public class Wine extends PortableLiquid {
3
4       @InstanceIdentifier
5       private String name;
6
7       @ObjectProperty(value = "madeFromGrape", range = WineGrape.class)
8       private WineGrape wineGrape;
9
10      @ObjectProperty(value = "hasColor", range = WineColor.class)
11      private Descriptor color;
12      @ObjectProperty(value = "hasBody", range = WineBody.class)
13      private Descriptor body;
14      @ObjectProperty(value = "hasFlavor", range = WineFlavor.class)
15      private Descriptor flavor;
16      @ObjectProperty(value = "hasSugar", range = WineSugar.class)
17      private Descriptor sugar;
18
19      @ObjectProperty(value = "hasMaker", range = Winery.class)
20      private Winery maker;
21
22      @ObjectProperty(value="locatedIn", range=Region.class)
23      private Region location;
24
25      @OntologyIgnore
26      private String wineAcidity;
```

Fig. 2. Wine Ontology Entity

IV. USE CASE AND VALIDATION

In order to demonstrate how the library is used a use case scenario based on the Wine ontology is presented in this section. Due to the complexity of the Wine ontology the scenario considers the following classes (see Fig. 3): *PortableLiquid, PinotNoir, Region, WineBody, WineColor, Descriptor, WineTaste, WineFlavor, WineGrape, WineSugar, Winery*. The basic operations on the ontology are presented using the defined entities.

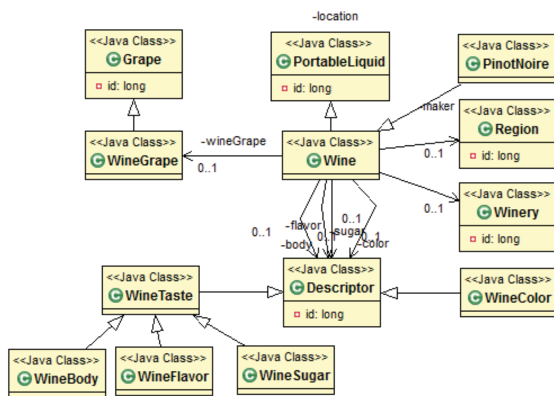

Fig.3 Java Model class diagram

The setup of this library is done via a configuration file (ont-config.config). The following parameters need to be setup in order to ensure the correct functionality of the library.

1. **ONT_FILE**= *path to the owl file.* The ontology file needs to be specified. If the ontology already exists, it will be loaded from this path; otherwise a file will be created containing an ontology corresponding to the Java model -
2. **ONT_URI** = *ontology URI* .The identifier of the Ontology (Uniform Resource Identifier)
3. **API_TYPE**=*JENA or OWLAPI.* Based on the user preferences, the API needs to be specified in this file. The API's that are available so far are : JENA and OWL API
4. **ENTITIES_PACKAGE**=*ro.tuc.dsrl.m2o.example.entities*. The user must specify the package that contains the classes that are going to be mapped to the ontology
5. **AUTO_GEN**=*true or false*. According to this flag, an ontology will be created based on the entities from the specified package. Otherwise the specified ontology will be loaded from the file.

In order to prove the accuracy of the code first generation functionality, a snippet of the entire Wine Ontology is created based on the above mentioned entities. The diagram of the created ontology is presented in Fig. 4.

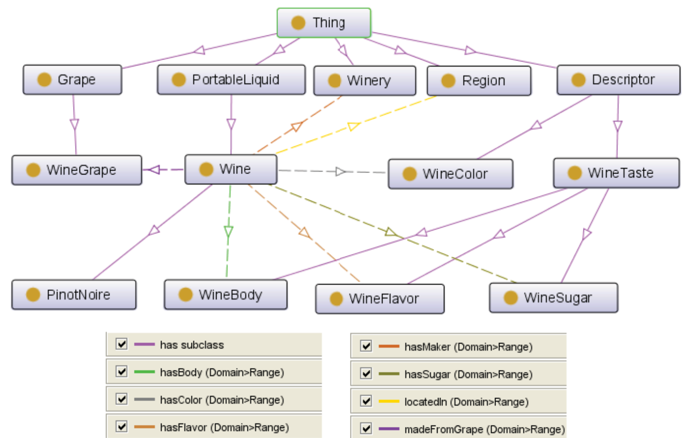

Fig. 4. Wine Ontology diagram snippet

In Fig. 5, the basic operations are implemented on the Wine entity. In order to achieve this, repository classes were defined for each entity of the model. It is assumed that all the entities corresponding to the object properties were already inserted in the ontology.

```
1   WineRepository repo = new WineRepository();
2   Wine wine = new PinotNoire("LateTannerPinotNoire", grape, wColor,
3                              wBody, wFlavor, wSugar, winery, region);
4   //Create Wine
5   repo.create(wine);
6
7   //Update Wine
8   wine.setLocation(region2);
9   repo.update(wine);
10
11  //Retrieve all wines;
12  List<Wine> allWines = repo.findAll();
13
14  //Retrieve wine by identifier
15  Wine retrievedWine = repo.findByIdentifier("LateTannerPinotNoire");
16
17  //Delete wine
18  repo.delete("LateTannerPinotNoire");
19
```

Fig. 5. Basic Operations defined on a Wine Ontology Entity

A. Code complexity reduction

Considering that a data model layer is already implemented (containing all the modeled classes) we evaluate the code complexity of the basic operations on the Wine class. Let us consider a scenario where individuals of the following classes are already inserted in the ontology: WineBody, WineColor, WineFlavor, Region, Winery, WineSugar and WineGrape. In order to insert a Wine entity from Java in ontology, we will provide the code snippets needed to achieve this operation in all of the following libraries: OWL API, Jena, and M2O. In Fig. 6, lines 1-13 represent the code needed to insert the Wine individual in ontology using the OWL API library. Lines18-24, the association for one object property is depicted.

```
1  OWLDataFactory factory = instance.getFactory();
2  OWLOntologyManager manager = instance.getManager();
3  OWLOntology ontology = instance.getOntology();
4  // create an individual of type Wine
5  OWLClass wineClass = factory.getOWLClass(IRI.create(NAMESPACE + "Wine"));
6  OWLNamedIndividual wineIndividual = factory.getOWLNamedIndividual
7                      (IRI.create(NAMESPACE + "Wine" + wine.getName()));
8  OWLClassAssertionAxiom classAssertion = factory.
9                      getOWLClassAssertionAxiom(wineClass, wineIndividual);
10 manager.addAxiom(ontology, classAssertion);
11 OWLDataProperty hasId = factory.
12                     getOWLDataProperty(IRI.create(NAMESPACE + "hasIdentifier"));
13 OWLDataPropertyAssertionAxiom hasIdAssertionAxiom = factory.
14                     getOWLDataPropertyAssertionAxiom(hasId, wineIndividual, wine.getName());
15 manager.addAxiom(ontology, hasIdAssertionAxiom);
16
17 // associate the Wine individual with a WineGrape individual
18 OWLObjectProperty madeFromGrape = factory.
19                     getOWLObjectProperty(IRI.create(NAMESPACE + "madeFromGrape"));
20 OWLNamedIndividual wineGrapeIndividual = factory.
21                     getOWLNamedIndividual(IRI.create(NAMESPACE + "WineGrape" + "_"+ wine.getWineGrape()));
22 OWLObjectPropertyAssertionAxiom madeFromGrapeAssertionAxiom = factory.
23                     getOWLObjectPropertyAssertionAxiom(madeFromGrape, wineIndividual, wineGrapeIndividual);
24 manager.addAxiom(ontology, madeFromGrapeAssertionAxiom);
```

Fig. 6 OWL API: Insert operation for Wine entity

Similarly, in Fig. 7 the Jena code is presented for inserting a Wine individual into ontology associating only one of the seven object properties.

```
1  OntClass wineClass = instance.getOntModel().getOntClass(NAMESPACE + "Wine");
2  Individual wineIndividual = instance.getOntModel().createIndividual
3                              (NAMESPACE + wine.getName(),wineClass);
4  DatatypeProperty datatypeProperty = instance.getOntModel().
5                           getDatatypeProperty(NAMESPACE + "hasIdentifier");
6  wineIndividual.addProperty(datatypeProperty, wine.getName(), XSDDatatype.XSDstring);
7
8  // associate the Wine individual with a WineGrape individual
9  Individual wineGrapeIndividual = instance.getOntModel().getIndividual(
10         NAMESPACE + "WineGrape" + "_" + wine.getWineGrape().getName());
11 ObjectProperty madeFromGrape = instance.getOntModel().
12                          getObjectProperty(NAMESPACE + "madeFromGrape");
13 wineIndividual.addProperty(madeFromGrape, wineGrapeIndividual);
```

Fig. 7 Jena: Insert operation for Wine entity

In Fig. 8, the insertion of the Wine entity is done using the M2O library. As a result, in order to perform a basic insert operation on the Wine entity (considering all seven object properties) from Jena, 32 lines of code were written. Similarly, for OWL API, 56 lines of code were necessary, while using M2O this was possible by writing 3 lines of code.

```
1  WineRepository repo = new WineRepository();
2  Wine wine = new PinotNoire("LateTannerPinotNoire", grape, wColor, wBody,
3                              wFlavor, wSugar, winery, region);
4  //Create Wine
5  repo.create(wine);
```

Fig. 8 M2O: Insert operation for Wine entity

We will measure how much the code complexity will be reduced using our API, by considering the information regarding the wine entity. Table IV shows how many lines of code are required in order to perform the basic operations in the case of using OWL API, JENA and M2O.

TABLE IV. CODE COMPLEXITY IN OWL API VERSUS JENA VERSUS M2O

	OWL API	JENA	M2O
CREATE	56	32	3
GET BY ID	45	38	1
GET ALL INDIVIDUALS	48	41	1
UPDATE	62	37	3
DELETE	6	5	1

We show how much the code complexity is reduced in the case of an entire project based on the wine ontology. The ontology contains: 150 classes out of which, 74 are wine subclasses and 17 properties. In order to test the code complexity reduction, let us consider having more classes similar to the Wine class in the ontology (classes having 7 object properties, but considering these already inserted in the ontology). We computed the number of lines of codes based on the results obtained in Table IV. We will vary the number of classes from 1 to 20 classes, and we will compare the result between the three APIs. According to results presented in Fig. 9, the code is reduced with 80% for Jena and 85% for OWL API.

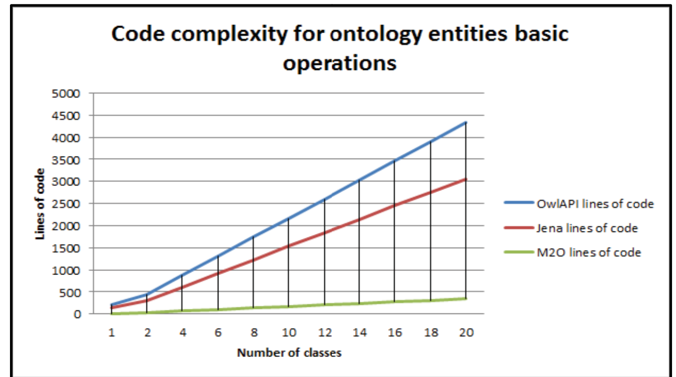

Fig. 9. Code complexity statistics

B. Performance evaluation for insertion and retrieval of information

In order to realistically evaluate the implemented library, the time of execution for the operations needs to be considered as well. We will measure the time necessary to insert and to retrieve information from the ontology related to wine concepts. The statistics address the time needed for inserting and retrieving wine objects that contain all their fields set with concepts already existing in the ontology (Descriptors, Winery, Region, etc.).

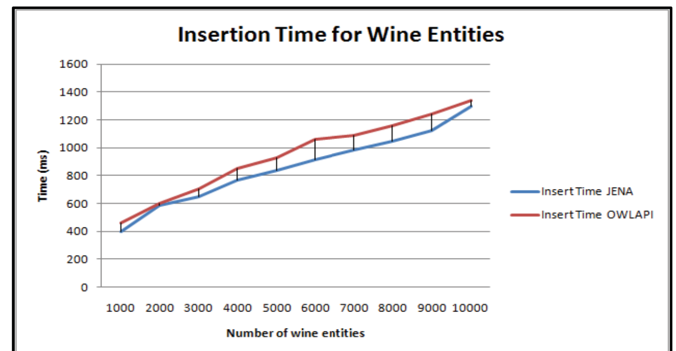

Fig. 10. Insertion time for wine entities

First, we will test the insertion time for different numbers of Wine entities. We vary the number of entities between 1000 wine entities and 10000 wine entities. The tests are run both using the M2O library, first as a wrapper over the OWL API library, and then as a wrapper over the Jena library. The results presented in Fig. 10 show better results for Jena, although the differences are small between the two libraries regarding the insertion time.

Next we will measure the time necessary to retrieve information about the wine individuals from the ontology by using again the two alternatives: OWL API and JENA.

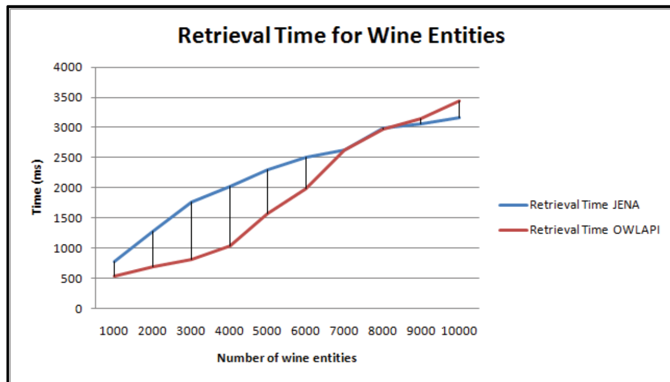

Fig. 11. Retrieval time for wine entities

The overall retrieval time is greater in the case of using JENA than in the case of using OWL API. But as the results show in Fig. 11, the OWL API shows a more abrupt rising in execution time, once the number of entities increases. For the retrieval process, an eager loading approach is used. This means that together with the wine entity all the other entities are loaded (Descriptors, Winery, Region, etc.).

V. Conclusion

In this paper we have presented a novel framework for integrating semantic modeling into complex software projects. Our framework will allow software engineers to benefit from the semantic advantages brought by ontologies in data modeling while eliminating the major problems encountered with integrating ontologies into object oriented applications, such as cumbersome development, high code complexity and slow learning curve for developers. We have tested the framework on the Wine Ontology, obtaining promising results. The code was reduced with up to 85% compared to the classical approaches using Jena or OWL API while the performance of the application remained the same. As future development we propose to design a generic approach for generating custom queries based on the syntax of a method's name (e.g. generate the query for retrieving entities named X for the method named *findByName(String X)*).

Acknowledgements

This work has been carried out in the context of the Ambient Assisted Living Joint Programme project Elders-Up! [2] and was supported by a grant of the Romanian National Authority for Scientific Research, CCCDI – UEFISCDI, project number AAL26/2014. This document is a collaborative effort. The scientific contribution of all authors is the same.

Refrences

[1] Hibernate - http://hibernate.org/

[2] Elders UP! EU AAL Project– http://www.eldersup-aal.eu

[3] Natalya F. Noy and Deborah L. McGuinness. ``Ontology Development 101: A Guide to Creating Your First Ontology''. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.

[4] Matthew Horridge, Sean Bechhofer, "The OWL API: A Java API for OWL Ontologies", Journal Semantic Web, Volume 2 Issue 1, January 2011, pp. 11-21

[5] Sean Bechhofer, Raphael Volz, Phillip Lord, "Cooking the Semantic Web with the OWL API", The Semantic Web – ISWC 2003, Lecture Notes in Computer Science, Volume 2870, 2003, pp. 659 – 675

[6] Sean Bechhofer and Nicolas Matentzoglu, University of Manchester, "The OWL API: An Introduction"

[7] Ayesha Ameen, Khaleel Ur Rahman Khan and B. Padmaja Rani, "Reasoning in Semantic Web Using Jena", Computer Engineering and Intelligent Systems, Vol. 5, No. 4, 2014

[8] Kruti Jani, Dr. V.M. Chavda, "A Study on Semantic Web Framework: Jena and Protege", Indian Journal of Applied Research, Volume 4, Issue 1, Jan 2014, pp. 143-145

[9] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, "Jena: Implementing the Semantic Web Recommendations", HP Laboratories Bristol, December 24th, 2003

[10] G. Klyne, J.J. Carroll, "RDF Concepts and Abstract Syntax", 2003, W3C

[11] Max Volkel, "RDFReactor – From Ontologies to Programmatic Data Access"

[12] Rok Zontar, Ivan Rozman, Vili Podgorelec, "Mapping Ontologies to Objects using a Transformation based on Description Logics", INFORMATION TECHNOLOGY AND CONTROL, 2014, T. 43, Nr. 3, pp. 230-243

[13] V. Podgorelec, M. Gresak, "Supporting the Study Process using Semantic Web Technologies", Electronics and Electrical Engineering, 2011, Vol. 116, No. 10, pp. 105-108

[14] R. Zontar, M. Hericko, "Adoption of object-oriented software metrics for ontology evaluation", In: Proceedings of the Fifth Balkan Conference in Informatics, Novi Sad, Serbia, 2012, pp. 298-301

[15] Davide Ancona, Viviana Mascardi, Ombretta Pavarino, "Automatic ontology extraction from Java libraries for machine-readable API documentation", 9th May 2010

[16] Peter Bartalos, Maria Bielikova, "An approach to object-ontology mapping", 2007

[17] Hanno-Felix Wagner, "JOM – The Java Ontology Mapper, Mapping OWL individuals to pre-generated Java ontology classes", Essen, 30th July 2012

[18] Protégé Wiki: OntologyBeanGenerator http://protegewiki.stanford.edu/index.php/OntologyBeanGenerator,

[19] Hans Jurgen Ohlbach, "Java2OWL A System for Synchronizing Java and OWL Version 1.1", Research Report PMS-FB-2012-2, March, 2012

[20] Gopinath Ganapathy, S. Sagayaraj, "To Generate the Ontology from Java Source Code OWL Creation", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 2, No. 2, Feb. 2011

[21] Graeme Stevenson, Simon Dobson, "Sapphire: Generating Java Runtime Artefacts from OWL Ontologies", Advanced Information Systems Engineering Workshops, CAiSE 2011 International Workshops, London, UK, June 20-24, 2011, pp. 425-436