Ontology-based documentation extraction for semi-automatic migration of Java code[*]

Davide Ancona
davide@disi.unige.it

Viviana Mascardi
mascardi@disi.unige.it

Ombretta Pavarino
ombretta.pavarino@virgilio.it

DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy

ABSTRACT

Migrating libraries is not a trivial task, even under the simplest assumption of a downward compatible upgrade. We propose a novel approach to partially relieve programmers from this task, based on the simple observation that class, method and field names and comments contained in a Java library should be a good approximation of its semantics, and that code migration requires knowing the semantic similarities between the two libraries.

Following this assumption, we borrow the main concepts and notions from the Semantic Web, and show how (1) an *ontology* can be automatically generated from the relevant information extracted from the code of the library; (2) semantic similarities between two different libraries can be found by running a particular *ontology matching* (a.k.a. *ontology alignment*) algorithm on the two ontologies extracted from the libraries. The main advantages of the approach are that ontology extraction can be fully automated, without adding ad-hoc code annotations, and that results and tools produced by the Semantic Web research community can be directly re-used for our purposes.

Experiments carried out even with simple and efficient freely available matchers show that our approach is promising, even though it would benefit from the use of more advanced ontology matchers possibly integrated with a component for checking type compatibility of the computed alignments.

[*]This work has been partially supported by MIUR DISCO Distribution, Interaction, Specification, Composition for Object Systems.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms

Documentation

Keywords

Code migration, ontology, ontology matching

1. INTRODUCTION

Migrating code for replacing a used library l with a new one l' may be a non trivial, resource consuming, and error prone task, even under the simplest case where l' is just a downward compatible upgrade of l. In this paper we will focus on the Java language, even though the proposed approach could be applied to any other programming language.

Let us consider the problem of migrating a Java application from JDK API 1.0 to 6.0. If such an application uses the deprecated method **boolean** `handleEvent(Event)` in class `java.awtComponent`, then a correct migration should replace such a method with **void** `processEvent(AWTEvent)`; however, automatically finding such a replacement is far from being simple, since the two methods are not even type compatible (neither the parameter types, nor the return types are compatible).

A possible solution to this problem consists in adopting a notion of *semantic similarity* between classes, methods, and fields, by considering names and comments as a good approximation of the semantics of a library; such information can be easily extracted automatically from the code. With this premise, we can borrow the main concepts and notions from the Semantic Web: the approximated semantics of a Java library consists of (the meaning of) its names, and of its comments, and the relationship existing among such names.

This approach allows an application to infer the semantic similarity of classes `Event` and `AWTEvent`, and methods `handleEvent` and `processEvent`. This can be achieved by (1) automatically generating an *ontology* from the relevant information extracted from the code of the library; (2) finding the semantic similarities between two different libraries by running a particular *ontology matching* (a.k.a. *ontology alignment*) algorithm on the two ontologies extracted from the libraries. This approach has several advantages.

Extraction of semantic information and generation of the corresponding ontology from a library can be fully automated, and no additional specific semantic annotations are required; based on the Javadoc technology, we have implemented an ontology extractor with a rather simple doclet (less than 1K LOC) suitable to be easily extended in order to get more advanced ontology extractors.

Another advantage consists in the possibility of reusing results and tools produced by the Semantic Web research community: the ontology extractor we have developed makes intensive use of Jena [4], a Java framework for building Semantic Web applications. Furthermore, the experiments we have carried out to compute the semantic similarities between entities of different Java libraries are based on Alignment API [6], a well-known open source tool that implements several simple and efficient ontology matching algorithms.

Information extracted and organized in an ontology can be fruitfully used for aiding developers during migration of software from an old library to a new one; in particular, our experiments show that programmers can be partially relieved from the burden of deducing from the documentation the correct correspondences between entities of the old and of the new library. This task is time consuming and in some cases may be also complex; our initial example shows that even when migration involves different versions of the same library, replacing just the deprecated entities is not enough: for instance, as of JDK 1.1, class `java.awt.Event` is obsolete (but not deprecated), but it should be replaced by class `java.awt.AWTEvent`. Migration becomes even more complex when it involves two different libraries where one is not an upgrade of the other, and, therefore, more serious compatibility issues may arise.

In general, automatic extraction of ontology-based documentation from code opens up other interesting possibilities: alignments between ontologies extracted from libraries l and l' may provide software metrics which are useful to get an estimate of the costs needed for migrating software from l to l'. Finally, advanced web search engines for domain specific libraries could fruitfully exploit automatic extraction of ontology-based documentation from software, to assist programmers during the software development process.

The paper is structured as follows. Section 2 provides the necessary background on OWL (Web Ontology Language), ontology matchers, Jena, and Javadoc, Section 3 presents the main design aspects and some implementation details of the ontology extractor we have developed. Experiments carried out with four ontology matchers are presented in Section 4. Finally, Section 5 is devoted to related work, and Section 6 draws conclusions.

2. BACKGROUND

In this section we introduce the OWL ontology language (Section 2.1), the Jena framework (Section 2.2), the ontology matching problem (Section 2.3), and the Javadoc tool (Section 2.4).

2.1 The Web Ontology Language OWL

In computer science, an ontology [11] formalizes a domain of knowledge by means of primitive concepts, called *classes* and *properties*, and their relationships. The definitions of classes and properties include information about their meaning and constraints on their logically consistent application.

OWL [19] is the most widespread language for representing ontologies, and it is considered a de-facto standard. OWL provides three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full; for our purposes OWL Lite has proved to be the most suitable choice.

For brevity, we summarize only the few aspects needed to understand the technical details contained in the paper. In the examples we use the more compact functional-style syntax, instead of the more concrete (but also verbose) RDF/XML exchange syntax.

Namespace. Namespaces are inherited by OWL from XML. XML namespaces provide a method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by URI references. A standard initial component of an ontology includes a set of XML namespace declarations that provide a means to unambiguously interpret identifiers and make the rest of the ontology presentation much more readable.

Classes. A class defines a group of individuals sharing some commonalities. For instance, the functional-style syntax for declaring class `Window` is `Declaration( Class( :Window ) )`.

Subclasses. Class hierarchies may be created by specifying that a class is a subclass of another class. For instance, `SubClassOf( :Frame :Window )` expresses in functional-style syntax the fact that class `Frame` is a subclass of class `Window`. In OWL, each user-defined class is implicitly a subclass of class `Thing`.

Properties. Properties define relationships between individuals (*object* properties) or relate individuals to data values (*datatype* properties). Each property has a *domain* and a *range*. For instance, the following functional-style syntax defines the object property `size` with class `Window` as domain, and class `Dimension` as range.

```
Declaration( ObjectProperty( :size ) )
ObjectPropertyDomain( :size :Window )
ObjectPropertyRange( :size :Dimension )
```

Analogously, the datatype property `opacity` is declared as follows:

```
Declaration( DataProperty( :opacity ) )
DataPropertyDomain( :opacity :Window )
DataPropertyRange( :opacity xsd:float )
```

In OWL, classes and properties declared in the same namespace must have distinct names.

2.2 The Jena library

Jena (`http://www.openjena.org/`) is a Java framework for building Semantic Web applications. It is an open source project grown out of work with the HP Labs Semantic Web Programme and includes the OWL API used for implementing the ontology extractor described in Section 3.

The most useful Jena feature for our purpose is the ontology model created through the Jena `ModelFactory` class. The factory provides a method `createOntologyModel` to create an ontology model which implements the `OntModel` interface and contains ontology data expressed in any of the three OWL sublanguages.

```
OntModel myOntoModel =
   ModelFactory.createOntologyModel(spec);
```

The `spec` parameter of type `OntModelSpec` specifies the ontology model settings with respect to sublanguage, in-memory storage, and inference capabilities.

Namespaces are set using the `setNsPrefix(String prefix, String URI)` method that declares that the namespace URI may be abbreviated by prefix. Once serialized on file, the RDF/XML writer will turn these prefix declarations into XML namespace declarations and use them in its output.

All of the classes in the ontology API that represent ontology entities have `OntResource` as a common super-class: `OntClass` and `OntProperty` are two of them, with intuitive meaning.

An ontology model can be created and modified in many ways.

An ontology in RDF/XML exchange syntax can be read from a file or retrieved from the URL corresponding to the ontology URI, by using the `read` method:

```
myOntoModel.read(URI, "RDF/XML");
```

New resources can be added to the ontology model using the corresponding method (`createClass`, `createObjectProperty`, `createDatatypeProperty`):

```
OntClass myClass =
   myOntoModel.createClass();
```

New associations can be added to the subclass relation:

```
myClass.addSubClass(mySubClass);
```

corresponds to the declaration

```
SubClassOf( :mySubClass :myClass ).
```

Once an ontology model is completed, the `write` method can be used to serialize the model on a file; for instance,

```
myOntoModel.write(str, "RDF/XML")};
```

writes in RDF/XML exchange syntax the ontology corresponding to `myOntoModel` on the file associated with Stream `str`.

2.3 Ontology Matching

A *correspondence* [9] between an entity E_1 belonging to ontology O_1 and an entity E_2 belonging to ontology O_2 is a 5-tuple $< Id, E_1, E_2, Rel, Conf >$ where Id is a unique identifier of the correspondence; E_1 and E_2 are the entities (e.g. properties, classes, individuals) of O_1 and O_2 respectively; Rel is a relation such as "equivalence", "more general", "disjointness", "overlapping", holding between the entities E_1 and E_2 (in this paper we only consider the equivalence relation); $Conf$ is a confidence measure (typically in the $[0, 1]$ range) holding for the correspondence between the entities E_1 and E_2.

An *alignment* of ontologies O_1 and O_2 is a set of correspondences between entities of O_1 and O_2.

A *matching process* can be seen as a function F which takes two ontologies O_1 and O_2, a set of parameters P and a set of oracles and resources Res, and returns an alignment A between O_1 and O_2.

Often, the main activity of a matching method is to measure a pair-wise similarity between entities and to compute the best match between them. These methods exploit the definitions of similarity and of distance, and may be roughly classified into "name-based", "structure-based", "extensional" and "semantic-based" according to the kind of input they operate on.

Since we exploit name-based methods (string-based and language-based ones) in our experiments, we give a brief account of them only.

String-based methods. These methods measure the similarity of two entities just looking at the strings, seen as mere sequences of characters, that label them. They include:

Substring Distance: measures the ratio of the longest common substring of two strings with respect to their length.

***n*-gram Distance:** two strings are the more similar, the more n-grams (sequences of n characters) in common they have [3].

SMOA Measure: similarity of two strings is a function of their commonalities (in terms of substrings) as well as of their differences [17].

Language-based methods. These methods exploit natural language processing techniques to find the similarity between two strings seen as meaningful pieces of text rather than sequences of characters. Some of these methods exploit external resources, such as WordNet [13], that provide semantic relations such as synonymy, hyponymy, and hypernymy, to compute the similarity.

2.4 Javadoc and Doclet

Javadoc is a JDK tool that parses declarations and comments contained in a Java program or library, to generate HTML human readable documentation. The list of source files and package names that have to be processed must be passed as an argument to the tool; all subpackages of a given package are recursively processed when the `-subpackage` option is provided.

Javadoc has proved to be the ideal tool for our purposes, since it has been expressly designed to automatically extract software documentation from the source code. The tool is based on the `javac` parser, but no typechecking is performed, therefore documentation can be successfully generated also when all method bodies are empty or, in general, do not typecheck (even though they must be syntactically correct); in this way developers can generate software documentation also at early stages of design when code cannot be successfully compiled yet.

The notion of *doclet* allows easy customization of the tool to generate any kind of documentation. A doclet is a class that uses the standard doclet API defined in the package `com.sun.javadoc` to inspect the Java source code and generate the desired form of documentation. The default doclet is the standard predefined one generating the usual HTML documentation; however, it is possible to hook up Javadoc with a specific customized doclet with the option `-doclet`. More in details, each doclet must provide the public and static method **`boolean start(RootDoc root)`** which is automatically invoked by Javadoc. `RootDoc` is a subinterface of `Doc` which is at the root of the interface hierarchy of the API, and allows visiting the parse tree generated by `javadoc`.

3. ONTOLOGY EXTRACTION

In this section we present `Ontlet`, a doclet which allows extraction of lightweight ontologies from Java source code, and which has been explicitly designed to be easily extended.

The implementation of such a doclet is succinct, thanks to the intensive use of the Doclet API and of the Jena library (see Section 2).

3.1 Basic design decisions

Simplicity has been one of the main design choices driving the development of `Ontlet`: the simplest possible translation has been considered for each element of the language. While for some entities, like classes, the simplest translation is also the most suitable one, for other entities, like methods, there is not a most natural choice. However, keeping the extracted ontology as simple as possible has several advantages. Our choice has allowed us to quickly develop a doclet (less than 1K LOC), with which we have been able to start experimenting our approach on the whole Java language and on large existing libraries (see Section 4). Experiments are essential to validate our design decisions, or to point out weak points, and to show interesting directions for enhancements; the doclet has been designed to ease modifications and extensions, to support an incremental approach to the problem: try the simplest solutions first, consider more elaborated ones only if the former do not work properly. Finally, keeping the generated ontology simple has allowed us to start our experiments with the simplest available matching algorithms, and to avoid performance penalties that may occur when manipulating complex ontologies or using sophisticated matching algorithms.

Ontology boundaries and dependencies. An ontology always corresponds to a set of Java packages which must be explicitly specified by the user. `Ontlet` recognizes the standard Java API, and maps it to a set of predefined ontologies; for non standard imported packages, the user has to provide an explicit mapping between imported packages and external ontologies with all information needed by `Ontlet` to create the required namespaces.

Namespaces. In OWL, classes and properties in the same namespace must have distinct names, therefore a different namespace must be created for each Java package, subpackage and class. For instance, if class `C` is declared in package `pck.test`, then the following namespaces are generated: `pck.test` and `pck.test.C` corresponding to the directory `~/Java/pck/test` and the file `~/Java/pck/test/C.java`, respectively.

Classes and interfaces. Java classes (including exceptions and enums) and interfaces are translated uniformly in corresponding ontology classes. Except for local and anonymous classes, nested classes are translated as well into ontology classes contained in the namespace corresponding to their enclosing class. Static nested and inner classes are treated uniformly.

For instance, given the following class declaration

class C_1 **extends** C_2 **implements** $I_1, \ldots I_n$ { ... }

the generated ontology will contain class C_1, declared as a subclass of C_2, and of $I_1, \ldots I_n$.

Finally, the `deprecated` tag detected by Javadoc is translated by using the standard OWL notation for deprecated classes.

Fields. Each Java class field is translated into an object or datatype property; excluding the special case of arrays, fields of reference type correspond to object properties, whereas fields with primitive types correspond to datatype properties. The domain of the property is the enclosing class of the field, while the range corresponds to the type of the field. The translation does not discriminate between static fields and instance variables

If a field is an array, then the translation is more elaborated: in this case the range of the generated object property is always the ontology class corresponding to the class `java.lang.reflect.Array`; the type information that cannot be easily encoded in the ontology are kept in form of annotations of the property.

Finally, if a field is deprecated, the corresponding generated property contains the standard OWL notation for deprecated properties.

Constructors and methods. Methods are translated into object properties where the domain corresponds to the enclosing class, whereas the range is the ontology class corresponding to class `java.lang.reflect.Method`. The property belongs to the namespace generated from the enclosing class, and its name coincides with the name of the corresponding method. The translation has been kept deliberately simple: there is no distinction between abstract, static and instance methods, throws clauses are ignored, and overloaded methods declared in the same class are always mapped to a unique property. However, the return type and the signature of overloaded methods are kept in the ontology in form of annotations of the corresponding object property. Keeping such information in form of annotation leaves the opportunity to develop specific matching algorithms based also on type compatibility.

When only some overloaded version of a method are deprecated, a special annotation is generated, bound to the corresponding method signature. The property is annotated as deprecated only when all overloaded methods are deprecated.

Finally, for constructors only annotations are generated, associated with the corresponding enclosing class; they contain information on the signatures of all overloaded versions, and on deprecated constructors.

To better understand how the translation works, we consider the two classes given in Figure 1. The ontology entities

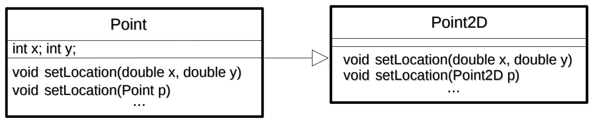
Point
int x; int y;
void setLocation(double x, double y)
void setLocation(Point p)
...
Point2D
void setLocation(double x, double y)
void setLocation(Point2D p)
...

Figure 1: Classes `Point2D` and `Point`

generated from the two classes are graphically described in Figure 2. Rectangles and ovals represent classes and properties, respectively. Domains and ranges of properties are represented by incoming and outgoing black arrows, respectively. White arrows represent subclass and subproperty relations. Light blue (or gray) corresponds to objects, dark orange (or gray) corresponds to datatypes.

Ignored Java features. The doclet ignores all kinds of class, method, and fields modifiers, as well as all information regarding generic classes and methods, and parameterized types. Furthermore, there is a handful of other information extracted by Javadoc that are disregarded by the doclet.

3.2 Design of `Ontlet`

To make the doclet easily extensible, we have implemented the bridge design pattern [10] for decoupling the ontology extraction process from the other relevant activities. Figure 3 describes the three classes which constitute the bulk of the application. `Ontlet` is the main class implementing the doclet; it provides the static method `start` which is automatically invoked by Javadoc. Such a method inspects the input source code through the parameter `root`: all classes of the specified packages are visited, and for each of them

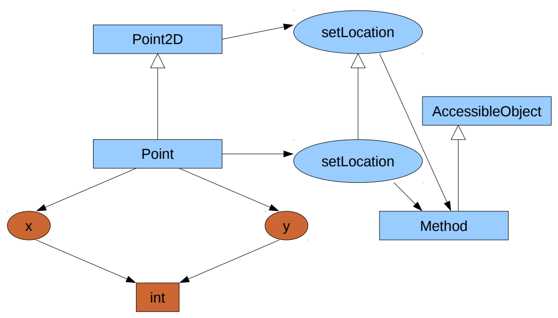

Figure 2: Translation of `Point2D` and `Point`

all declared fields, constructors, and methods are considered. All nested classes are processed as well, accordingly to the behavior of method `allClasses` of `RootDoc`. `Ontlet`

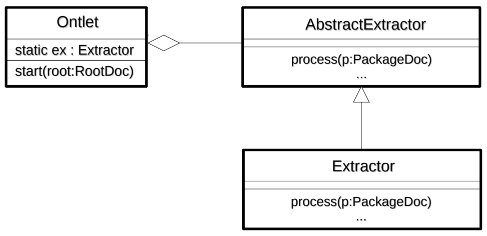

Figure 3: Main classes of the Ontlet

delegates to class `Extractor` the generation of all ontology entities, through the object stored in the static field `ex` of type `AbstractExtractor`. The visited program elements are those filtered by the access control Javadoc options (`-public`, `-protected`, `-package`, and `-private`). `Ontlet` is also responsible for processing all Javadoc options, and properly initializing the static variable `ex` with an instance of class `Extractor`; during such an initialization all options processed by the doclet are passed to the object implementing the ontology extractor. In case class `Extractor` is extended, proper initialization of field `ex` with an instance of the subclass of class `Extractor` can be managed by redefining method `start`, possibly in a subclass of `Ontlet`.

AbstractExtractor class. This abstract class defines all basic features of an extractor. It contains abstract overloaded methods for processing all main Java entities:

```
abstract void process(PackageDoc pd);
abstract void process(ClassDoc cd);
abstract void process(ConstructorDoc cd);
abstract void process(FieldDoc fd);
abstract void process(MethodDoc md);
```

The method for processing classes assumes that the passed class belongs to the current package, that is, the last processed package; similarly, methods for processing constructors, fields, and methods assume that the passed arguments belong to the current class, that is, the last processed class. Finally, `AbstractExtractor` is responsible for creating and initializing the proper Jena ontology model, for storing all options processed by `Ontlet`, and for serializing the generated ontology into a file specified in the options. Several languages can be selected when writing an ontology.

Extractor class. This class implements the informal specification given in Section 3.1, by overriding all abstract methods inherited from `AbstractExtractor`. More sophisticated extraction processes can be obtained by extending such a base class.

4. EXPERIMENTS

In order to test the feasibility of our approach we have extracted six ontologies from three pairs of Java libraries, each consisting of an older and a newer version of the same library, and run on each pair four freely available implementations of ontology matching algorithms.

To be able to correctly evaluate an alignment computed by an ontology matcher, one has to provide first a *reference alignment* that contains only all correct associations, and that is typically built manually. This is a time consuming task consisting in thoroughly analyzing the API documentation to discover how deprecated entities should be correctly replaced. Indeed, one of the main motivations of our work is just to allow users to save this time (at least partially), by using a tool able to generate all correct associations.

Since two versions of the same library are likely to have a very similar structure, these experiments allow us to establish whether in these cases it is possible to obtain good results also with those simpler ontology matchers that disregard the structure of ontologies.

Of course, it would be also very interesting to perform experiments to compare different libraries, rather than different versions of the same library. However, in this case building a reference alignment is a much more complex task, and we expect that more sophisticated matching algorithms are required to get significant results.

The libraries we selected are very different in number of declarations, ranging from `java.awt` with hundreds of classes, and thousands of fields and methods, to `java.net` with less then one hundred classes and fields, and around 500 methods. The percentage of deprecated entities varies as well: `java.awt` and `java.net` do not have deprecated classes, but have a significant percentage of deprecated methods (reaching its maximum with `java.awt`) and less deprecated fields, whereas `java.security` has more deprecated classes than methods and fields.

On each pair of extracted ontologies we have run three string-based ontology matching algorithms, SMOA, substring and n-grams, and one language-based algorithm that uses WordNet to exploit the semantics of concepts appearing in the ontology.

For each experiment we have forced replacement of deprecated entities with non deprecated ones by simply removing all deprecated entities from the target ontologies.

4.1 Followed methodology

We used the substring, n-gram, SMOA, and WordNet-based ontology matchers offered by the Alignment API[1] 4.1.

[1]`http://alignapi.gforge.inria.fr/`

The accuracy of the algorithms have been estimated by comparing the generated alignments with the reference ones in terms of the standard notions of *precision*, *recall* and *F-measure* [7].

To compute precision and recall, the alignment a returned by the algorithm is compared to a reference alignment r. Precision is given by the formula $P(a, r) = \mid r \cap a \mid / \mid a \mid$ and recall is defined as $R(a, r) = \mid r \cap a \mid / \mid r \mid$. A perfect precision score of 1.0 means that every correspondence computed by the algorithm was correct (correctness), whereas a perfect recall score of 1.0 means that all correct correspondences were found (completeness).

F-measure is the harmonic mean of precision and recall: $F = 2 \cdot P(a, r) \cdot R(a, r)/(P(a, r) + R(a, r))$.

4.2 Results and evaluation

Table 1 shows the obtained results in term of precision, recall and F-measure. Figures for all four ontology matchers are quite similar, from which we could deduce that none of the algorithms performs significantly better than the others. The outcomes of the experiments seem also to suggest that results are independent of the dimension of the ontology, and of the percentage of deprecated entities.

If we consider the fact that we have used the simplest available ontology matchers, and that the developed doclet is less than 1K LOC, results are promising, but also reveal the limitations of using matchers that completely disregard the structure of ontologies. The use of WordNet deserves some more comments. Using sophisticated language-based matching methods should be considered with care, since there is no evidence that the their complexity brings significant improvements. Among the methods offered by the Alignment API, the one based on WordNet gives results comparable to those of the other methods if we only look at the F-measure. In all the experiments, its recall is greater or equal than the one of the other methods, and its precision is lower. This means that, at least for the ontologies we have considered, the more sophisticated methods retrieve more correct correspondences than the others, but also generate more noise.

A possible explanation of the fact that the matcher based on WordNet does not perform better than the others, is that names of entities in Java code often contain acronyms and abbreviations that are meaningless if a general purpose vocabulary is used (it is not by coincidence that package `java.security` is full of acronyms, and give the worst results). It would be interesting to repeat these experiments after having extended WordNet with typical words from the Computer Science and Object-Oriented Programming jargon.

5. RELATED WORK

There exist other proposals for extracting ontologies from software artifacts, but all those we are aware of are mainly focused on reverse software engineering [22, 23, 16, 2, 24, 25, 12, 15, 16, 21].

Comparison with related work is considered w.r.t. to different important aspects: aims and reuse.

Aims. We implemented a *flexible and customizable framework for extracting OWL ontologies from Java libraries.* Concerns about the library features that the extracted OWL ontology will represent and about the final application where it will be employed are left to the user of our framework. By extending the basic doclet, users can easily implement their own ontology extractor, and decide which details of the library should be documented. All the proposal reviewed in this section are instead driven by very specific goals and application constraints, and hard-wire the ontology extraction method in the implemented or suggested algorithm.

Reuse. Coherently with our aim, we developed a framework easy to maintain and to evolve thanks to the *reuse of widely adopted open-source Java libraries and tools* (Javadoc, Doclet, and Jena).

Names of the ontology concepts and properties that we generate are the class and method names *as they appear in the source code*, whereas in most of the related papers mentioned above, they are meaningful words derived by means of a natural language processing stage from the names of source code entities. Our choice is intentional, is driven by our will of "not reinventing the wheel", and can be explained by comparing the proposal by Ratiu, Feilkas, and Jürjens to ours. In that paper, the authors take many APIs as input and generate one ontology whose concepts match the main concepts available in all the APIs. Ratiu et al. 1) extract one graph from each API, where the labels of graph nodes are the strings that appear in the API and 2) implement from scratch an ad-hoc graph matching algorithm that takes names and graph structure into account, for generating a single ontology from many graphs.

Let us suppose that a programmer wants to use our framework for reaching the same goal. She should 1) use our framework for extracting one ontology from each API, where the labels of the ontology elements are just the strings that appear in the API, and 2) use one of the dozens available ontology matching algorithms (`http://www.ontologymatching.org/`) that exploit both name-based and structure-based techniques for generating the resulting merged ontology.

Given this bunch of choice, we think that extracting a raw ontology from the API and deferring the ontology matching to a successive stage is a winning choice in order to take advantage of the more and more valuable results that the ontology matching community is producing.

Similar considerations hold in case the extracted ontology should be used for other purposes: ontology merging and fusion methods [8, 14] could be used to help creating a unified library from different existing ones whereas multilingual ontology mapping [18] could be exploited to suggest correspondences between classes and methods labeled using different languages. Due to the liveliness of the research in the ontology field, it is likely to find some already implemented software that meets the user's requirements.

6. CONCLUSION

We have defined and implemented a simple and extensible doclet for generating semantic information in the form of ontologies from Java libraries, with the main aim of supporting semi-automatic migration of code due to upgrade or replacement of libraries. We have experimented the approach by extracting ontologies from different versions of four significant packages of the standard Java API, and by running four simple freely available ontology matchers on the generated ontologies, to evaluate the accuracy of the computed alignments. The results are promising, if we consider that they have been obtained from real and large examples by writing less than 1K LOC, and by running very simple matchers; however, they have also revealed the limitations of using matchers that completely disregard the structure of

		Matching algorithm			
Package	**Indicator**	**Substr**	**N-gram**	**SMOA**	**WordNet**
java.awt	Precision	0.61	0.63	0.59	0.60
	Recall	0.54	0.51	0.52	0.54
	F-measure	0.57	0.56	0.55	0.57
java.net	Precision	0.66	0.70	0.67	0.66
	Recall	0.60	0.59	0.60	0.60
	F-measure	0.63	0.64	0.63	0.63
java.security	Precision	0.47	0.54	0.47	0.47
	Recall	0.44	0.42	0.44	0.44
	F-measure	0.46	0.47	0.45	0.45

Table 1: Alignment results

ontologies.

More experiments are needed to better evaluate and enhance our approach; we envisage two different directions. On one hand, it is important to perform experiments to compare also different libraries, and not just different versions of the same library.

On the other hand, it would be interesting to evaluate the result one can obtain by running more sophisticated ontology matcher that take into account also the structure of the generated ontologies.

7. REFERENCES

[1] K. Arnold and J. Gosling. *The Java^{TM} Programming Language, Third Edition*. Addison-Wesley, 2000.

[2] K. Bontcheva and M. Sabou. Learning ontologies from software artifacts: Exploring and combining multiple sources. In *SWESE'06*, 2006.

[3] E. Brill, S. Dumais, and M. Banko. An analysis of the askmsr question-answering system. In *EMNLP 2002*, 2002.

[4] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *WWW (Alternate Track Papers & Posters)*, pages 74–83, 2004.

[5] R. D. Cosmo, F. Pottier, and D. Rémy. Subtyping recursive types modulo associative commutative products. In *TLCA 2005*, pages 179–193, 2005.

[6] J. David, J. Euzenat, F. Scharffe, and C. T. dos Santos. The Alignment API 4.0. *Semantic Web Journal*, 2, 2011. To appear.

[7] H. H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. In *NODe 2002*, pages 221–237, 2002.

[8] D. Dou, D. Mcdermott, and P. Qi. Ontology translation by ontology merging and automated reasoning. In *EKAW'02*, pages 3–18, 2002.

[9] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, 2007.

[10] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[11] T. Gruber. Ontology. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 1963–1965. Springer US, 2009.

[12] M. Hepp and A. Wechselberger. OntonaviERP: Ontology-supported navigation in ERP software documentation. In *ISWC 2008*, pages 764–776, 2008.

[13] G. Miller. WordNet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[14] N. F. Noy and M. A. Musen. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *AAAI/IAAI 2000*, pages 450–455, 2000.

[15] D. Ratiu, M. Feilkas, and J. Jürjens. Extracting domain ontologies from domain specific APIs. In *CSMR 2008*, pages 203–212, 2008.

[16] M. Sabou. From software APIs to web service ontologies: A semi-automatic extraction method. In *ISWC 2004*, pages 410–424, 2004.

[17] G. Stoilos, G. B. Stamou, and S. D. Kollias. A string metric for ontology alignment. In *ISWC 2005*, pages 624–637, 2005.

[18] C. Trojahn, P. Quaresma, and R. Vieira. A framework for multilingual ontology mapping. In *LREC'08*, 2008.

[19] W3C. OWL Web Ontology Language Overview – W3C Recommendation 10 February 2004, 2004.

[20] W3C. OWL 2 Web Ontology Language Document Overview – W3C Recommendation 27 October 2009, 2009.

[21] H. H. Wang, N. Gibbins, T. Payne, A. Saleh, and Y. Li. Transitioning applications to semantic web services: An automated formal approach. *International Journal of Interoperability in Business Information Systems, IBIS*, 2008.

[22] C. A. Welty. Augmenting abstract syntax trees for program understanding. In *ASE'97*, pages 126–133, 1997.

[23] H. Yang, Z. Cui, and P. O'Brien. Extracting ontologies from legacy systems for understanding and re-engineering. In *COMPSAC'99*, pages 21–26, 1999.

[24] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev. Ontology-based program comprehension tool supporting website architectural evolution. In *WSE'06*, pages 41–49, 2006.

[25] H. Zhou, J. Kang, F. Chen, and H. Yang. OPTIMA: An Ontology-based PlaTform-specIfic software Migration Approach. In *QSIC'07*, pages 143–152, 2007.