Automatic ontology extraction from Java libraries for machine-readable API documentation

Davide Ancona   Viviana Mascardi   Ombretta Pavarino
DISI - University of Genova, Italy
{davide,mascardi}@disi.unige.it,ombretta.pavarino@virgilio.it

Abstract

While almost all programming languages are equipped with suitable tools for extracting human-readable documentation from software, little effort has been devoted to generating machine-readable information which could be fruitfully exploited by applications for enhancing software development.

This paper is focused on ontology-based documentation, a topic which opens up new interesting scenarios in the fields of code refactoring, software migration, and reverse engineering. More in details, we propose a Java framework for extracting OWL ontologies from Java libraries, based on Javadoc technology for automatic documentation extraction, and on Jena, a Java framework for building Semantic Web applications. In this way, programmers can easily implement their own ontology extractor, and decide which features of the library should be documented by the generated ontology, by extending the basic Semlet (a Doclet able to generate semantic documentation) provided by our framework.

We have experimented the framework by defining a basic ontology generator which allowed us to extract a valid OWL Lite ontology from all `java.*` packages of the standard class library.

Keywords Ontology, machine-readable API documentation, Javadoc

1. Introduction

While almost all programming languages provide suitable tools for extracting human-readable documentation from software, little effort has been devoted to generating machine-readable information which could be fruitfully exploited by applications for enhancing software development.

Most of the works (see Section 5 for a survey on related papers) which use ontologies for representing relevant semantic information extracted from software are mainly focused on Reverse Software Engineering, that is, they aim at generating models which help the programmers to understand the software and, therefore, mainly cover disciplines as software requirements and design.

However, ontology-based documentation opens up new interesting scenarios also for software development and maintenance: in particular, tools for semi-automatic porting and migration of libraries, and also 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 and maintenance process.

Let us consider, for instance, the issue of modifying a program p after that a library l has been replaced with a new library l' which is not completely compatible with the previous one. This scenario may occur either when l is replaced with a library l' independently developed by a third party not concerned with compatibility w.r.t. l (software porting), or when l' is just a major upgrade of l (software migration). Take for instance the JDK libraries; even though full compatibility with older versions should be guaranteed by deprecated components, if one really wants to take advantage of the features offered by the new versions, program p needs necessarily to be modified and adapted to correctly use the new offered interfaces.

If one can easily extract the relevant semantic information out of l and l' in form of two corresponding ontologies o and o', then the needed modification on p could be generated in a semi-automatic way by aligning o and o' with a suitable ontology matching algorithm mapping the elements of o to the corresponding ones in o'.

Our work is motivated by the fact that the few proposals which use ontology extraction for software migration and porting rely on ad hoc ontology extraction and ontology matching algorithms which are hardwired in the application; as a consequence, since these approaches are not modular, they cannot be easily adapted to other kinds of ontology extraction and ontology matching algorithms.

To overcome this problem, we have defined a framework, called Semlet, for rapid development of tools for automatic ontology extraction from Java libraries. The framework is quite simple, since it is completely based on Javadoc (the standard Java tool for extracting documentation from programs) and Jena, a widespread Java library for developing semantic web applications based on RDF and OWL, and processing ontology formalisms built on top of RDF. Thanks to Javadoc, Jena, and the design of Semlet, users can rapidly develop different ontology extraction algorithms, to experiment with different ontology matching algorithms.

We have successfully experimented Semlet by developing a plain ontology extractor in less than 400 lines of code; such an extractor was able to quickly generate (less than one minute on a dual-core laptop) the ontology corresponding to all standard `java.*` packages, containing around 10K different entities.

The paper is structured as follows. Section 2 provides the necessary background on the ontology language OWL, Jena, and Javadoc. Section 3 is focused on the main design aspects of Semlet, whereas Section 4 presents the plain extractor developed with our framework. Finally, Section 5 is devoted to the related work and the conclusion.

2. Background

According to T. Gruber [9],

In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application.

In this section we introduce the OWL ontology language (Section 2.1) and the Jena framework (Section 2.2) that we used for modeling the ontology extracted from the source code of a Java library. Identification of relevant components from the Java library given in input to our framework is performed via the Javadoc tool presented in Section 2.3.

2.1 The Web Ontology Language (OWL) family

Among the languages for representing ontologies, OWL [19] is one of the most widespread ones. OWL stands at the topmost layer of the standardized technologies in the Semantic Web Stack (see Figure 1). The Semantic Web Stack was introduced for the first time by T. Berners-Lee, Director of the World Wide Web Consortium (W3C), in a keynote session at XML 2000 [3]. It represents the hierarchy of Semantic Web languages, where each layer grounds on the layers below.

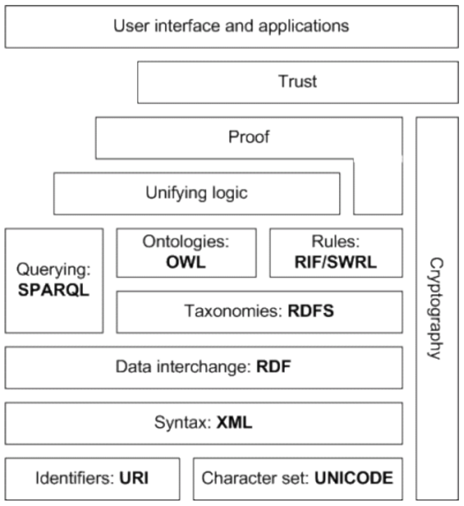

Figure 1. The Semantic Web Stack [3].

The bottom layers contain hypertext web technologies: Internationalized Resource Identifier (IRI) is a generalization of URI and provides means for uniquely identifying semantic web resources; Unicode serves to represent and manipulate text in many languages; XML is a markup language that enables creation of documents composed of structured data [17]; XML Namespaces provide a way to use markups from more sources [18].

Middle layers contain technologies standardized by W3C to enable building semantic web applications.

- Resource Description Framework (RDF, [21]) allows to create statements about resources in form of triples.
- RDF Schema (RDFS, [20]) provides basic vocabulary for RDF and allows, for example, to create hierarchies of classes and properties.
- Web Ontology Language (OWL) extends RDFS by adding more advanced constructs to describe semantics of RDF statements; since it is based on description logic [2], it empowers the semantic web with reasoning capabilities.

  OWL provides three increasingly expressive sub-languages: OWL Lite, OWL DL, and OWL Full.

  - OWL Lite supports those users primarily needing a classification hierarchy and simple constraints. It has a lower formal complexity than OWL DL.
  - OWL DL gives the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, a class cannot be an instance of another class). The "OWL DL" name is due to its correspondence with Description Logics.
  - OWL Full gives the maximum expressiveness and the syntactic freedom of RDF with no computational guarantees.

  Each of these sub-languages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded.

  In October 2009 the specification of OWL 2 was released [23]. OWL 2 is a conservative extension of OWL 1: all OWL 1 Ontologies remain valid OWL 2 Ontologies, with identical inferences in all practical cases. In our work we only consider OWL 1 ontologies because this is the language supported, at the time of writing, by the Jena library (see Section 2.2).
- SPARQL [22] is a RDF query language and can be used to query any RDF-based data, including statements involving RDFS and OWL.

Top layers contain technologies that are not yet standardized or contain just ideas what should be implemented in order to realize Semantic Web.

Since giving an account of the complete OWL language is out of the scope of this paper, we limit ourselves to summarize the few aspects that are really necessary to understand how our framework works.

Namespace. Namespaces are inherited by OWL from XML. XML namespaces provide a simple 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.

Class. A class defines a group of individuals that belong together because they share some common properties. The OWL class element, identified by owl:Class, is a subclass of the RDFS class element, rdfs:Class. The rationale for having a separate OWL class construct lies in the restrictions on OWL DL (and thus also on OWL Lite), which imply that not all RDFS classes are legal OWL DL classes.

Subclass. Class hierarchies may be created by making one or more statements that a class is a subclass of another class. This can be achieved by using the rdfs:subClassOf element defined by RDFS. In OWL, each user-defined class is implicitly a subclass of owl:Thing.

Property. Properties have originally being defined in RDF and can be used to state relationships between individuals (object properties, owl:ObjectProperty) or from individuals to data values (data type properties, owl:DatatypeProperty). Both object and data type OWL properties are subclasses of the RDF class rdf:Property. In OWL properties are first-class objects that exist independently from the existence of user-defined classes. Relationships between property and classes depend on the property domain and range that may be (must be, as far as the domain is concerned) classes. However, a property can be defined specifying neither its domain nor its range, which are then set to the default class Thing.

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 Extractor described in Section 3.

One of the most useful feature that Jena provides for our purpose is the ontology model created through the Jena ModelFactory class. The createOntologyModel method that the factory provides can be used to create an ontology model which implements the OntModel interface and contains ontology data expressed in a given ontology language:

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

The spec parameter of type OntModelSpec specifies the ontology model settings with respect to language, in-memory storage, 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 filled with resources and relationships among them in many ways:

- the ontology can be read from a file or retrieved from a URL representing the ontology URI using the read method :

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

- new resources can be added to the ontology model using createClass, createDatatypeProperty, createObjectProperty :

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

- new relationships can be created; for example:

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

  states that mySubClass is in rdfs:subClassOf relation with myClass.

Once the ontology definition is completed, the write method with its parameters can be used to serialize the ontology model on file (for example, myOntoModel.write(str, "RDF/XML") for writing all the ontology data contained in myOntoModel on the file associated with Stream str, in the RDF/XML format).

2.3 Javadoc and Doclet

Javadoc. Javadoc is a Java tool that parses declaration and documentation comments in a set of Java source files and packages and produces, by default, a corresponding set of HTML pages describing the classes, inner classes (but not anonymous inner classes), interfaces, constructors, methods and fields.

The input to Javadoc may be a list of .java files or package names separated by space, or the −subpackage option followed by package names that causes the recursive parsing of all the subpackages of the given ones.

Javadoc produces one complete document each time it is run; it cannot be used incrementally. It requires and relies on the Java compiler to do its job. The Javadoc tool calls part of javac to compile the declarations, ignoring the member implementation.

Javadoc can run on .java source file that are pure stub files with no method bodies. This allows the programmer to produce documentation of software at the early stage of design, and also for source files whose code contains errors and that could not be compiled yet.

A recent project named *Classdoc* (`http://classdoc.sourceforge.net/`) aims at providing an alternative to Javadoc by working on .**class** files instead of on .java ones. Classdoc has some limitations discussed in `http://classdoc.sourceforge.net/classdoc10/classdoc.html#limitations`. The most severe one is that it only works on bytecode of classes developed with JDK up to 1.3.

Doclet. In order to generate its output, Javadoc calls a doclet, namely a Java program which uses the Doclet API to specify the format of the Javadoc output. By default, the Standard Doclet, that generates a three-part browsable HTML documentation file for each processed class, is used.

The programmer can define a doclet that extends the Standard one, if she wants to generate an HTML documentation with a different style. However, if she wants to generate a documentation completely different form the HTML (standard or ad-hoc) one, she can define her own brand new doclet by extending Doclet and implementing the start method with a RootDoc parameter.

The Doclet API offers methods for navigating through classes, their fields and their methods, and to collect any information about them. Inheritance allows the programmer either to extend the behavior of the Standard Doclet or to redefine it in order to generate an output that documents the input classes and packages in whatever user-defined format.

Classdoc implements the Java Doclet API so that any Doclet can be used to generate output from bytecode.

3. An abstract doclet for ontology extraction

In this section we present Semlet, a customizable doclet for ontology extraction. Semlet has been designed with the main intent of providing a framework for generating ontologies from Java libraries, allowing users to easily adapt the ontology extractor provided by Semlet to their specific needs.

The framework is based on the Doclet API com.sun.javadoc which provides an easy mechanism for extracting out of a Java program all those syntactic information useful for code documentation, and on the Jena API library com.hp.hpl.jena which is a framework for developing semantic web applications based on RDF and OWL. In particular the API com.hp.hpl.jena.ontology is a toolkit for processing ontology formalisms built on top of RDF (specifically RDFS, OWL Lite and DL).

3.1 Basic assumptions

To simplify the design of the framework, Semlet has been conceived for implementing only a specific subset of ontology extraction algorithms, complying with a number of reasonable assumptions that we have identified during the development of the project. However, we strove for ensuring a right balance between simplicity and flexibility, and tried to exclude only the most esoteric ontology extractors.

Ontology granularity. An ontology always corresponds to a set of Java packages which must be explicitly specified by the user. For what concerns the imported packages, Semlet 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 Semlet to create the required namespaces.

A class is always a class. One of the main principles which has driven our design is that a Java class[1] must always correspond to an ontology class; furthermore, there must be a bijection between the classes of a program and the classes of the corresponding ontology. As a direct consequence of this principle, the inheritance relation must coincide with the subtyping relation between ontology classes. For instance, the ontology class extracted from the following class declaration

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

must be a subclass of the ontology classes corresponding to C_2 and $I_1, \dots I_n$.

Namespaces. In Java both packages and classes define namespaces, therefore two classes with the same name, but declared in different packages, must correspond to different ontology classes. The same reasoning applies to nested classes as well: two nested classes with the same name, but declared in different classes must correspond to different ontology classes. For this reason, Semlet always preserves the structure of Java namespaces in the extracted ontology. 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.

```
<rdf:RDF
  ...
  xmlns:pck.test="file:~/Java/pck.test#"
  xmlns:pck.test.C="file:~/Java/pck/test/C.java#"
  ...
</rdf:RDF>
```

Recognized Java constructs. Since Semlet is based on the Javadoc technology, ontology extraction may only depend on the parts of the program which are parsed by Javadoc, which, roughly correspond to all the program except the implementation (method bodies and initializers). This means that for instance, it would not possible to extract an ontology based on additional information retrieved with any kind of program analysis on the implementation. To do that, the standard behavior of Javadoc must be changed by redefining its core classes.

3.2 Design of the framework

The framework exploits the bridge design pattern [8] for decoupling inspection of Java code, performed by Semlet, from the construction of the ontology, which is demanded to an Extractor (Figure 2). Method start (RootDoc root) in Semlet contains invocations to the methods exposed by the Extractor interface with target object ex. Semlet is responsible for the traversal of the input library (represented by root), whereas ex is a class variable of type Extractor, containing the object with interacts with Jena and generates the entities of the ontology. Note that ex needs to be static, since method start executed by Javadoc is required to be static, hence can only access the class members of Semlet. To allow more flexibility, Semlet communicates with the extractor by passing the pertinent Doc objects which have to be processed. For instance, the method for processing classes has a parameter of type ClassDoc; in this way the extractor can easily retrieve all information needed for generating the required ontology entities. Hence, the Jena API is not visible to Semlet, whereas the Doclet API is used both by Semlet and by the extractor classes.

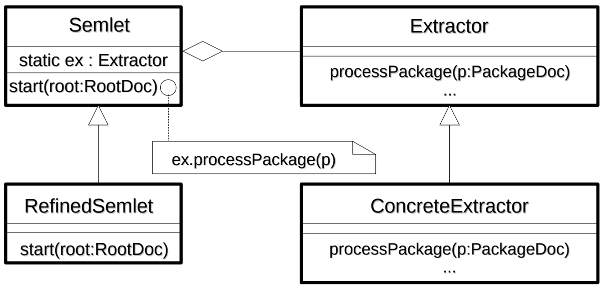

Figure 2. Design of the framework

The traversal performed by Semlet corresponds to the following simplified[2] Java code.

```
public class Semlet {
 ...
 protected static Extractor ex;

 public static boolean start(RootDoc root) {
  ...
  for(PackageDoc pd:root.specifiedPackages()) {
   ex.processPackage(pd);
   for(ClassDoc cd:pd.allClasses()) {
    if(cd.isInterface())
     ex.processInterface(cd);
    else if(cd.isError())
     ex.processError(cd);
    else if(cd.isException())
     ex.processException(cd);
    else ex.processClass(cd);
    visitClassDoc(cd);
   }
  }
  ...
 }
 private static void visitClassDoc(ClassDoc cld) {
  for(FieldDoc fd:cld.fields())
   ex.processField(fd);
  for(ConstructorDoc cnd:cld.constructors())
   ex.processConstr(cnd);
  for(MethodDoc md:cld.methods())
   ex.processMethod(md);
 }
 ...
}
```

Method start processes all packages specified by the user as arguments; for each package all included classes (those which are not filtered by the access control Javadoc options −public, −protected, −package, and −private) are processed, by taking into account the different kinds of classes: interfaces, errors, exceptions and ordinary classes (including enums). In this way the implementor of

[1] Unless specified, with class we mean also interfaces and enum types.

[2] We have omitted options, exceptions handling, and initialization of ex.

Extractor can easily decide which kinds of class must be included in the generated ontology and how; for instance, if one wants to exclude exceptions and errors, then it suffices to provide an empty body for methods processException and processError. Finally, all constructors, fields and methods of each included class in the package are processed, with a corresponding method of Extractor interface. Note that all nested classes are processed together with top-level classes, accordingly to the behavior of method allClasses. If for some reasons nested classes have to be visited after top-level ones, the traversal of root defined in Semlet has to be redefined by overriding method start in a subclass of Semlet.

As shown in the code of Semlet, interface Extractor exposes eight different methods for processing the main elements of a Java program. For avoiding proliferation of methods in Extractor, the specialization of the behavior of such methods is left as a responsibility of the implementor of Extractor. For instance, if one wants to deal differently with class and instance methods, then the check md.isStatic() must be inserted in the body of processMethod, and the corresponding behavior has to be factored out in two new methods. As already explained in Section 2.3, the Doclet API provides a rich set of methods for checking all properties of a program element; this gives a great amount of flexibility to the programmer who can choose the desired level of implementation refinement of the methods offered by Extractor.

3.3 Implementing a concrete extractor

As already explained, the class variable ex of Semlet is essential for implementing the bridge design pattern as depicted in Figure 2. However, ex is not initialized by Semlet, therefore running Javadoc with Semlet will simply produce a NullPointerException.

To allow Javadoc generating an ontology from a Java program, class Semlet needs to be subclassed and an implementation of interface Extractor has to be provided. Subclassing of Semlet is primarily required for correctly initializing class variable ex with an instance of an extractor class, that is, an implementation of Extractor. Note that simply adding a static initializer in the subclass of Semlet for defining ex does not work.

```
public class RefinedSemlet extends Semlet {
 static { ex=new ConcreteExtractor(); }
}
```

Indeed, since the method start invoked by Javadoc is not declared in RefinedSemlet, but simply inherited by Semlet, class RefinedSemlet will not be initialized (see rules 12.4.1 of the Java Language Specification [1]), and therefore the execution will throw a NullPointerException. Therefore, the only practical solution is overriding method start. Furthermore, initializing ex in method start rather than in a static initializer allows more flexibility and the implementation of the correct initialization sequence, where ex is first initialized by Semlet, and then by its subclass. Here is a simplified version of the pattern for subclasses of Semlet.

```
public class RefinedSemlet extends Semlet {

 public static boolean start(RootDoc root) {
  Conf conf=createConf(root.options());
  ex=new ConcreteExtractor(conf);
  return Semlet.start(root);
 }
}
```

An object of type Conf is first created by invoking the corresponding method createConf inherited by Semlet; such an object contains all option details needed for correctly creating an extractor (for instance, the string of the base URI of the extracted ontology). After initialization of ex, method start can execute the corresponding overridden method of class Semlet, if one would like to reuse the traversal of root as implemented by Semlet; otherwise, the statement **return** Semlet.start(root); needs to be replaced with the appropriate code for implementing a different traversal.

If RefinedSemlet introduces new options, then the pattern shown above has to be refined. A subclass SubConf of Conf has to be declared, to correctly deal with the new options; such a class will contain a constructor SubConf(Conf conf,...) where the ellipsis are the arguments corresponding to the new options.

```
public SubConf(Conf conf,...) {
super(conf); // inherited options
... // added options
}
```

Consequently, we obtain the following pattern for class RefinedSemlet.

```
 public static boolean start(RootDoc root) {
  SubConf conf=createConf(root);
  ex=new ConcreteExtractor(conf);
  return Semlet.start(root);
 }

 protected static SubConf
                  createConf(String[][] opts) {
  ... // handling of added options
  return new SubConf(conf,...);
 }
}
```

4. PlainExtractor: an example of concrete extractor

In this section we present a simple extractor, called PlainExtractor, developed with our framework.

We used PlainExtractor to generate an ontology describing a subset of the classes in the core Java API 1.6, namely those contained in the java.∗ packages and subpackages. The generated ontology is available at the URL `http://www.disi.unige.it/person/AnconaD/JavaDocumentationOntology/java`. This URL was also used as the URI identifying the generated ontology namespace. In the remaining of this section we will refer to this ontology as "the Java ontology" and to its namespace as javaNS.

Part of the Java ontology visualized by opening it with the Protégé 4.1 editor (`http://protege.stanford.edu/`) and using the "OWL Viz" visualizer is shown in Figure 3. The ontology contains 1413 classes, 8331 object properties and 1516 datatype properties according to the Protégé "OWL Model metrics" function.

Generating the Java ontology and serializing it on a file required approximately 40 seconds on a Linux 2.6.31 based system with Intel double core CPU 1.73GHz and 1GB of memory.

Besides defining all the relevant entities from the core Java API 1.6, the Java ontology also defines two special classes, Action and Array, whose purpose is explained in Section 4.1.

4.1 PlainExtractor main features

In the sequel we discuss the basic features of our PlainExtractor by means of a simple example of Java library dealing with bikes.

Access level modifiers. PlainExtractor considers access level modifiers only with respect to the selection of the elements to be processed by Javadoc. In our implementation we do not consider the modifiers in any other way and we just ignore them during the ontology construction. By setting the −**public**, −**protected** or −**private** Javadoc tool command line parameters we assert which elements participate in the construction of the resulting ontology, namely those whose visibility is greater than the one stated by the command line.

Classes and interfaces. A Java class c generates a corresponding OWL class c; interfaces are treated as classes (we recall that in OWL a class can be subclass of more classes). If c extends or implements $c_1 \ldots, c_n$, then the OWL class c is a subclass of the OWL classes $c_1 \ldots, c_n$. Generic classes are treated as simple classes, that is, their parameters are not considered. Tables 1 and 2 show two simple examples of class extraction. In the examples, bikeNS and mntBikeNS represent the namespace automatically assigned by the PlainExtractor to the Bike and MntBike classes respectively, as explained in Section 3.

public class Bike	`<owl:Class rdf:about=`
	`    "bikeNS:Bike">`
	`<rdfs:subClassOf`
	`    rdf:resource=`
	`    "java.lang:Object"/>`
	`</owl:Class>`

Table 1. Java class c that extends no class.

public class
MntBike
extends Bike
	`<owl:Class rdf:about=`
	`    "mntBikeNS:MntBike">`
	`  <rdfs:subClassOf`
	`    rdf:resource=`
	`    "bikeNS:Bike"/>`
	`</owl:Class>`

Table 2. Java class sc that extends class c.

Methods. All methods correspond to an OWL owl:ObjectProperty. This plain extractor does not consider method parameters and return type, but consider a method declared in a class c as a property (that is, a relation) whose domain is c, and whose range is the special class named Action. Overriding methods do not generate a new property, whereas abstract, class and instance methods are all treated in the same way; indeed, modifiers do not affect the translation.

Table 3 shows a simple example of translation of a method.

public BikeFeatr getFeatr()
`    {`
`        ...`
`    }`

`<owl:ObjectProperty rdf:about="bikeNS:getFeatr">`
`    <rdfs:domain rdf:resource="bikeNS:Bike" />`
`    <rdfs:range rdf:resource="javaNS:Action"/>`
`</owl:ObjectProperty>`

Table 3. Methods.

Fields. All fields are translated into either DatatypeProperty or ObjectProperty, depending on their types: a field whose type is primitive corresponds to a DatatypeProperty, otherwise is translated into an ObjectProperty. Class and instance fields are treated in the same way: modifiers do not affect the translation.

A field f of class c whose type is a basic type t with a corresponding data type in XML corresponds to an OWL datatype property whose identifier is f, whose domain is c, and whose range is the XML data type that corresponds to t (all prefixed by the proper namespace). Table 4 shows the OWL property corresponding to field cadence of the class Bike.

public int
cadence;
	`<owl:DatatypeProperty`
	`    rdf:about="bikeNS:cadence">`
	`  <rdfs:domain`
	`    rdf:resource="bikeNS:Bike"/>`
	`  <rdfs:range`
	`    rdf:resource="xsd:int"/>`
	`</owl:DatatypeProperty>`

Table 4. Field with a basic type.

A field f of class c whose type is the class c' defined in the Java library corresponds to an OWL object property whose identifier is f, whose domain is c, and whose range is c' (all prefixed by the proper namespace). Table 5 shows the OWL counterpart of the field ft of class Bike, with type BikeFeatr. The namespace associated with BikeFeatr in the ontology is bikeFeatrNS.

public
BikeFeatr ft;
	`<owl:ObjectProperty`
	`    rdf:about="bikeNS:ft">`
	`  <rdfs:domain`
	`    rdf:resource="bikeNS:Bike"/>`
	`  <rdfs:range rdf:resource=`
	`    "bikeFeatrNS:BikeFeatr"/>`
	`</owl:ObjectProperty>`

Table 5. Field with type c.

Arrays. We translate fields defined as arrays of any dimension into properties whose domain is the class to which the field belongs to, and whose range is the special Array class defined in the Java ontology. Table 6 shows the OWL counterpart of a log array field of class Bike used for logging rides made with a bike.

public
String[] log;
	`<owl:ObjectProperty`
	`    rdf:about="bikeNS:log">`
	`  <rdfs:domain`
	`    rdf:resource="bikeNS:Bike"/>`
	`  <rdfs:range`
	`    rdf:resource="javaNS:Array"/>`
	`</owl:ObjectProperty>`

Table 6. Arrays.

4.2 Implementation of PlainExtractor

The constructor of the PlainExtractor concrete class initializes some instance variables designed to contain information about base URI, about importing external ontologies or just referencing them in the to-be-generated ontology, and the directory where the OWL output file will be stored. The values for all these variables are passed to the constructor with an instance of class Conf created by method createConf of Semlet. which processes the custom command line options.

The instance variable that represents the OntModel is instantiated using the OntModelSpec.OWL_DL_MEM value, which configures the OntModel to use the OWL DL language profile, no reasoner, and the in memory storage mode.

The predefined Action and Array OntClass classes are also added to the OntModel.

Construction of the ontology model. PlainExtractor processes all the packages provided by the Semlet by simply creating a new namespace that corresponds to the base URI of the ontology concatenated to the full name of the package. A namespace, constructed in the same way, is also added for each class contained in each package.

Field processing. Each field of a Java class is translated into one OntModel Object or Datatype property, depending on the declared type of the field, and the resulting property is added to the OntModel. The property range is defined as discussed in the *Fields* paragraph of Section 4.1. Array fields are Object properties whose range is the predefined Array OntClass.

In case of Object properties, the namespace corresponding to the declared type of the field is also added to the OntoModel.

The logical reference to the Java class c defining the field is obtained by setting the domain of the property to the OntClass corresponding to c.

Method processing. For each method an ObjectProperty having its range in the Action OntClass (as stated above) is added to the OntModel.

As in the case of the field, the logical reference to the Java class c defining the method is obtained by setting the domain of the property to the OntClass corresponding to c.

Writing the model. A dedicated method of PlainExtractor writes the obtained OntModel to a file, whose name is defined in the Conf object passed by Semlet. Several languages can be selected when writing an ontology: PlainExtractor uses "RDF/XML-ABBREV" which corresponds to using OWL abbreviations in order to obtain a more readable XML document.

5. Related work and conclusion

Very few proposals for extracting ontologies from software artifacts exist. Those we are aware of were born inside research groups working in the Software Engineering – an, in particular, Reverse Software Engineering – field. In the summary given by Table 7, contents written in ***bold italic*** refer to those features that our proposal shares with other ones.

Welty [25] describes a manual method for generating a CLASSIC [5] ontology that represents all the data- and instruction-level features of a SmallTalk source code fragment including value assignments, method creation and calls, switch and returns. Welty's proposal is the only one placed at this level of granularity: in the other proposals the implementation of classes is not considered.

Yang, Cui, and O'Brien [26] extract an RWSL [27] ontology from legacy systems by (i) capturing information by means of a Reengineering Assistant [28], (ii) analysing the obtained information to construct classes, relations, functions and instances, (iii) form the ontology, and (iv) evaluate, validate, and document it. The process is fully automatic and is demonstrated on a COBOL program.

Many Sabou's papers deal with ontology extraction from software artifacts. In [14], she describes a semi-automatic approach for extracting a single domain ontology from the Javadoc documentation of multiple Java libraries (that form the "corpus") in the same domain. The extraction of parts of speech and of verb-noun pairs from the corpus is automatic using an existing Part Of Speech tagger, while the identification of relevant pairs and the ontology construction are manual. The paper says nothing on the format of the resulting ontology. In [4] that approach is extended to take multiple sources (source code, software manuals, discussion forums) as input for generating OWL ontologies, and the process is fully automatic.

Jin and Cordy [11] describe the OASIS methodology for reengineering tools to share services and assist maintainers in carrying out software analysis and program comprehension tasks. A manually built domain ontology, represented in English as a cross-referenced compilation of representational concepts and services, is used by conceptual services adapters to share and filter services.

Zhang, Witte, Rilling, Haarslev [29] developed the SOUND (Software Ontology for UNDerstanding) environment to support ontology-based program comprehension. A Software Ontology is divided into the Source Code Ontology and the Documentation Ontology. The OWL-DL Source Code Ontology has been designed by hand to formally specify major concepts of Object-Oriented Programming languages, and has been populated in an automatic way.

Zhou, Kang, Chen, Yang propose OPTIMA, an Ontology-based PlaTform-specIfic software Migration Approach [30]. The OWL-DL ontology integrated in OPTIMA has been designed and implemented by hand, following a well defined 8-steps methodology. Feature location techniques are used to build the links between ontology concepts and related source code. Based on proposed domain ontology, program transformation rules are defined for software migration between different platforms.

OntoNaviERP by Hepp and Wechselberger [10] uses ontologies and automatic annotation of large HTML software documentation of Enterprise Research Planning (ERP) systems in order to improve their usability and accessibility. The OWL skeleton ontology is generated in a semi-automatic way and the Wordnet plug-in for Protégé has been used to augment the concepts by synonyms and lexical variants.

The proposal by Ratiu, Feilkas, and Jürjens [13] shares with Sabou's one [14] the same aim of building domain ontologies from APIs that target the same domain. In order to identify domain concepts based on similarities of several APIs, possibly written in different OO languages, a graph-based representation of the program elements from the public interface of the APIs is used and names of program elements are explicitly modeled. A graph-matching algorithm that uses the similarity of program element names and the similarity of paths is then run on the graphs extracted from APIs to produce a single RDF-like ontology.

Wang, Gibbins, Payne, Saleh, Li [24] abstract a formal specification in Z [15] from procedural program code (languages mentioned in their paper include Pascal, COBOL, FORTRAN, C), verify and validate the formal model, and automatically generate domain ontology and semantic Web service markup from specifications.

Finally, it is worth mentioning the EU-funded "Transitioning Applications to Ontologies" TAO project (IST-2004-026460) whose goal is to make transitioning existing legacy applications to ontologies fast and effective, thus allowing companies to build a reusable transitioning process; minimize consulting time during migration and integration; minimize costs; reduce integration overheads and limit risk. Two of the papers mentioned above [4, 24] are tightly related to this project.

Besides the evident shallow differences between the proposals discussed above and ours regarding the extraction method (automatic, semi-automatic, manual), the granularity of the extracted ontology, the source software artifact, and the target ontology language, there are even deeper differences in the aim of the work and in the followed engineering approach.

- **Aim.** 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

Year	Authors	Meth.	Gran.	From	To	For
1997	Welty [25]	M	Instr.	SmallTalk code	CLASSIC [5]	Program understanding and maintainance
1999	Yang, Cui, O'Brien [26]	***A***	***API***	Procedural code (runn. ex. COBOL)	RWSL [27]	Program understanding and legacy system engineering
2004	Sabou [14]	SA	***API***	***Java***, Javadoc	RDF-S, ***OWL***	Building domain ontologies from APIs that target the same domain
2005	Jin, Cordy [11]	M	Serv.	Services offered by SW tools	English	Re-engineering tools to share services and assist maintainers
2006	Zhang, Witte, Rilling, Haarslev [29]	SA	***API***	***Source code*** and documentation ***of Java-like lang.*** (runn. ex. Java)	***OWL***	Program understanding and maintenance
2006	Bontcheva, Sabou [4]	***A***	***API***	Multiple sources (source code, software manuals, discussion forums)	***OWL***	Software artifacts maintenance and re-use
2007	Zhou, Kang, Chen, Yang [30]	M	***API***	Procedural code (runn. ex. C)	***OWL***	Software migration
2008	Hepp and Wechselberger [10]	SA	***API***	Documentation of the ERP software	***OWL***	ERP software documentation annotation
2008	Ratiu, Feilkas, Jürjens [13]	***A***	***API***	***Java-like code*** (runn. ex. Java, .NET, Eclipse)	RDF-like	Building domain ontologies from APIs that target the same domain
2008	Wang, Gibbins, Payne, Saleh, Li [24]	***A***	***API***	Z model and languages for which "Z model extractors" exist	***OWL***	Transitioning legacy applications to semantic Web Services
2010	Ancona, Mascardi, Pavarino	***A***	***API***	***Java code***, bytecode †	***OWL***	Providing a customizable framework for machine-readable API documentation

Meth.: ontology extraction method (**M**, manual; **SA**, semiautomatic; **A**, automatic).

Gran.: granularity level of the created ontology (**instr**, the ontology represents each single instruction in a piece of code; **serv.**, service offered by a software application; **API**, Application Programming Interface).

From: types of the documents from which the ontology is extracted; when a running example is provided by the referenced paper, we report the running example language (**runn. ex.**) .

To: format of the extracted ontology.

For: purpose of the system.

† Bytecode can be used as an input to our framework if Classdoc is used instead of Javadoc, and with the same limitations that hold for the Classdoc usage.

Table 7. Systems that extract ontologies form software artifacts

basic Semlet and implementing the Extractor interface that the framework provides, 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 of results achieved inside the Java community.** 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).
- **Reuse of results achieved inside the ontology community.** 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 proposals mentioned above, they are meaningful words derived by means of a natural language processing stage from the names of source code entities.

As far as the last point is concerned, our choice is intentional 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. The graph matching algorithm uses a string-based similarity measure for understanding in which cases two nodes of two graphs represent the same concept, and also exploits structural similarity between 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 hundreds of available ontology matching algorithms [7] for generating the resulting merged ontology.

We claim that our approach is much more flexible that Ratiu et al.'s one: according to the reference web site for ontology matching projects, `http://www.ontologymatching.org/`, almost 70 papers on ontology matching algorithms were published in conferences and journals only in 2009 and 2010, and more than 30 systems are available to the research community. Among those ontology matching algorithms and systems, any researcher can find the right solution for her needs, from algorithms that use simple string-based approaches, to those that look at ontology concepts as pieces of information and hence exploit natural language processing techniques and WordNet to cope with their semantics, to those that look at the ontology structure induced by the *isA* relation, to those that exploit DL reasoners for checking the matching consistency, and many others.

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 [6, 12] could be used to help creating a unified library from different existing ones whereas multilingual ontology mapping [16] 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.

Because of the promising and still almost unexplored scenarios that ontology-based documentation opens, we hope that our step in this direction may be the first one of a long and profitable journey: the main future development of our research work will consist in extending and consolidating our working prototype in order to make it a fully-fledged framework suitable for the needs of a large community of users.

References

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

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[3] T. Berners-Lee. Semantic Web - XML2000, 2000. `http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html`.

[4] K. Bontcheva and M. Sabou. Learning ontologies from software artifacts: Exploring and combining multiple sources. In *2nd International Workshop on Semantic Web Enabled Software Engineering, SWESE'06, Proceedings*, 2006.

[5] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In *Principles of Semantic Networks*, pages 401–456. Morgan Kaufmann, 1991.

[6] D. Dou, D. Mcdermott, and P. Qi. Ontology translation by ontology merging and automated reasoning. In *EKAW Workshop on Ontologies for Multi-Agent Systems, EKAW'02, Proceedings*, pages 3–18, 2002.

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

[8] 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.

[9] T. Gruber. Definition of ontology. In *Encyclopedia of Database Systems*. Springer, 2008. To appear.

[10] M. Hepp and A. Wechselberger. OntonaviERP: Ontology-supported navigation in ERP software documentation. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan, editors, *7th International Semantic Web Conference, ISWC 2008, Proceedings*, volume 5318 of *LNCS*, pages 764–776. Springer, 2008.

[11] D. Jin and J. R. Cordy. Ontology-based software analysis and reengineering tool integration: The OASIS service-sharing methodology. In *21st IEEE International Conference on Software Maintenance, ICSM 2005, Proceedings*, pages 613–616. IEEE Computer Society, 2005.

[12] N. F. Noy and M. A. Musen. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *7th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence, AAAI/IAAI 2000, Proceedings*, pages 450–455. AAAI Press / The MIT Press, 2000.

[13] D. Ratiu, M. Feilkas, and J. Jürjens. Extracting domain ontologies from domain specific APIs. In *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, Proceedings*, pages 203–212. IEEE, 2008.

[14] M. Sabou. From software APIs to web service ontologies: A semi-automatic extraction method. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *3rd International Semantic Web Conference, ISWC 2004, Proceedings*, volume 3298 of *LNCS*, pages 410–424. Springer, 2004.

[15] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., 1989.

[16] C. Trojahn, P. Quaresma, and R. Vieira. A framework for multilingual ontology mapping. In N. Calzolari, K. Choukri, B. Maegaard, J. Mariani, J. Odjik, S. Piperidis, and D. Tapias, editors, *6th International Language Resources and Evaluation, LREC'08, Proceedings*. European Language Resources Association (ELRA), may 2008. http://www.lrec-conf.org/proceedings/lrec2008/.

[17] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition) – W3C Recommendation 26 November 2008, 2004.

[18] W3C. Namespaces in XML 1.0 (Second Edition) – W3C Recommendation 16 August 2006, 2004.

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

[20] W3C. RDF Vocabulary Description Language 1.0: RDF Schema – W3C Recommendation 10 February 2004, 2004.

[21] W3C. RDF/XML Syntax Specification (Revised) – W3C Recommendation 10 February 2004, 2004.

[22] W3C. SPARQL Query Language for RDF – W3C Recommendation 15 January 2008, 2008.

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

[24] 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.

[25] C. A. Welty. Augmenting abstract syntax trees for program understanding. In *International Conference on Automated Software Engineering, ASE'97, Proceedings*, pages 126–133. IEEE Computer Society, 1997.

[26] H. Yang, Z. Cui, and P. O'Brien. Extracting ontologies from legacy systems for understanding and re-engineering. In *23rd International Computer Software and Applications Conference, COMPSAC'99, Proceedings*, pages 21–26. IEEE Computer Society, 1999.

[27] H. Yang, X. Liu, and H. Zedan. Tackling the abstraction problem for reverse engineering in a system re-engineering approach. In *International Conference on Software Maintenance, ICSM'98, Proceedings*, pages 284–293. IEEE Computer Society, 1998.

[28] H. Yang, P. Luker, and W. C. Chu. Measuring abstractness for reverse engineering in a re-engineering tool. In *International Conference on Software Maintenance, ICSM'97, Proceedings*. IEEE Computer Society, 1997.

[29] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev. Ontology-based program comprehension tool supporting website architectural evolution. In *8th IEEE International Symposium on Web Site Evolution, WSE'06, Proceedings*, pages 41–49. IEEE Computer Society, 2006.

[30] H. Zhou, J. Kang, F. Chen, and H. Yang. OPTIMA: An Ontology-based PlaTform-specIfic software Migration Approach. In *7th International Conference on Quality Software, QSIC'07, Proceedings*, pages 143–152. IEEE Computer Society, 2007.

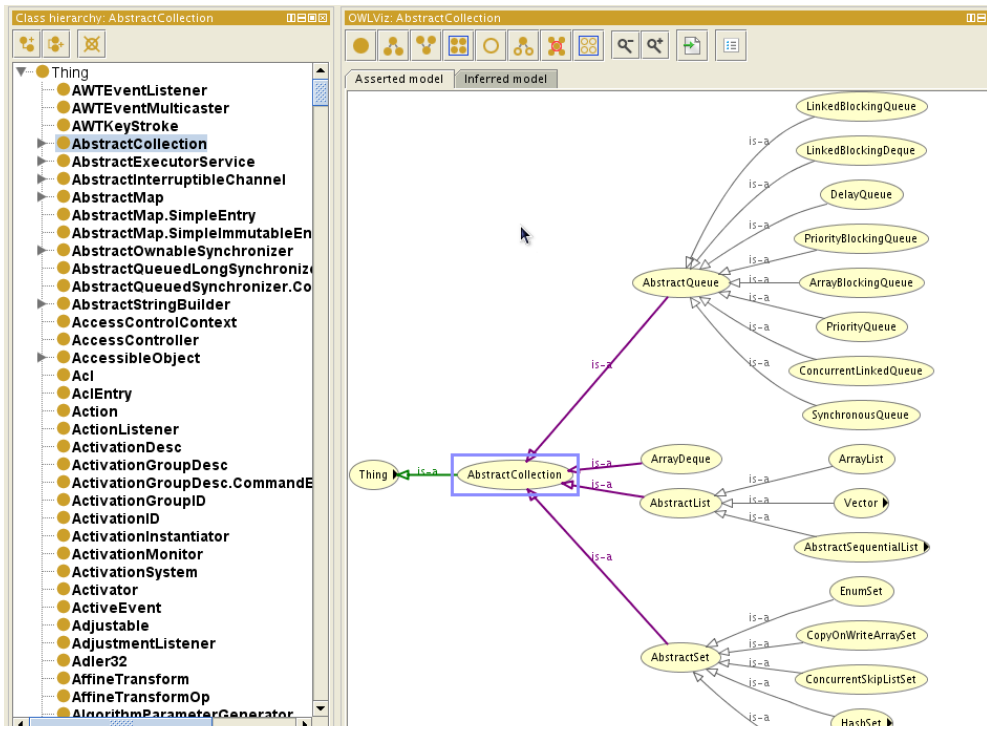

Figure 3. Part of the Java ontology generated by the PlainExtractor from the core Java API 1.6 library