INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians–
Universität
München
LMU

Java2OWL
A System for Synchronising Java and OWL
Version 1.1

Hans Jürgen Ohlbach

http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-2012-2, March 2012

Java2OWL – A System for Synchronising Java and OWL Version 1.1

Hans Jürgen Ohlbach
Institute for Informatics
University of Munich
E-Mail: ohlbach@lmu.de

April 17, 2012

Abstract

Java2OWL is a Java software library for synchronising Java class hierarchies with OWL concept hierarchies. With a few extra annotations in Java class files, the Java2OWL library can automatically map Java class hierarchies to OWL ontologies. The instances of these Java classes are automatically mapped to OWL individuals and vice versa. OWL reasoners can be used as query processors to retrieve instances of OWL concepts, and these OWL individuals are mapped to corresponding instances of the Java classes. Changes of the Java object's attributes are automatically mapped to changes of the corresponding attributes of the OWL individuals, thus keeping Java and OWL synchronised. With minimal programming overhead the library allows one to combine the power of programming (in Java) with the expressivity and the reasoning power of OWL. This paper introduces the main ideas and techniques. Examples in the appendix illustrate how the system operates. The Java2OWL Application Interface (API) can be found in the Javadoc generated documentation.

Contents

1 Motivation

Object oriented programming languages like Java combine the procedural aspects of programming with some declarative-logical features contained in the Java class hierarchy and the instance-class relationship. Whereas the procedural aspects are as strong as one can expect from modern programming languages, the logical aspects are rather restricted. In particular

- the ontology, i.e. the logical structure, implicitly contained in the class hierarchy of Java programs is only accessible from within the program and can not be exported to other systems;
- the instance-class relationship is fixed at the creation time of an object. Objects can not live independently of classes and can not change their membership relation to classes;
- the class hierarchy in Java is a tree; no multiple inheritance is possible;
- there is no logical reasoning available. This means in particular that fine grained selection of objects is difficult. For example, queries like "select all students in the Bachelor programme who passed the exam X last year and who live close to the university" can not be evaluated with the Java built-in mechanisms.

Nevertheless, these are desirable features, which is indicated by the rising popularity of modelling languages like UML, and in particular by the more and more widespread use of the logic-based Ontology Working Language (OWL[1]). OWL is the W3C-standard for Description Logics [1]. Its main features are

- one can define ontologies independently of any particular application or programming language;
- the OWL-language is very expressive and more features are being added as one learns how to extend the logical calculus for them;
- since it is logic based, various forms of reasoning can be performed;
- the class hierarchy can be a DAG, thus, multiple inheritance is allowed;
- individuals can live independently of classes. Their membership relation with classes is determined by logical reasoning, and it can change when new information about an individual becomes available;
- there are expressive languages which can be used to pose complex queries to an ontology. They are evaluated by logic reasoners.

The disadvantage of OWL is the lack of procedural features; OWL is no programming language. For an OWL ontology together with an A-Box (a set of individuals) one can draw all possible inferences, i.e. compute the class hierarchy and the membership relation of the individuals with the classes. After this, however, the ontology becomes a static object which can do nothing by itself.

An OWL ontology just living in a file system is not of much use. Therefore application interfaces (API) have been developed which allow programs to load an ontology, access its data, manipulate the ontology, and save it back to files, all within ordinary programs. The most recent one is the Java OWL API[2] for OWL-version 2 [5], developed as part of the CO-ODE project[3] and the TONES project[4] in the Manchester University[5]. It allows one to load, access and manipulate the structure of OWL ontologies and to integrate them with OWL reasoners.

Java programs using the OWL API for loading ontologies have two class hierarchies in parallel, the Java class hierarchy and the OWL class hierarchy. For applications where these hierarchies are sufficiently different, this is no problem. If, however, the class hierarchies overlap, one gets a considerable synchronisation problem. Overlapping class hierarchies are of interest if one wants to add procedural features to the classes. Since OWL has no procedural features, the only choice is to define corresponding Java classes with the corresponding methods.

[1]OWL: http://www.w3.org/TR/owl2-overview/
[2]OWL API: http://owlapi.sourceforge.net/
[3]CO-ODE project: http://www.co-ode.org/
[4]TONES project: http://www.tonesproject.org/
[5]Manchester University: http://owl.cs.manchester.ac.uk/

As an illustrating example, consider the partial modelling of university structures. There are people, students, tutors, lecturers, professors, secretaries, technical staff etc. There are programmes of study, Bachelor, Master and PhD programmes. There are lectures, seminars, excursions etc. There are scientific areas, natural sciences, humanities, social sciences etc. All this can be modelled in OWL, but there is no chance to add procedural features to these OWL classes. This can only be done in corresponding Java classes. Combining Java classes and their methods with corresponding OWL classes as the basis of OWL-reasoning, however, may turn out to be a very useful thing. The Java methods do the computation, and the OWL classes can in particular be used to select sets of individuals by determining the instances of OWL concept expressions. This is very analogous to querying databases. The difference is that OWL concept expressions are more expressive, and they do not depend on a particular database scheme.

The Java2OWL library presented in this paper targets the synchronised coexistence of Java classes and OWL classes. It makes it possible to exploit the strengths of both systems without too much programming overhead.

The next chapter illustrates a typical workflow when developing and running an application which uses the Java2OWL library. Since Java2OWL brings together the Java world with the OWL world, including Description Logics and the OWL API, there is a sometimes quite confusing usage of notions. In order to reduce the confusion at least a bit and make the paper better comprehensible, Section 3 gives a short overview over the most important notions and their usage in the different systems. A more general introduction into the different parts of the library and the ideas behind them is given in Section 6. Technical details are presented in Section 7. This section should give the reader a more precise idea how the system operates. For using it in a concrete application, the Javadoc generated documentation is recommended. A few performance tests are documented in Section 9. Finally, the Appendix addresses aspects which have not directly to do with the main operations of the Java2OWL system, but which are necessary to know for writing concrete applications. How to install all the components necessary for the Java2OWL library is documented in Appendix A. Appendix B gives an overview about the Manchester Syntax for writing OWL concept expressions. Appendix C presents the mechanism for logging events and errors in the system. Appendix D gives an overview about the correlations between the Java data types and the OWL data types. Finally, Appendix E contains some examples which illustrate how the system works.

The report gives no introduction to Java and OWL. The reader of this report should therefore be quite familiar with Java and have at least some understanding of the meta-programming facilities in Java (Java reflection). Some knowledge about OWL and a basic understanding of the OWL API are also very helpful. Programmers wanting to use the Java2OWL library should of course be familiar with OWL and the OWL API.

2 A Typical Workflow

A typical workflow in the development and application of Java programs using the Java2OWL library consists of two major phases:

1. the preparation phase, which consists of the programming and the compilation phase,
2. the Java application phase which starts with the steps a Java application has to perform in order to work with the Java2OWL library.

2.1 The Preparation Phase

This phase consists of the following steps:

1. **Background Ontology:** Most applications which want to use ontologies do not start from scratch, but us already existing ontologies, or develop specialised ontologies for this application. Therefore the Java2OWL library assumes the existence of a “background ontology”

with predefined classes and properties. This background ontology can be loaded into the Java2OWL-compiler which then extends it with OWL classes generated from Java classes.

2. **Annotating Java Classes:** Java classes which are to be mapped to OWL classes must be annotated in a special way. The class-level annotations control the generation of OWL classes and the method-level annotations control the generation of the OWL-properties. Typically, the Java class has getter methods, setter methods, and for collection-valued attributes, adder and remover methods. The annotations of the getter methods are sufficient to tell the system how to access and manipulate the Java-attributes and to synchronise them with the OWL-properties. The details can be found in Section 6.1.

3. **Compiling the Extension Ontology:** The annotated Java classes are mapped to newly generated OWL classes which are added to the background ontology. The getter methods are mapped to OWL-properties, either data properties, mapping objects to data types like integer, or object properties, mapping objects to other objects.

   The so extended background ontology is then saved as "extension ontology". Of course one can use the background ontology for different purposes and generate different extension ontologies. The `J2OCompiler` is explained in Section 6.2.

4. **Extending the Extension Ontology:** In some applications it may be necessary to manually extend the extension ontology, for example by adding special individuals. Therefore one can manipulate the extension ontology outside the Java2OWL-system in an almost arbitrary way before it is used in the application program. One should, however, not delete items from the Java-generated classes. Since OWL provides an include mechanism, it is possible to extend the extension ontology by just including it into another ontology. This way, the generated extension ontology itself need not be touched when it is extended further. This is in particular useful if the Java program is changed and a new version of the extension ontology is generated. It is not necessary to redo the extension of the extension ontology.

5. **Injecting Synchronisation Code into the Java classes:** Since Java classes and OWL classes coexist in the Java2OWL system, their instances also coexist. This means, for each Java object which is an instance of an annotated Java class, there is a corresponding OWL individual. This OWL individual should mirror the attributes of the Java object. If the Java object changes its attributes, the changes should automatically trigger changes of the attributes of the OWL individual. The code for forwarding the changes of the Java-attributes to the OWL side must be contained in the setter, adder and remover methods of the Java class. In order to relieve the Java programmer from writing this "synchronisation code" the Java2OWL library contains a "Synchroniser" class which automatically injects the "synchronisation code" into the compiled Java classes. This can be done either in a separate step after the ordinary Java compilation, or it can be done "on the fly" in the class loader. In this case a Java-agent[6] must be loaded when the program is started. Its premain-method adds a corresponding transformer to the system class loader. For details see Section 6.3.2.

2.2 The Java Application Phase

As soon as all the preparations have been successfully completed, i.e. the Java application with the annotated classes has been build and the extension ontology is ready to be used, the application can start. These are the typical steps, the application must do before its main job can start:

1. **Setting up the system:** This is simply done by creating a J2OManager. The J2OManager itself creates an J2OOntology manager, a J2OClass manager, a J2OIndividual manager and a logger for logging important events. It installs a default OWL-reasoner. The OWL-reasoner may be changed later on.

   One can create several J2OManagers, but they are completely independent of each other and do not interact with each other.

[6] Java-agent: http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html

2. **Loading the Extension Ontology:** The J2OOntology manager must then be asked to load the extension ontology.

3. **Linking the Extension Ontology:** In this step the internal data structures are created which keep track of the correspondences between Java classes and OWL classes.

4. **Creating Java-Objects and OWL individuals:** The system is now ready to create instances of classes. It is possible to instantiate Java classes and let the Java2OWL-system map them to OWL individuals. The other direction is also possible. OWL individuals contained in the A-Box of the extension ontology can be mapped to instances of the corresponding Java classes.

5. **Keeping Java-Objects and OWL individuals synchronised:** Changes to the attributes of instances should only be done by calling setter, adder, and remover methods of the Java-objects. The injected synchroniser-code automatically forwards the changes to the OWL-side. The OWL API allows for changing the attributes of OWL individuals directly, but these changes will then not be mapped to the Java side. Therefore this is not recommended.

6. **Reclassification of Instances:** OWL is logic based. Therefore OWL reasoners can derive information which was only implicitly contained in the OWL ontology generated from the Java side. This derived information can then be mapped back to the Java side. There are two typical situations where this can occur:

   - An object's class may be changed: Consider a Java class `Student` with attribute `semester`, and a subclass `Freshman` whose semester is frozen to the value 1. An instance of the Java class `Student` whose semester is set to 1 should now become an instance of `Freshman`. This is not directly possible in Java. In the Java2OWL-system, however, the Java-objects are contained in wrapper-objects. Therefore it is possible to exchange a Java-object inside a wrapper object with another Java object. In the `Student`/`Freshman`-example this would be performed by creating an instance of `Freshman` and transferring all non-static fields of the `Student`-instance to the new `Freshman`-instance. The new `Freshman`-instance then replaces the `Student`-instance inside the wrapper object.
   - An object's attributes may be changed: Consider a Java class with a *transitive* `has-Part` attribute. If this is mapped to OWL then an OWL reasoner can derive the transitive closure of the `has-Part` relation. Mapped back to Java the `has-Part` attribute is automatically filled with the transitive closure of the `has-Part` relation (see the Example E.3).

   The application program can ask the Java2OWL-system at any time to do this reclassification of instances whose attributes have been changed.

3 Notions

Java2OWL brings the world of Java programming together with the OWL world. OWL is a standardised realisation of Description Logics. OWL itself, however, is still an abstract specification. The concrete technical realisation a programmer has to deal with is the OWL API. Java, OWL, the OWL API and Description Logics deal with the same or quite similar things, but each one uses its own lingo with sometimes the same, and sometimes different notions for the same or similar things. In order to make this report comprehensible, we need to clarify the technical notions.

Class, class expression, concept, concept expression: From a mathematical point of view, classes, class expressions, concepts, concept expressions all stand for sets of objects. A Java class specifies the set of instances which can be generated with the constructor methods. The corresponding notion in the OWL API is `OWLClass`. An `OWLClass`, however, can not

be instantiated in the same way as a Java class. An OWL reasoner can only find out that a particular individual is an instance of an OWL class.

OWL classes in the OWL API and in OWL form the signature elements of a formal language for specifying new sets by combining features of other sets, the *class expressions* (OWL lingo) or *concept expressions* (Description Logic lingo). The formal language for writing class expressions which is used in the OWL API is the Manchester Syntax[7] (App. B). It is a user-friendly compact syntax for OWL 2 ontologies.
An example for a class expression in Manchester Syntax[8] is

Person and hasAge some int[>= 65].

It specifies the set of persons older than 64.

In the OWL API there is a parser for class expressions in Manchester syntax. It turns them into instances of the class `OWLClassExpression`:

The notion *concept* is used in Description Logics for what in the OWL API is a `OWLClass` of `OWLClassExpression`.

In this report we shall use the notions *class* and *class expression*. If it is necessary to distinguish between Java and OWL, we write 'Java class' or 'OWL class'.

Objects, instances, individuals: All these name members of the sets named class, class expression, concept or concept expression. The notion 'instance' is used in Java for objects generated by a constructor of a Java class. Whereas 'instance' implicitly expresses the relation to the class whose constructor created it, the term 'object' is used if the relation to the class is not important. Since there are class hierarchies in Java, an object c generated by a class A which is a subclass of a class B is also considered as instance of B.

In Description Logics and in OWL one speaks of 'individuals'. In contrast to Java, where objects are instances of some class, individuals in Description Logics and OWL exist independently of classes. A set of individuals as part of an ontology is sometimes called "A-Box".

The OWL API distinguishes further between named individuals (`OWLNamedIndividual`) and anonymous individuals (`OWLAnonymousIndividual`). Java2OWL works only with named individuals when it maps Java objects to OWL.

In this report we shall us 'object' and 'instance' for Java objects and 'individual' for OWL individuals.

Attributes, instance variables, fields, properties, relations, features, roles, role fillers: In our context, attributes, instance variables, fields, properties, relations, features, roles, all stand for functions or binary relations. Partial functions map a given object to at most one other object. Total functions map a given object to exactly one other object. Binary relations map a given object to any number of other objects ('role fillers' in Description Logic lingo).

In Java one speaks of 'instance variables' and 'class variables'. Instance variables map Java objects to some value, whereas class variables map classes to some value. The Java reflection system uses the notion (and class) 'Field' for both, instance and class variables. It is good Java practice not to expose fields outside the class where they are defined, but to access them via getter and setter methods. Therefore getter and setter methods also can represent attributes.

In Description Logics 'Abstract features' map individuals to other individuals, and 'concrete features' map individuals to some concrete domain (strings, integers etc.) [6]. 'Abstract features' are also called *roles*. For a binary relation r and an object x, the 'role fillers' for x is the set $\{y \mid r(x, y)\}$.

[7]Manchester Syntax: http://www.w3.org/2007/OWL/wiki/ManchesterSyntax
[8]Manchester Syntax: http://www.co-ode.org/resources/reference/manchester_syntax/

In OWL, 'properties' is the generic term for both functions and binary relations. The OWL API distinguishes `OWLDataProperty` and `OWLObjectProperty`. Both can be functions or binary relations. `OWLDataProperty` map individuals to basic data types. `OWLObjectProperty` map individuals to other individuals.

In this report we shall use 'attribute' as the generic term for functions and binary relations, both in Java and in OWL.

The notion 'property' is reserved for OWL properties, either data or object properties.

4 Java Class Hierarchy Versus OWL Class Hierarchy

The intrinsic Java ontology which is implicitly contained in the Java class hierarchy is simpler than the OWL ontology: mono-inheritance causes the Java class hierarchy to be become a tree, and the constructor mechanism for generating new instances of classes causes Java objects to be an instance of exactly one Java class. In contrast to this, OWL has multi-inheritance, and OWL individuals may be instances of several classes. Mapping a Java ontology to an OWL ontology is therefore not problematic. This part of the Java2OWL library is purely technical.

The Java2OWL library, however has three further features which are more problematic:

- it can map OWL individuals back to Java objects,
- it can attach attributes to Java objects, which are not foreseen in the Java class definition, and which must therefore be immediately forwarded to the OWL side,
- it can reclassify Java objects when further information becomes available.

If OWL individuals are to be mapped back to Java objects, we have to deal with the problem that an OWL individual may be an instance of several OWL classes. Therefore one can not just create an instance of *one* particular Java class as corresponding Java object. If extra attributes can be attached at Java objects by forwarding them to the OWL side, this may cause a new situation. For a Java object o as an instance of the Java class C which corresponds to a unique OWL object o' as instance of the OWL class C' the OWL instance o' may suddenly become an instance of other OWL classes, and therefore there is no longer this unique correspondence between o and o'. The problem turns up when the Java object is reclassified to incorporate the new information.

The solution to these problems in the Java2OWL library so far is *experimental* because no real application where these problems turned up have been tried. The solution is to encapsulate the correspondences between Java objects and OWL individuals in *individual wrappers* (see the class `IndividualWrapper` 7.5.1). An individual wrapper encapsulates an OWL individual together with *several* Java objects. Java programs should therefore not work with Java objects in the usual way, but with individual wrappers.

The typical example which illustrates the multi-inheritance phenomenon is the OWL class `Amphibious-Vehicle` as subclass of `Ship` and `Surface-Vehicle`. Since in Java there can not be a class `Amphibious-Vehicle` extending `Ship` and `Surface-Vehicle`, there is a problem. In Java one could in principle define either `Ship` as a class and `Surface-Vehicle` as an interface, and then define a class `Amphibious-Vehicle` extending `Ship` and implementing `Surface-Vehicle`, or the other way round. In this case, however, there can not be instances of `Surface-Vehicle`, which might not be a good idea. Therefore when mapping an Amphibious-Vehicle instance from OWL to Java, Java2OWL creates an individual wrapper containing a Java Ship instance and a Java Surface-Vehicle instance. It takes care that the common attributes of both Java objects are pointer-equal.

Even when multi-inheritance is avoided at the OWL side, there can be situations where an OWL individual must be mapped to several Java objects. As an example consider a Java class `Student` and a Java class `Teacher` which have nothing to do with each other, except that the `Teacher` class has an attribute `teaches` (some students). Both `Student` and `Teacher` are mapped to the OWL ontology. Now suppose, `James` is an instance of `Student`, and we add the extra information that `James` is not only a student, but teaches another student `Tom`. This information is not attached

to the Java object `James`, but forwarded to the corresponding OWL individual `James'`. At this moment, `James'` becomes an instance of the OWL class `Teacher'`. If this information is mapped back to the Java side, besides the Java object `James` as instance of the Java class `Student`, we need also an instance, say `James-Teacher`, of the Java class `Teacher`. Both Java objects can be wrapped in an individual wrapper, which at first glance, solves the problem.

There is, however, a further problem. Suppose `James` does not teach another student `Tom`, but James teaches himself, `James`. The Java instance `James-Teacher` has an attribute `teaches` of type `Student`. This attribute has to be filled with James, but which James, the `Student` instance `James` or the `Teacher` instance `James-Teacher`? In this case it is clear by the type of the `teaches` attribute, that the Student instance `James` is the only Java object which can be put into the `teaches` attribute. In other examples, however, there may be several choices. The current implementation is such that it takes the first one whose Java class fits the class of the attribute. Only practical applications can show what is really needed in these situations.

5 Java Data Types Versus OWL Properties

Java objects have attributes (instance variables), and these attributes are typed. In this section it is explained which attribute types are mapped in which way to OWL properties.

OWL distinguishes two kinds of properties:

OWLDataProperty: these describe basic datatype-valued attributes of individuals. OWL has a number of basic data types built-in, for example, integer, float, double, etc., but also strings. Examples could be a string-valued `name`-attribute, or an integer-valued `age`-attribute.

OWLObjectProperty: these describe relations between OWL individuals. Examples are the `hasPart` relation between physical objects, or the `hasParent` relation between persons.

OWL properties can be *functional*, i.e. they map an OWL individual to a single value, or *relational*: they map an OWL individual to several values.

Depending on the type of the Java attribute, the following cases are distinguished:

Data Types: This case covers attributes which are primitive, i.e. boolean, byte, short, int, long, float, double, or of the corresponding wrapper types, or of type String. They are mapped to *functional OWLDataProperties*.

Collection Data Types: These types are either arrays or collections of Data types. Examples are `String[]` or `Set<Integer>`. Attributes of these types are mapped to *relational OWLDataProperties*.

Object Types: Attributes whose type is a Java2OWL-annotated class are mapped to *functional OWLObjectProperties*.

Collections Object types: Examples may be `Person[]` or `ArrayList<Person>` where Person is a Java2OWL annotated class. These attributes are mapped to *relational OWLObjectProperties*.

Map Data Types: Types like `HashMap<String,Integer>` are treated in a special way. For each key parameter in such a map a new *functional OWLDataProperty* is created, which maps the individual to the corresponding value part. The key class in a `Map<Key,Value>` can be any class whose instances can be turned into a *unique* string. Together with the attribute name, this string determines the OWLDataProperty name.

The following example explains the idea:
Consider a class with an attribute `HashMap<Locale,String> explanations`. For a given object it contains explanation strings for different languages, and the languages are determined by the Locale. Suppose the explanations contain a German string for the Locale 'de'

and an English string for the Locale 'en'. The whole map would be mapped to the two *functional OWLDataProperties* `explanations_de` with the German string and `explanations_en` with the English string.

Map Collection Data Types: These are types like `HashMap<Locale,String[]>` or `HashMap<Locale,Set<String>>`. They are treated in a similar way as above, but the keys of the maps are mapped to *relational OWLDataProperties*.

Map Object Types: An example could be an attribute `HashMap<Locale,Person> language-expert` which maps a Locale to a Person, where Person is a Java2OWL annotated class. In this case each key is mapped a *functional OWLObjectProperty*.

Map Collection Object Types: For this case the example could be an attribute `HashMap<Locale,Person[]> language-experts` which maps Locales to arrays of Persons. Each key is now mapped to a *relational OWLObjectProperty*.

5.1 Accessor Groups

It is good Java practice to keep the instance variables private and to access them via special *accessor methods*, typically getter and setter methods. Therefore the information the Java2OWL system needs about the attributes to be mapped to OWL is *not* taken from the instance variables themselves, but from the corresponding getter method. There must be a public getter method whose return type determines the way the attribute is mapped to OWL.

In the rest of the paper, we therefore do not consider Java instance variables any more, but the corresponding getter methods!

If the return type of the getter method is not primitive, there are usually a number of further accessor methods which can manipulate the attributes. Some of them are required by the application, some of them are necessary for Java2OWL, in particular if OWL individuals are to be mapped to Java objects.

We distinguish again the following cases which correspond to the above classification of the attribute types.

Data Types and Object Types: There may be several setter methods. All of them must accept the value to be set as *the first argument*. If OWL individuals are to be mapped to Java objects, there must be one setter method with *exactly one argument*.

As an example, consider an attribute `String name`, with a getter method `String getName(){return name}` and two setter methods `void setName(String name){...}` and `void setName(String name, boolean log){...}`. The first setter method is required if OWL individuals are to be mapped to Java objects.

Collection Data and Object Types: The following accessor methods make sense:

setter methods for assigning the whole collection at once.

adder methods for adding a single item to the collection.

remover methods for removing a single item.

a clearer method for clearing the whole collection.

The setter, adder and remover methods should take the object to be set, added or removed *as first argument*. If OWL individuals are to be mapped to Java objects, there must be an adder method with *exactly one argument*.

Map Data and Object Types: like `HashMap<Locale,String>`.
The following accessor methods make sense:

setter methods for assigning the whole map at once.

adder methods for adding a single (key-value) pair to the map.

remover methods for removing a single key.

a clearer method for clearing the whole map.

a getKey method for turning a String into the corresponding key data structure.

If OWL individuals are to be mapped to Java objects, there must be an adder method with *exactly two arguments*, the key and the value, and a getKey method. The getKey method, however, is only necessary if the key type is not a string. In the `HashMap<Locale,String>` example, the getKey method is needed to turn a string into a Locale.

Map Collection Data and Object Types: like `HashMap<Locale,Set<String>>`.
The following accessor methods make sense:

setter methods for assigning the whole map at once.

adder methods for adding a single (key-value) pair to the map.

adder methods for adding a single (key-collection) pair to the map.

remover methods for removing a (key-value) pair from the map.

remover methods with only a key argument for removing the whole collection which corresponds to the key from the map.

a clearer method for clearing the whole map.

a getKey method for turning a String into the corresponding key data structure.

If OWL individuals are to be mapped to Java objects, there must be an adder method with *exactly two arguments*, the key and the value (not the collection as a whole), and a getKey method. The getKey method, however, is again only necessary if the key type is not a string.

A getter method together with the corresponding setter, adder, remover, clearer and getKey methods form an **accessor group**.

6 Components of the Library

This section gives an overview of the main components of the system. It provides enough information that potential users know what the system can do for them. The technical details which are necessary to actually program with Java2OWL are contained in Section 7, the appendix and the Javadoc generated documentation.

6.1 Annotated Java classes

The first aspect a programmer who wants to use Java2OWL is confronted with, is the structure of the Java classes to be mapped to OWL classes. The following restrictions to the structure of the Java classes are important:

- There must be a constructor method with an empty argument list. It is used when OWL individuals are mapped to Java objects.
- All attributes which are to be mapped to OWL-attributes must be accessible by getter, setter, adder and remover methods.
- The mapping of Java classes to OWL classes is controlled by special annotations of class and method definitions.

Since Java 5 it is possible to annotate Java classes, methods and fields with extra information. Java comes with a set of built-in annotations like @Override. It is, however, also possible to define new types of annotations and to access them with the corresponding Java-reflection methods.

6.1.1 The Class-Level Annotation

A simple example illustrates the annotation of classes:

```
@J2OWLClass(name = "Person", OWLSuperClasses = "LivingThing")
public class TestPerson {...}
```

The `name`-attribute `Person` causes the mapping of the Java class `TestPerson` to a newly created OWL class `Person`. The `OWLSuperClasses`-attribute `LivingThing` causes the generated OWL class `Person` to become a subclass of the OWL class `LivingThing`, which must be part of the background ontology.
The specification of the `J2OWLClass` annotation is

```
public @interface J2OWLClass {
    String name() default "";
    String OWLSuperClasses() default "";
    String EquivalentClass() default "";
    boolean synchronise() default true;
    String naming() default "getName";}
```

The four attributes have the following meaning:

name specifies the name of the generated OWL class. If the name-attribute is omitted then the name of the Java class becomes the name of the OWL class.

OWLSuperClasses: This can be a comma-separated list of OWL classes. It causes the generated OWL class to become a subclass of the given OWL classes. These OWL classes must be contained in the background ontology. Of course, the Java-side class hierarchy is also mapped to corresponding OWL-subclass relationships.

EquivalentClass: This is an OWL class expression in Manchester syntax which causes the generated OWL class to be equivalent to the given class expression. To illustrate the effect of this construct consider a class `Student` with attribute `Semester` and a subclass

```
@J2OWLClass(EquivalentClass = "semester value 1")
public static class Freshman extends Student {
    public int getSemester() {return 1;}
```

The specification `extends Student` together with `EquivalentClass = "semester value 1"` cause the corresponding OWL class to be axiomatised:

`Freshman = Student and semester value 1`,

such that students in the first semester can be classified as `Freshman`. See Example E.2 for more details.

synchronise: This triggers the insertion of synchroniser code into the accessor methods. By default it is set to true, and the synchroniser code is inserted. If synchronisation is not necessary by some reason one can suppress the insertion of synchroniser code by setting this flag to false.

naming: OWL individuals can be identified by their International Resource Identifiers (IRI). A part of the IRI is an ordinary name, which has to be chosen in some way. In order to be able to associate OWL individuals generated from Java-objects with the original Java-objects, one can chose the name such that the correlation becomes obvious. To this end one can provide as `naming`-attribute the name of a String-valued getter method. The string returned by this method becomes the fragment-part of the IRI. The default is not the `toString`-method because this method usually yields a longer description of an object, and not a short name. It is recommended to program a String valued getName()-method which returns a short but

meaningful *and unique* identifier for the object. It is important that the identifier is unique because OWL can not distinguish different items with the same IRI.

If the `naming`-attribute is the empty string and there is no String-valued `getName()` method then a name `<OWL classname>_<counter>` is generated for the new OWL individuals.

6.1.2 The Method Level Annotations

In Section 5.1 it was explained that Java attributes which are to be mapped to OWL are characterised by *accessor groups*. The main part of an accessor group is a getter method which returns the attribute value to be mapped to OWL. The other components of an accessor group are setter, adder, remover, clearer and getKey methods. All of them are specified in a single annotation of the getter method.

A typical example for an annotated getter method is

```
@J2OWLProperty(name = "hasName", setter="setName")
public String getName() {return name;}

public void setName(String name) {this.name = name;}
```

The `@J2OWLProperty` has a larger number of attributes. The specification is:

```
public @interface J2OWLProperty {
    String name() default "";
    boolean local() default false;
    String setter() default "";
    String adder() default "";
    String remover() default "";
    String clearer() default "";
    String getKey() default "";

    boolean transitive() default false;
    boolean symmetric() default false;
    boolean asymmetric() default false;
    boolean reflexive() default false;
    boolean irreflexive() default false;
    boolean total() default false;
    int atleast() default -1;
    int atmost() default -1;

    boolean addRangeAxiom() default true;}
```

The meaning of these attributes is:

name: is the name of the corresponding OWL-property. If the name is the empty string then the name of the getter method is taken.

local: If this flag is set to true then the generated name is prefixed with the class name. This means in particular that OWL-properties generated from non-local getter methods can be used in more than one class. Examples where this makes sense are the `hasPart` relation or the `hasName`-property. If a getter method name G is mapped to some OWL-property name P then the system first checks whether a property with this name (i.e. its IRI's fragment equals P) is already known in the extension ontology or some of the ontologies imported by

the extension ontology. In this case it takes the previously introduced property. Only if no such property is found it generates a new one.

This way one can in particular map Java-attributes to OWL-properties for which further information has already been specified in the background ontology, for example functionality or transitivity.

setter,adder,remover: These are the comma separated names of the corresponding methods which are responsible for the same attribute. They all must accept the object to be set, added or removed as the *first argument*, or, in case of Map type attributes as the first two arguments (key-value pairs). The type of the argument of the setter methods must be such that the result of the getter method can be a legal first argument to these methods (autoboxing of primitive data types is supported). The type of the arguments of the adder and remover methods for non-Map type attributes must be such that the result of the getter method, or components of the result if the property is relational, can be a legal first argument to these methods (autoboxing of primitive data types is again supported). If the corresponding property is functional then at least one of the setter methods must have exactly one argument. If the corresponding property is relational then at least one of the adder methods must have exactly one argument. For Map-type attributes the first argument is usually the key type of the map, and the second argument must accept the value type, or the component type of the value type.

clearer: This must be the name of a parameter-less method which empties collection valued attributes.

transitive,symmetric,asymmetric,reflexive,irreflexive: These are properties of OWLObjectProperties. This makes sense because OWLObjectProperties are actually nothing else than binary relations between individuals. The corresponding axioms are added to the extension ontology.

total: This flag makes only sense for functional properties. If set to true it enforces that the function is total, but only for the OWL class generated from this particular Java class. A property like `hasName`, for example, may be total in a class A, i.e. their instances must have a name, and partial in a class B where their instances need not have a name.

atleast,atmost: These values are for relational properties. They specify the minium and maximum number of 'role fillers'. The default value -1 causes no such restriction to be added.

addRangeAxiom: The range type of the getter methods can be mapped to corresponding OWL classes or OWL-data types. If `addRangeAxiom` is true then the corresponding range type axiom is specified for the OWL-property generated from the getter method. This may help detecting hidden modelling errors, in particular when attributes in different classes are mapped to the same OWL-property.

getKey: denotes a method which can turn a string into the data structure which is used as the key-part in a Map-type attribute. If the key-part is already a string then there need not be a getKey method.

6.2 The Java2OWL Compiler

The Java2OWL compiler, implemented in the class `J2OCompiler`, generates OWL classes from annotated Java class files. For the preparation phase (see Sect. 2.1) it has a main method such that it can be called to generate the extension ontology from a given list of compiled Java classes. In the application phase (see Sect. 2.2) its methods can be used to link the Java classes with the OWL classes i.e. to build the internal data structures necessary for managing the correspondences between Java and OWL.

6.2.1 The Java2OWL Compiler in the Preparation Phase

In the preparation phase the Java2OWL-compiler can be invoked by calling

`java Java2OWL/J2OCompiler <parameters>`

If there are no parameters it just prints a help text. Otherwise the arguments to the program consist of filenames for Java .class files, directory names with Java .class files and a number of key-value parameters. There can be any number of Java .class files and directories.

The key-value pairs are

bgo ("background ontology", optional): The value is the filename for the background ontology. It can be omitted if there is no background ontology.

exof ("extension ontology file", mandatory): The value is the filename where the newly created extension ontology is to be saved.

exoi ("extension ontology IRI", optional): The value is the IRI for the newly created extension ontology. By default some reasonable name is created from the background ontology or from the user name.

sync ("synchronise", optional, default: false): The value is true/false. If true, then synchroniser code is injected into the class files.

reasoner (optional): The value is the name of the reasoner to be used. The following reasoners are available: [HermiT, Pellet, FaCT++]. The default reasoner is HermiT, and the default timeout for the reasoner is 10 seconds.

debug (optional, default: true) If the value is not false then the system is put into debug mode. In this case the first error causes the compiler to stop.

log (optional): The value 'all' causes *all* logs to be printed, otherwise only error messages are printed.

The program reads the background ontology, creates an extension ontology, reads the class files, analyses its annotations, fills the extension ontology with the translated Java classes and saves the extension ontology. Error messages are printed to System.out.

Notice that not all Java class files to be translated need to be given as input. For a given class C to be compiled to OWL the J2OCompiler automatically translates the following classes:

- all annotated superclasses of C,
- all annotated *static* inner classes of C,
- all annotated range type classes of the annotated getter methods in C.

For Map-type attributes like `Map<String,Set<Person>>` the component type of the map's value type, in the example `Person` is *not* automatically translated.

The translation need not be started manually. The main-method of the J2OCompiler can also be called from within another program. An example could be

```
String args = new String[]{"build/test/classes/TestPackage/TestVehicle.class",
              "exoi", "http://www.lmu.de/Vehicle.owl",
              "exof", "test/TestPackage/Vehicle.owl",
              "sync","true","debug","true"};
   J2OCompiler.main(args);
```

6.2.2 The Java2OWL-Compiler in the Application Phase

When an application is launched and the extension ontology is loaded, the application must link the Java classes with the OWL classes.

A typical sequence of Java statements for this purpose could be:

```
J2OManager manager = new J2OManager("TestManager");
J2OOntologyManager omg = manager.getOntologyManager();

omg.addIRIMapper(new AutoIRIMapper( new File("test/Java2OWL"),true);

OWLOntology extOntology =
    omg.loadOntology("test/Java2OWL/ExtensionOntology.owl", false);
omg.setReasoner("HermiT", true, false, 10000);

J2OCompiler compiler = manager.getCompiler();
compiler.class2OWL(TestStudent.class, TestAddress.class,
    TestScience.class, TestStudyProgramme.class);
compiler.finishTranslation(null);
```

The statement

```
omg.addIRIMapper(new AutoIRIMapper( new File("test/Java2OWL"),true);
```

adds a so called IRI-mapper to the ontology manager. Its purpose is to locate ontologies included by the extension ontology in the file system (see the OWL API[9] for details). The actual translation from Java to OWL is done in the two statements:

```
compiler.class2OWL(TestStudent.class, TestAddress.class,
    TestScience.class, TestStudyProgramme.class);
compiler.finishTranslation();
```

The first statement prepares the translation and collects the relevant data. The `finishTranslation`-method does all the remaining work. The `compiler.class2OWL`-method need not be given all the classes to be processed. Annotated superclasses, inner classes and range type classes of getter methods are automatically processed as well.

6.2.3 Compile Time Errors

The `J2OCompiler` analyses annotated Java class and combines the extracted structures with the background ontology. In this step it tries to detect as many programming errors as possible.

Here are examples for errors it can detect:

- Inconsistencies in the annotations of getter methods:

  ```
  @J2OWLProperty(name = "hasFriend",
        reflexive = true, irreflexive = true)
  public Set<Person> getFriends() {return friends;}
  ```

  This is a trivial inconsistency because a relation can not be reflexive and irreflexive at the same time.

- Java data types which can not be mapped to OWL data types:
  The Java data types which can be mapped to OWL data types are the primitive Java data

[9] OWL API: http://owlapi.sourceforge.net/javadoc/index.html

types boolean, byte, short, int, long, float, double and the String type. All other built-in Java data types, for example BufferedString, are not mapped to OWL data types. A specification like

```
@J2OWLProperty(name = "hasName")
public BufferedString getName() {return name;}
```

therefore causes an error.

- Classes without necessary Java2OWL annotations:
  Consider a class `Person` with a getter method

  ```
  @J2OWLProperty(name = "hasAddress")
  public Address getAddress() {return address;}
  ```

  This causes a functional Object Property `hasAddress` to be created, which maps individuals of type `Person` to individuals of type `Address`. If there is no class `Address` in the background ontology and the Java class `Address` is not annotated, the relation `hasAddress` makes no sense in the extension ontology.

The `J2OCompiler` can of course not detect all errors which may course trouble at run time. Therefore run time error handling is also necessary (see Sect. 6.4.2).

6.3 Synchronisation Between Java and OWL

6.3.1 Java to OWL Synchronisation

Java to OWL synchronisation means that changes to attributes of Java objects are forwarded to the corresponding OWL individuals. The Java2OWL library provides two different possibilities to do this:

Life Synchronisation: This means that all changes to the attributes of Java objects are immediately forwarded to the corresponding OWL individual. This is only possible if the annotated Java classes got 'synchroniser code' injected. Life Synchronisation can be turned on and off at any time. This can be done at class level, i.e. for all instances of a given class. The activation/deactivation of class level life synchronisation can be overwritten by activating/deactivating it for single objects. By default life synchronisation is deactivated at class level, and activated at object level. It can therefore be activated by calling `activateSynchronisation(...)` of the `J2OClassManager` class.

Block Synchronisation: This means that changes to the attributes of Java objects are not forwarded to OWL for a while, and at some time the application decides to do this 'en bloc' for all attributes of the object. The classes `IndividualWrapper` and `J2OIndividualManager` have methods `synchroniseWithOWL` for this purpose. If this is the only technique used in an application then no 'synchroniser code' need be injected into the annotated Java classes.

6.3.2 The Java2OWL Synchroniser

The `J2OSynchronizer`-class is used to insert extra 'synchroniser code' into the setter, adder, remover and clearer methods of the compiled Java class files. It can be used in two ways:

- the J2OCompiler calls it when the `sync`-flag is set to true,
- the `J2OSynchronizerAgent` calls it in its premain-method. This causes a further transformer to be installed in the class loader. This transformer calls the J2OSynchronizer to inject the synchronizer code when a compiled class is loaded. In this case one must start the Java application with `java -javaagent:J2OSynchronizerAgent.jar ...`

The second way is more comfortable because, as long as the Java2OWL annotations in a Java class are not changed, one can repeatedly recompile a Java class with the javac-compiler, because the synchroniser code is injected at load time. The first way makes the loading of compiled Java classes into an application more efficient because no manipulation of compiled Java class files is necessary. On the other hand, it means that each time a Java class is recompiled with javac, one has to call the J2OCompiler. A reasonable strategy is therefore to use the second method as long as a program is under development, and generate compiled Java classes with injected synchroniser code when the program is not changed any more.

6.3.3 The Structure of the Injected Synchroniser Code

The injected synchroniser code serves two purposes:

- it forwards changes to attributes of Java objects to the OWL side, and
- it maintains three flags:
  - the first one is to turn object-level synchronisation on and off,
  - the second one indicates changes to the attributes since the last block synchronisation,
  - the third one indicates changes to the attributes since the last reclassification.

  All three flags are part of the integer variable `Java2OWLFlags`.

The structure of the injected synchroniser code can be exposed with a Java decompiler, for example Jad[10]. First of all, a number of extra fields and a new method is inserted:

```
public static J2OIndividualManager j2OWLIndividualManager;
public int Java2OWLFlags;
private static Method J2OWL_getter_getName =
     J2OSynchronizer.getMethod(TestPackage/TestPerson, "getName");
...
J2OIndividualManager J2OSynchronisationActive() {
     Java2OWLFlags |= 6; // indicates changes to an attribute.
     return  (Java2OWLFlags & 1) != 0 ? null : j2OWLIndividualManager;}
```

The variable `j2OWLIndividualManager` has by default the value `null`. In this case no synchronisation is performed for all instances of the class. This variable is set and reset by calling the `activateSynchronisation-` or `deactivateSynchronisation` methods of the `J2OClassManager` (See Section 7.4). Since the `j2OWLIndividualManager`-variable is static it allows for activating or deactivating synchronisation for all instances of the class at once.

Individual activation/deactivation for single instances can be achieved by setting/resetting the rightmost bit of `Java2OWLFlags`. This can be done by calling the `activateSynchnisation-` or `deactivateSynchronisation` methods of the individual wrapper containing the object. The default value of `Java2OWLFlags` is 0 which causes that object-controlled synchronisation is turned on.

Static variables like `J2OWL_getter_getName` are created for all getter methods. They are bound at initialisation time to pointers to the getter methods themselves.

Each setter, adder, remover and clearer method is changed by inserting calls to methods which do the synchronisation. As an example, a typical setter method

```
public void setName(String name){
   this.name = name;}
```

is changed to

[10] Jad: http://www.kpdus.com/

```
public void setName(String name){
 if(J2OSynchronisationActive() != null)
        J2OSynchronisationActive().exchangePropertyValue(this,
          J2OWL_getter_getName, getName(), name);
  this.name = name;}
```

The actually injected code is slightly optimised and can not be represented in Java syntax.

The working of this code slice together with the method `J2OSynchronisationActive()` is quite sophisticated. The same variables `Java2OWLFlags` and `J2OIndividualManager`, and the same `J2OSynchronisationActive()` method get injected into each class. As an example, consider a class `Person` with the setter method `setName`, and a *subclass* `Studenten extends Person`. Both get the variables `Java2OWLFlags` and `J2OIndividualManager`, and the `J2OSynchronisationActive()` method injected. The value of the variable `J2OIndividualManager` may be different for the `Person` and for the `Studenten` class, maybe because synchronisation has been turned off for persons, but turned on for students. Suppose for an instance of the `Student` class, say `Paul`, the setter method `setName` method is called. The `setName` is actually part of the `Person` class and not of the `Student` class. Nevertheless, `this.J2OSynchronisationActive()` calls the `J2OSynchronisationActive()` method for `Paul`'s class, namely `Student`. The `J2OSynchronisationActive()` method therefore accesses the `J2OIndividualManager` variable of the `Student` class, and this is the variable which is relevant for the `Student` instance `Paul`. The same mechanism ensures that the value of the `Java2OWLFlags` variable which is relevant for `Paul` is chosen, namely the variable for the `Student` class, and not the variable of the `Person` class, although `setName` belongs to the `Person` class.

The bytecode manipulation for injecting the synchroniser code has been programmed with the Byte Code Engineering Library[11] (Apache Commons BCEL™) [3].

6.4 The J2OOntology Manager

The `J2OOntologyManager` is a kind of interface between the OWL API and the application. The OWL API consists of three main components:

- the `OWLOntologyManager` which stores all the data belonging to an ontology,
- a `OWLDataFactory` which creates the structures necessary to interact with the ontology,
- a reasoner. There can be several reasoners, but only one can be active within a J2OManager at any time.

Since the interaction with the OWL API is sometimes quite cumbersome, the `J2OOntologyManager` hides the pecularities of the OWL API. Its main tasks are:

- loading and saving ontologies;
- setting up reasoners;
- getting information about the components of the ontology;
- making changes to the extension ontology.

6.4.1 The Interaction with the OWL API

The OWL API stores information about an ontology in two ways:

- The `OWLOntologyManager` in the OWL API itself has an internal representation of the ontology. Various interface method can access this information. Since OWL is based on Description Logic, changes to the ontology are submitted to the OWL API as *logical axioms*. Whenever a change to the ontology is to be made, the `OWLDataFactory` must create a logical axiom and then either add it to the ontology, or remove it from the ontology.

[11] Byte Code Engineering Library: http://commons.apache.org/bcel/

- The reasoner must of course also have an internal representation of the ontology. Since reasoners in different programming languages are available (FaCT++, for example, is written in C++), the information about the ontology must be forwarded from the `OWLOntologyManager` to the reasoner.

Changes to an ontology are therefore done in three steps:

1. Logical axioms are created and stored in a list. This does not yet change the ontology itself at all.
2. The `OWLOntologyManager` is asked to apply the changes, i.e. to integrate the axioms into the ontology.
3. The new information is forwarded to the reasoner. This may either be done as soon as the `OWLOntologyManager` integrates a new axiom, or the changes may be buffered and made effective at a later step. In this case the ontology may have become inconsistent before the reasoner has a chance to check it. The `J2OOntologyManager` makes sure that whenever the reasoner is asked to do something, the buffer is first flushed to the reasoner.

6.4.2 Sources of Inconsistencies in the Extension Ontology

Inconsistency is not a relevant notion in an ordinary programming context. Therefore it is a reasonable question to ask where inconsistencies in Java2OWL-managed ontologies can come from. Certain sources of inconsistencies, which can be detected by the compiler, have been discussed in Sect. 6.2.3. In this section we discuss inconsistencies which may show up at run time.

Here are some examples for inconsistencies:

- Inconsistencies between the background ontology and the annotations of getter methods:
  In a background ontology, one might for example specify a *functional* property `hasName`, and in a Java class a getter method. An annotated getter method may be

  ```
  @J2OWLProperty(name = "hasName")
  public String[] getNames() {return names;}
  ```

  where the return type is an array. This is not inconsistent with the functionality of `hasName` as long as the result of the `getNames()`-method has just a single element. As soon as the `getNames()`-method returns a longer array, it becomes inconsistent with the required functionality of `hasName`.

  This is a case where a compiler could issue a warning, but the situation may be intended and there may be no problem at run time.

- Constraint Violations:
  The annotation

  ```
  @J2OWLProperty(name = "hasFriend", atmost = 3)
  public Set<Person> getFriends() {return friends;}
  ```

  specifies that one can have at most three friends. If `getFriends()` returns a set with more than three friends, this contradicts the constraint `atmost = 3`.

6.4.3 Checking the Consistency of the Extension Ontology

Checking for inconsistencies by the reasoner is an expensive operation. Therefore it is a strategic decision when to check the consistency of the extension ontology. During development and debugging it is useful to find inconsistencies as early as possible. To this end, the `J2OManager` can be put into *debug mode* by calling `setDebugMode(true)`. It causes each change to the ontology to be immediately forwarded to the `OWLOntologyManager` and to the reasoner and to ask the reasoner to immediately check the consistency.

If the system is not in debug mode, i.e. `setDebugMode(false)` has been called, then changes to the ontology are buffered as long as possible. The changes to the ontology are forwarded to the `OWLOntologyManager` and to the reasoner

- either when the reasoner is asked to compute some information,
- or when new information overwrites old information, and it is necessary to retrieve the old information in order to generate the corresponding remove-axioms.

These operations are hidden in the `J2OOntologyManager`. The only consequence for the application is that inconsistencies in the ontology may show up long after they have been introduced. In this case it may be difficult to figure out where the inconsistency originally came from.

6.5 The J2OClass Manager

The class `J2OClassManager` manages the correspondences between the Java classes and the OWL classes. To this end it has two auxiliary data structures:

`ClassWrapper` which stores the pair [Java Class, OWL Class] together with information about the mapped attributes (Section 7.4.1).

`PropertyWrapper` which stores the correspondences between the Java getter, setter, adder, remover and clearer methods on the one side, and the corresponding OWL-property on the other side.

If the `J2OClassManager` is asked to get for a Java class the correspondence to the OWL class it does not return an OWL class, but a `ClassWrapper`. From the `ClassWrapper` one can get the OWL class. The same holds for the attributes. The details can be found in Section 7.4.

6.6 The J2OIndividual Manager

The `J2OIndividualManager` manages the correspondences between the Java objects and the OWL individuals. The main purposes of the `J2OIndividualManager` are therefore

- mapping Java objects to OWL individuals and vice versa and
- synchronising the Java object's attributes with the corresponding OWL properties.

The correspondences between Java objects and OWL individuals are encapsulated in the class `IndividualWrapper`. Individual wrappers are actually the object level front end to the OWL ontology. Each individual wrapper contains one OWL individual and, due to multi-inheritance of OWL one or more corresponding Java objects. The most important operations, an individual wrapper can perform are:

- activating/deactivating life synchronisation
- block synchronisation with OWL
- reclassification of the Java objects,
- attaching extra attributes to the objects which are not foreseen in the Java class definitions. These attributes are actually attached at the OWL individual in the OWL ontolgoy.

For more details see Section 7.5.

7 Detailed Documentation of the Various Components

This section should provide sufficient details of the Java2OWL library's public interface to plan concrete applications. The remaining details which are necessary for actually implementing an application are contained in the Javadoc generated documentation. Only the Java2OWL classes and methods which are needed for programming the application are public. The compilation and synchronisation is done by the protected methods in the Java2OWL library. They need not be publicly exposed.

Most Java2OWL classes have a number of public methods which are only for administrative purposes. They are not listed here.

7.1 The Class `J2OManager`

This is the top manager for the Java2OWL system. It contains all the other necessary managers. One can create several instances of `J2OManager`, but all of them work with different ontologies. Each `J2OManager` is either in the status `COMPILE` or in the status `RUN`. This is indicated by the `enum J2OManager.Status`. Since a `J2OManager` creates all other managers and further relevant objects, the other classes have no public constructors.

The Public Interface

Constructor: J2OManager(String name, boolean debugMode)
creates a `J2OManager` with the given name and debug mode for the RUN status. (The COMPILE status is not exposed to the public.)

The `J2OManager` is only an administrator for the system. It plays no active part in the operations. Therefore there are only methods for accessing the other components of the system:

J2OOntologyManager getOntologyManager()
: returns the ontology manager.

J2OClassManager getClassManager()
: returns the class manager.

J2OIndividualManager getIndvidualManager()
: returns the individual manager.

J2OCompiler getCompiler()
: returns the Java2OWL compiler.

Logging getLogger()
: returns the logger.

void setDebugMode(boolean flag)
: puts the system in/out of debug mode. This flag is automatically forwarded to the other components of the system. Some of them, in particular the logger (see App. C), can for itself overwrite the general setting of the debug-mode flag.

7.2 The Class `J2OCompiler`

The `J2OCompiler` is in particular needed in the application phase for linking the Java classes with the OWL classes in the extension ontology. For this purpose it has the following public methods:

void class2OWL(Class... classes)

void class2OWL(Collection<Class> classes)
: These methods start the translation of the annotated Java classes into OWL classes. Only the internal data structures are created.

boolean finishTranslation()
finishes the translation of the annotated Java classes into OWL classes. The internal data structures are turned into OWL axioms. This method should be called after `class2OWL` has been called for *all* classes to be translated.

The whole linking process is split into these two parts because different getter methods of different classes can be mapped to the same OWL property. Therefore it is necessary to analyse *all* annotated Java classes first. Only when the whole information about the different getter methods is collected the system can do the actual linking.

7.3 The Class `J2OOntologyManager`

The `J2OOntologyManager` is the application interface to the OWL ontologies. In most cases its operation needs actually not be exposed to the application because the J2OClass manager, the J2OIndividual manager and the wrapper classes provide a more convenient interaction with OWL. For advanced applications it may, however, be useful to access the internal components directly. To this end there are a number of public methods available. The first group consists of pure administrative methods.

Administrative Methods:

OWLOntology getBackgroundOntology()
returns the background ontology.

OWLOntology getExtensionOntology()
returns the extension ontology.

void addIRIMapper(AutoIRIMapper iriMapper)
adds an IRIMapper to the ontology manager. The IRIMapper helps finding imported ontologies.

OWLOntology loadOntology(IRI iri, boolean background)
loads either the background ontology or the extension ontology from the given IRI.

OWLOntology loadBackgroundOntology(String name)
loads the background ontology from the given file.

OWLOntology loadExtensionOntology(String name)
loads the extension ontology from the given file.

boolean saveExtensionOntology(IRI iri)
saves the extension ontology to the given IRI.

boolean saveExtensionOntology(String filename)
saves the extension ontology to the given file.

boolean saveExtensionOntology()
saves the extension ontology to the same file where it was loaded from.

OWLDataProperty getFirstOWLDataProperty(String name)

OWLObjectProperty getFirstOWLObjectProperty(String name)
return the first OWLDataProperty/OWLObjectProperty in the extensionOntology's import closure whose IRI-fragment equals 'name', or null if there is no such property.

Set<OWLDataProperty> getOWLDataProperties(String name,boolean background)

Set<OWLObjectProperty> getOWLObjectProperties(String name,boolean background)
returns the OWLDataProperties/OWLObjectProperties in the ontology's import closure whose IRI-fragment equals 'name', or null if there is no such property. If `bachground = true` then the ontology is the background ontology, otherwise it is the extension ontology.

static Set<String> getReasoners()
returns the names of all available reasoners.

OWLReasoner setReasoner(String reasonerName, boolean buffering, boolean monitorProgress, int timeout)
installs a reasoner with the given name for the extension ontology. If `buffering = true` then a buffering reasoner is installed. This is recommended because otherwise each change to the ontology immediately invokes the reasoner. `monitorProgress = true` causes logs of the reasoner's operations to be printed. `timeout` is the timeout in milliseconds. If a particular invocation of the reasoner exceeds this timeout it is stopped.

boolean applyChanges(boolean checkConsistency)
This method forwards the change axioms to the extension ontology. If `checkConsistency = true` and the system is in debug mode then the changes are forwarded to the reasoner as well and the reasoner checks the consistency of the extension ontology. If it has become inconsistent then error messages are sent to the logger and the method returns false. If the system is not in debug mode then the reasoner is not invoked in this method.

The second group of methods may invoke the reasoner to retrieve information from the ontology. Invoking the reasoner may cause the following exceptions to be thrown: AxiomNotInProfileException, ClassExpressionNotInProfileException, FreshEntitiesException, InconsistentOntologyException, TimeOutException, ReasonerInterruptedException, UnsupportedEntailmentTypeException, ReasonerInternalException (See the OWL API[12] documentation for details). All these exceptions are caught by the `J2OOntologyManager` and an error message is logged. If the system runs in debug mode this error message causes it to stop. If the system is not in debug mode an error message is still logged, but the system tries to continue with reasonable assumptions. It may, however, fail at a later stage.

Methods Involving Reasoning:

Set<OWLClass> getOWLClasses(OWLNamedIndividual individual)
returns the set of lowest OWL classes in the OWL class hierarchy which contain the individual.

Set<Object> getDataPropertyValues(OWLNamedIndividual individual, OWLDataProperty property)
returns all role-fillers for data-properties as Java objects. The returned objects are either strings (in the Java class String) or elements of the wrapper classes for numbers and booleans. If the property is functional then the result contains at most one element.

Set<OWLNamedIndividual> getObjectPropertyValues(OWLNamedIndividual individual, OWLObjectProperty property)
returns all role-fillers for object-properties as OWLNamedIndividuals.

OWLClassExpression parseClassExpression(String classExpressionString)
parses a class expression string in Manchester Syntax (see App. B) to obtain an internal representation of a class expression.

Set<OWLNamedIndividual> getInstances(OWLClassExpression classExpression)
returns all OWL-instances of the given class expression.

Set<OWLNamedIndividual> getInstances(String classExpressionString)
returns the OWL-instances of the class expression.

[12]OWL API: http://owlapi.sourceforge.net/javadoc/index.html

void removeIndividual(OWLNamedIndividual individual)
creates the change axioms for removing the individual from all ontologies.

void removeAllIndividuals(OWLClass ocl)
creates the change axioms for removing all individuals in the given class.

boolean isConsistent()
returns true if the extension ontology is consistent.

7.4 The Class `J2OClassManager`

The `J2OClassManager` manages the correspondences between the Java classes and the OWL classes. The correspondences themselves are stored as instances of the class `ClassWrapper`.

7.4.1 The Class `ClassWrapper`

This class stores the correspondence between a Java class and an OWL class. The only public methods are

Class getJavaClass() returns the Java class.

OWLClass getOWLClass() returns the OWL class

The class `ClassWrapper` also stores the correlations between Java attributes and OWL properties. This information is only internally used. It is therefore not publicly exposed.

7.4.2 Public Methods of the `J2OClassManager`

Set<Class> getClasses()
returns the set of classes managed by the class manager.

Collection<ClassWrapper> getCLWs()
returns the ClassWrappers managed by the class manager.

boolean isMapped(Class jcl)
returns true if the Java class is directly mapped to an OWLClass.

ClassWrapper getCLW(Class jcl, boolean direct)
returns the ClassWrapper for the given Java class, or null if none can be found. If `direct = true` then only the direct mapping from Java classes to OWL classes is considered. If `direct = false` then the lowest superclass of jcl for which a mapping can be found is searched.

OWLClass getOWLClass(Class jcl, boolean direct)
returns the OWL class for the given Java class, or null if none can be found. If `direct = true` then only the direct mapping from Java classes to OWL classes is considered. If `direct = false` then the lowest superclass of jcl for which a mapping can be found is searched.

Set<ClassWrapper> getCLWs(OWLClass ocl)
yields the ClassWrappers which corresponds to the given OWL-class. Due to multiple inheritance in OWL the superclasses of the OWL class can form a tree (or actually a DAG). This tree is searched for the OWL classes closest to ocl which have a corresponding Java class.

Set<Class> getJavaClasses(OWLClass ocl)
yields the Java classes which corresponds to the given OWL-class.

Set<ClassWrapper> getCLWs(Set<OWLClass> ocls)
yields the ClassWrappers which corresponds to the given OWL-classes.

void activateSynchronisation(Object... classes)

void deactivateSynchronisation(Object... classes)
activates/deactivates synchronisation for all given Java classes. The given Java classes may be individual classes, arrays of classes or collections of classes.

7.5 The Individual Manager

We consider at first the auxiliary class `IndividualWrapper`.

7.5.1 The Class `IndividualWrapper`

Since the mapping between Java objects and OWL individuals need not be bijective, several Java objects may correspond to one OWL individual. Therefore there is an extra class `IndividualWrapper` which keeps all the information about the mapping between the Java objects and the OWL individuals. The instances of `IndividualWrapper` are a kind of front end for the application.

The following methods access the internal data of the individual wrapper.

OWLNamedIndividual getIndividual()
returns the OWL individual belonging to the individual wrapper.

ArrayList<Object> getObjects()
returns the objects belonging to the individual wrapper.

Object getObject(Class jcl)
returns the object among the stored objects which is an instance of the given Java class, or null if there is no such object.

Object getFirstObject()
returns the first object among the stored objects. If it is known that the mapping between Java objects and OWL individuals is bijective, there is only one such object, and therefore this method is the simplest way to get it.

reclassify(boolean forceReclassification)
reclassifies the object (see Example E.2). If `forceReclassification` is false then reclassification is done only when the object's attributes have been changed since the last reclassification. This saves resources, but may miss derivable changes to the object's attributes (e.g. computing the transitive closure of a transitive relation).

The Algorithm for Reclassifying Java Objects

Let $o_1, ..., o_n$ be the Java objects wrapped in an individual wrapper, and I be the corresponding OWL individual.

The algorithm performs the following steps to reclassify $o_1, ..., o_k$.

1. The reasoner computes the set $OC_1, ..., OC_n$ of OWL classes which are lowest in the OWL class hierarchy such that I is a member of $OC_1, ..., OC_n$ (this is sometimes called *realisation*).

2. For each OC_i in $OC_1, ..., OC_n$ the lowest superclasses in the OWL class hierarchy are determined which have corresponding Java classes. Let $O_1, ..., O_m$ be these classes and $J_1, ..., J_m$ be the corresponding Java classes. $J_1, ..., J_m$ do not contain subclasses of each other.

3. The set O of new objects to replace $o_1, ..., o_k$ is determined as follows:
   For each class J_i in $J_1, ..., J_m$:

   - If there is an object o in $o_1, ..., o_k$ whose class is a equal to J_i or a superclass of J_i then o is put into O.
   - If there is an object o in $o_1, ..., o_k$ whose class is a proper subclass of J_i then an instance o' of J_i is created, all fields of o are transferred to o' and the remaining fields are filled by transferring the corresponding properties of the OWL individual to o'.

4. For all classes J in $J_1, ..., J_m$ for which none of the above cases was applicable a new instance o is created and put into O. For those fields f in o for which an object in O exists which has a field f' with the same name and type the contents of f' is transferred to f. The remaining fields are filled by transferring the corresponding properties of the OWL individual to o.

5. In oder to ensure that the fields of the objects in O with the same name and type get pointer-equal values, before transferring the properties from OWL to Java, it is checked whether a previously filled object o' has the same field as o. In this case the reference of the field value of o' is copied to the field value of o.

Extra Attributes

In contrast to Java where the object's attributes are fixed at compile time, OWL individuals can get any number of properties at run time. The following methods allow applications to attach extra properties to the Java objects. They are not stored within the Java objects, but forwarded to the ontology. Therefore they can be taken into account by the reasoner.

boolean setObjectPropertyValue(OWLObjectProperty property, Object value)
sets the value as role-filler for the individual's property and returns true if the operation succeeded.

IndividualWrapper[] getObjectPropertyValues(OWLObjectProperty property)
returns the role-fillers for the individual's property as an array of IndividualWrappers.

boolean setDataPropertyValue(OWLDataProperty property, Object value)
sets the value as OWL attribute for the individual's property.

Set<Object> getDataPropertyValue(OWLDataProperty property)
gets the OWL attributes for the individual's property as Java objects.

These methods can also be used if the extension ontology contains an A-Box whose individuals have properties which can not be mapped to Java objects. As an example, consider a Java class `Person` and an OWL class `Person` with an OWL individual `Einstein` which has a property `Year_of_Nobel_Prize_Award`. If the Java class `Person` would have such an attribute, it would be empty for almost all instances. In this case it might be useful to keep the property at the OWL side and access it from the ontology directly.

Synchronisation

Java to OWL Synchronisation at object level can be controlled with the following methods for individual wrappers:

activateSynchronisation(boolean log)

deactivateSynchronisation(boolean log)
activates/deactivates life synchronisation at object level for all objects in the individual wrappers. If `log = true` then the activation/deactivation is logged.

synchroniseWithOWL()
performs block synchonisation for all objects in the individual wrapper.

7.5.2 The Class `J2OIndividualManager`

The individual manager servers different purposes

- it stores all the Java objects and OWL individuals for which correspondences have been established,
- it has methods for mapping new Java objects to OWL individuals,

- it has methods for mapping OWL individuals to Java objects,
- it can invoke the OWL reasoner to select individuals as instances of OWL class expressions,
- it can perform operations on collections of individual wrappers.

Accessing the Stored Java Objects and OWL Individuals

The stored Java objects and OWL individuals can not be accessed directly. The following methods return IndividualWrappers from where the Java objects and OWL individuals can be retrieved.

IndividualWrapper getIWR(Object object)
returns for a given Java-object the stored IndividualWrapper, or null if there is none.

IndividualWrapper getIWR(OWLNamedIndividual individual)
returns for a given OWL individual the stored IndividualWrapper, or null if there is none.

Mapping Java Objects to OWL Individuals and Vice Versa

The following methods can be used to establish new correspondences between Java objects and OWL individuals:

IndividualWrapper object2OWL(Object object)
creates for a Java instance of an annotated class the corresponding OWL individual (wrapped into an IndividualWrapper). For the object's annotated Java attributes whose values are Java objects which can be mapped to OWL the corresponding OWL individuals are created as well (and this process continues recursively).

IndividualWrapper individual2Java(OWLNamedIndividual individual)
creates for an OWL individual a corresponding Java object (or several of them if mulitple inheritance is necessary) and turns all OWL property values into Java attributes. If OWL property values are themselves individuals then these individuals are also mapped to Java objects, and this continues recursively.

void reclassify(boolean forceReclassification, Object... indWrapper)
reclassifies all objects in the given individual wrappers. The `indWrapper` arguments may be single individual wrappers, or arrays or collections of individual wrappers.

If `forceReclassification` is false then reclassification is done only when some of the object's attributes have been changed since the last reclassification.

The Algorithm for Mapping OWL Individuals to Java Objects

The algorithm is quite similar to the reclassification algorithm. It performs the following steps to map an OWL individual I to Java objects:

1. The reasoner computes the set $OC_1, ..., OC_n$ of OWL classes which are lowest in the OWL class hierarchy such that I is a member of $OC_1, ..., OC_n$ (this is sometimes called *realisation*).
2. For each OC_i in $OC_1, ..., OC_n$ the lowest superclasses in the OWL class hierarchy are determined which have corresponding Java classes. Let $O_1, ..., O_m$ be these classes and $J_1, ..., J_m$ be the corresponding Java classes. $J_1, ..., J_m$ do not contain subclasses of each other.
3. The classes $J_1, ..., J_m$ are instantiated to the objects $o_1, ..., o_m$.
4. For $i = 1, ..., m$ the fields of o_i with annotated getter-methods are filled with property values transferred from the OWL individual I.
5. In oder to ensure that the fields in the objects $o_1, ..., o_m$ with the same same name and type get pointer-equal values, before transferring the properties from OWL to Java, it is checked whether a previously filled object o_j has the same field as o_i. In this case the reference of the field value of o_j is copied to the field value of o_i.

Queries to the Reasoner

All OWL individuals which correspond to Java objects are in particular available to the OWL reasoner. The following methods can be used to retrieve instances of OWL class expressions as IndividualWrappers from where the Java objects can then be obtained.

ArrayList<IndividualWrapper> getInstances(String classExpression)
This method parses the class expression and asks the reasoner to compute all instances of the class expression. The instances are automatically mapped to Java objects. The `classExpression` must be in Manchester Syntax (see Sect. B).

ArrayList<IndividualWrapper> getInstances(OWLClassExpression classExpression)
This method asks the reasoner to compute all instances of the class expression. It actually uses the corresponding method of the J2OOntologyManager, which returns a set of OWL individuals. These instances are then mapped to Java objects. Notice that only OWL individuals which have already been mapped to Java can be retrieved this way.

The method `j2oOntologyManager.parseClassExpression(cle)` can be used to parse a string into an `OWLClassExpression`. This may save time if the same class expression is to be used repeatedly.

Controlling Synchronisation

The following methods activate and deactivate synchronisation at object level between Java objects and OWL individuals.

void activateSynchronisation(boolean log, Object... individualWrappers)

void deactivateSynchronisation(boolean log, Object... individualWrappers)
activates/deactivated life synchronisation for all objects in all given individual wrappers. The individual wrappers may be just single individual wrappers or arrays or collections of individual wrappers. If `log = true` then the activation/deactivation is logged.

void synchroniseWithOWL(Object... individualWrappers)
does block synchronisation for all objects in the individual wrappers. The `individualWrappers` can be single individual wrappers or arrays or collections of individual wrappers.

8 Thread Safety

The typical workflow of a Java2OWL application has been documented in Section 2. In the preparation phase, Java2OWL is involved in mapping annotated Java classes to OWL. This operation is relatively cheap, and it needs to be done only during the development of an application. Therefore it is realised as a *single thread* Java program. No thread synchronisation is necessary.

The first steps in the application phase comprise linking the Java classes to the extension ontology. The classes and methods involved are the same as the ones involved in the preparation phase. Therefore this step is also realised as a single thread application. The application program should therefore spawn threads only *after* the linking step. If threads try to access the extension ontology before the linking is finished, *anything can happen.*

The following types of interaction with the extension ontology are possible during the main application:

- creating Java objects and mapping them to OWL;
- mapping OWL individuals to Java objects;
- changing attributes of Java objects and immediately forwarding the change to OWL;
- block synchronisation of Java objects with OWL (if synchronisation has been turned off for a while);

- reclassification of Java objects;
- querying the extension ontology to retrieve instances of OWL class expressions.

Not all of these operations have immediate effects, but the information is stored in different caches. Some operations involve external OWL reasoners, maybe even written in another programming language. Therefore making them thread safe is not easy. Moreover, one should keep the overhead for thread safety as small as possible. The goal is of course to make all these operations thread safe with minimal costs. This has been tried as best as possible. However, since thread safety is extremely difficult to test, no guarantee can be given that all cases have been covered.

9 Performance of the System

A thorough performance analysis of the Java2OWL system is not really possible. The input are Java programs and OWL ontologies, and there is no clear output which can be measured. Moreover, a significant part of the system is the OWL reasoner, and this is not under control of the Java2OWL system. The complexity of the OWL reasoning tasks depends on the OWL constructs used in the application. The complexity classes range from polynomial to undecidable. Typical examples are in the PSPACE range, which means that heuristics have an important influence on the behaviour of the reasoners.

The few experiments whose results are listed below might give a rough impression on the performance of the various parts of the system. They have been performed with a Java class `Student`, with attributes `name` and `semester` and a subclass `Freshman` which are students in the first semester (see Example E.2). The classes have been translated to OWL and then a number of measurements have been performed.

In all experiments a sequence of lists of `Student` instances have been created, and for each list a certain operation has been performed. The time it took to perform the operation on the list has been measured, and divided by the length of the list. The resulting times in milliseconds indicate the time it took to perform a single operation. The lengths of the lists are 1000, 2000, ..., 10000. The measurements were done with a Dell notebook with a 2.2 GHz dual core processor.

Java to OWL axioms: Each `Student` has been translated to OWL axioms, but the axioms are not yet integrated into the ontology. The times in milliseconds per operation are:

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
ms	0.17	0.19	0.14	0.04	0.009	0.01	0.008	0.032	0.0049	0.0055

It is not clear why the times are so different. One effect could be the Just-in-Time compilation of Java which after a while decides to compile the Java methods into native machine code.

Java to OWL indivdiduals: In this experiment the translated Student instances are integrated as OWL individuals into the OWL ontology. The times in milliseconds per operation are:

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
ms	0.43	0.39	0.21	0.16	0.048	0.016	0.028	0.011	0.08	0.03

Java to OWL individuals with consistency test: In this experiment the translated Student instances are integrated as OWL individuals into the OWL ontology, and each time a consistency test is performed by the OWL reasoner. All three reasoners are tried. The times in milliseconds per operation are:

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
FaCT++	0.556	0.14	0.122	0.046	0.065	0.013	0.024	0.07	0.026	0.09
Pellet	0.534	0.13	0.135	0.076	0.027	0.064	0.025	0.077	0.025	0.1]
HermiT	0.576	0.17	0.107	0.120	0.111	0.037	0.012	0.031	0.013	0.09

Java-OWL synchronisation: This time the `Student` instances which have been mapped to OWL get their semester changed to 1, and this is forwarded to the ontology. It does not involve OWL reasoning. The times in milliseconds per operation are:

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
ms	0.39	0.07	0.037	0.012	0.07	0.0077	0.023	0.11	0.0075	0.0075

Querying OWL: In this experiment we take the `Student` instances whose semester had been changed to 1, and map them to OWL individuals. Afterwards the OWL reasoner is asked to retrieve all instances of the OWL `Freshman` class. All three reasoners are tried. The operation is much more expensive than the previously investigated operations. Therefore only much smaller lists of `Student` instances are tried. The times in milliseconds per operation are:

size	100	200	300	400	500	600	700	800	900	1000
FaCT++	0.93	0.455	0.167	0.187	0.168	0.167	0.17	0.22	0.17	0.18
Pellet	20.66	7.57	6.81	8.02	11.15	12.3	13.5	15.3	18.9	22.4
HermiT	15.39	5.14	6.72	10.0	13.22	17.12	21.14	27.8	32.5	37.6

For this class of examples FaCT++ is about 20 times faster than the other reasoners. Since OWL reasoning is a heuristically controlled search there may well be other classes of examples where the other reasoners are faster.

OWL to Java: In this experiment the `Student` instances whose semester had been changed to 1, and which had been mapped to OWL individuals are deleted, and afterwards reconstructed as `Freshman` instances form the OWL individuals. The times in milliseconds below indicate how long it took to turn an OWL individual into a Java object.

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
ms	0.175	0.119	0.115	0.13	0.134	0.222	0.176	0.198	0.188	0.238

Reclassification: In this experiment the `Student` instances whose semester had been changed to 1 are reclassified to `Freshman` instances. All three reasoners are tested. The times in milliseconds per operation are:

size	100	200	300	400	500	600	700	800	900	1000
FaCT++	1.23	1.11	0.44	0.43	0.38	0.29	0.38	0.29	0.24	0.3
HermiT	18.64	5.48	6.66	9.49	12.56	16.22	20.2	25.73	30.06	35.08
Pellet	22.13	7.58	10.45	13.13	15.24	16.36	18.96	20.05	25.76	29.24

Again, FaCT++ is much faster than the other reasoners.

The times vary quite a lot with the different lengths of the lists. An important observation is, however, that the times per operation do not depend much on the size of the data. Sometimes the operations become even faster with increased amounts of data.

10 Summary

This report describes the Java2OWL library version 1.1. The program itself together with the documentation can be obtained from http://www.pms.ifi.lmu.de/Java2OWL. The following files can be downloaded

- Java2OWL.jar contains the compiled code of the library. The J2OCompiler within Java2OWL.jar is the class with the main function. Therefore the compiler can be called by just executing `java -jar Java2OWL.jar ...`,
- J2OSynchronizerAgent.jar contains the compiled Java agent with the synchroniser for the class loader,
- Java2OWL.tgz contains the entire NetBeans project with source code, examples and documentation,
- Java2OWL_doc.tgz contains the Javadoc generated files,
- J2OReport.pdf is this report as pdf file.

The system has not yet been tested in a real application. It is planned to use it for a university wide information system. It is quite clear that real applications will require further changes to the library. Suggestions for improvements may be E-mailed to ohlbach@lmu.de.

10.1 Synchronising OWL with Other Programming Languages

There are quite a number of other object oriented programming languages where a synchronisation with Java would make sense. Therefore the question is whether the ideas and techniques of Java2OWL can be used for these other programming languages as well. Java2OWL in its current version depends on a number of special features of Java

- the OWL API for Java. At the time being there seems to be no similar OWL API for other programming languages. There are APIs for XML and RDF in other programming languages. They could provide the basis for an OWL API, but a lot of work still has to be done.

- the possibility to annotate classes and methods and to read the annotations programmatically. This seems to be a unique feature in Java. In other languages, for example C and C++ there are *compiler directives* or *preprocessor directives* which are processed by a preprocessor of the compiler. This would be the right tool to achieve something similar to the Java annotations. It is, however, very unlikely that the standard preprocessors could be used as they are. It might therefore be necessary to write an extra preprocessor.

  An alternative to annotations in the source code is of course to put the extra information into extra files, typically XML files. This was the way many Java applications operated before the introduction of annotations. Besides the extra work to write the extra files, there is always the problem to keep the source files and the extra files consistent.

- Java reflection. In the Java2OWL library Java reflection is heavily used to analyse the Java program and to generate the extension ontology. There are only a few other languages with similar features, for example LISP. If reflection is not available there are two alternatives:

  1. the source code is parsed and the information is extracted from the source code directly.
  2. the programmer puts the information about the structure of the program, in our case the getter and setter methods and the class hierarchy, into extra files. This is error prone and means extra work for the programmer.

- byte code engineering. In Java2OWL the synchroniser code is automatically injected into the compiled class files. This could of course also be done manually by the programmer; again, it means extra work and is error prone. More convenient would be a specialised preprocessor for changing the source code before compiling it.

To summerise, the ideas of Java2OWL can also be realised in other object oriented programming languages, provided there is a good OWL API. For the concrete realisation there are two possibilities:

- A specialised preprocessor for the compiler extracts the necessary information from the source code, generates the extension ontology and transforms the source code to include the synchroniser code.

- If this is not possible the programmer can put the information into extra files and insert the synchroniser code manually into the programs.

When a compiled application program which uses Java2OWL gets started, the first thing it has to do is to link the Java class hierarchy with the OWL hierarchy in the extension ontology. In this step Java2OWL uses reflection again to inspect the Java classes. Alternatively a preprocessor could put the corresponding information extracted from the source files into extra files. At run

time a linker could take the information from there to build up the data structures for correlating Java with OWL.

The impression is that Java has some quite unique features which allow one to build such a system in a way which requires minimal extra work by the application programmer, and which is therefore much less error prone than the alternatives in other programming languages.

11 Appendix

A Installing the Libraries and the Reasoners

In order to get the Java2OWL library to work you have to

- download the files Java2OWL.jar and J2OSynchronizerAgent.jar,
- download the latest version of the OWL API[13] [5], unzip it to get the file owlapi-bin.jar.
- install at least one of the supported reasoners. Notice that HermiT is the default reasoner. If it is not installed one must change the reasoner (by calling `setReasoner` in the `J2OOntologyManager`) before using the system.

The `.jar` files can either be put into the Java extension directory, for example in /usr/lib/jvm/java-6-sun-1.6.0.24/jre/lib/ext, or in a directory contained in the Java CLASSPATH.

The current version of the Java2OWL library supports three reasoners, HermiT [2], Pellet [7] and FaCT++ [8]. They can be installed as follows:

HermiT: download HermiT.zip[14], unzip it and put the file HermiT.jar into a place where Java can find it;

Pellet: download the latest version of Pellet[15], unzip the zip archive. It is recommended to locate the pellet.jar and aterm-java-x.x.jar libraries in the lib folder and to place these two files into a directory where Java can find it. Experience has shown that other .jar files contained in the zip archive are also necessary. The best is to put all of them into a place where Java can find them.

FaCT++: FaCT++ is written in C++. Therefore you must download the FaCT++ OWLAPI[16] as well as the FaCT++ precompiled binaries[17] for your platform (OS X, Linux or Windows). Place the FaCT++OWLAPI-vx.x.x.jar file in a place where Java can find it. Unzip the native libraries and ensure that the directory containing the FaCTPlusPlusJNI library is in your java.library.path. For example if the libraries reside in /usr/lib/ then you should add: -Djava.library.path="/usr/lib" to the arguments for your Java application.

[13] OWL API: http://sourceforge.net/projects/owlapi/
[14] HermiT.zip: http://hermit-reasoner.com/
[15] Pellet: http://clarkparsia.com/pellet/download
[16] FaCT++ OWLAPI: http://code.google.com/p/factplusplus/list
[17] FaCT++ precompiled binaries: http://code.google.com/p/factplusplus/list

B Manchester OWL Syntax

The Manchester OWL Syntax is used for formulating class expressions in queries to the reasoner. The complete specification can be found in http://www.w3.org/2007/OWL/wiki/ManchesterSyntax. This section contains a short overview which is taken from http://www.co-ode.org/resources/reference/manchester_syntax/

Boolean Class Constructors

Boolean class constructor symbols have simply been replaced by their English language counterparts.

OWL	DL Symbol	Manchester OWL Syntax Keyword	Example
intersectionOf	$\sqcap$	and	Doctor and Female
unionOf	$\sqcup$	or	Man or Woman
complementOf	$\neg$	not	not Child

Restrictions

Restrictions use an infix syntax rather than a prefix syntax. The Description Logic symbols that were previously used to indicate the type of restriction have been replaced with English language keywords.

OWL	DL Symbol	Manchester OWL Syntax Keyword	Example
someValuesFrom	$\exists$	some	hasChild some Man
allValuesFrom	$\forall$	only	hasSibling only Woman
hasValue	$\ni$	value	hasCountryOfOrigin value England
minCardinality	$\geq$	min	hasChild min 3
cardinality	$=$	exactly	hasChild exactly 3
maxCardinality	$\leq$	max	hasChild max 3

Complex Class Expressions

Complex class expressions can be constructed using the above boolean constructors and restrictions. For example,

Person and hasChild some (Person and (hasChild only Man) and (hasChild some Person))
describes the set of people who have at least one child that has some children that are only men (i.e. grandparents that only have grandsons). Note that brackets should be used to disambiguate the meaning of the expression.

Data Values and Data Types (OWL1.1)

Data values may be untyped or typed (eg int, boolean, float etc). The types available will depend on tool support, but will include those specified in the XSD recommendation.

Constants can be expressed without type by just enclosing them in double quotes, or with type like in 'hasAge value "21"^^long'.

Usage of these data types in more general expressions is possible through their shortened name, for example 'hasAge some int'

Several additional XSD facets can also be used to create new data types: 'Person and hasAge some int[>= 65]'

Multiple facets can also be used. For example, when wishing to express numeric ranges: 'Person and hasAge some int[>= 18, <= 30]'

XSD facet	Meaning
< x, <= x	less than, less than or equal to x
> x, >= x	greater than, greater than or equal to x
length x	for strings, the number of characters must be equal to x
maxLength x	for strings, the number of characters must be less than or equal to x
minLength x	for strings, the number of characters must be greater than or equal to x
pattern regexp	the lexical representation of the value must match the regular expression
totalDigits x	number can be expressed in x characters
fractionDigits x	part of the number to the right of the decimal place can be expressed in x characters

Warning:
The OWL-API removes an OWLDataProperty or an OWLObjectProperty completely if it is no longer used. A class expression like "hasFriend some Person" would therefore *raise a ParserException* if there are no friends any more in the ontology.
A test like `getFirstOWLDataProperty("hasFriend") == null` can be used to check this situation.

C Data Logs

The Java2OWL library hides many important operations from the application developer. In order to help him understand what is going on, and to debug his application, extensive data logs are provided. Since the logs can become very long, there is a special class `Logging` with mechanisms to selectively turn data logging on and off. `Logging` is an interface to the Java `Logger` class. A J2OManager automatically creates a `Logging` object and forwards it to the other components of Java2OWL package. In order to configure it, one can get the installed logger by calling the method `public Logging getLogger()` for the J2OManager, and then call various configuration methods.

The possibilities to configure the `Logging` object are described in the subsequent paragraphs.

Handler

The `Logger` forwards its output to a *handler*. The handler is responsible for forwarding the messages to the final destination. The default handler just prints it to `System.out`. This can be changed by the `setHandler` method. It comes in two forms:

setHandler(Logging.Destination where, String filename)
The `Logging.Destination` can be `OUT, ERR, BUFFER`, or `FILE`. `OUT` sends the logs to `System.out`. `ERR` sends the logs to `System.err`. `BUFFER` sends the logs to an in-memory buffer. `FILE` sends the logs to a file with the given `filename`. If `BUFFER` is chosen then the logs can be accessed with the methods `String getContents()` and `boolean hasContents()`.

setHandler(Handler newHandler, boolean closable)
In this case a user defined handler (see `java.util.logging`) can be created and installed. The flag `closable` indicates that the handler must be closed at the end. (System.out and System.err, for example, should *not* be closed.)

A handler can be changed at any time. The old handler is closed and the new handler is opened.

Some handler maintain internal buffers for the messages. This might cause confusion because messages may not appear immediately at the output device. The internal buffer can be *flushed* by calling the method `flush()`. Error and warning messages, however, are immediately flushed.

Messages with Semantics

In order to distinguish different classes of messages, and to be able to turn them on and off, each message is labelled with a tag from the `Labelling.Semantics` enum. The different tags are:

`INFO`	General information
`CONTROLFLOW`	entering, exiting, installation messages
`ERROR`	error and warning messages
`OCC`	OWL class creation
`OPC`	OWL property creation
`OIC`	OWL individual creation
`JOC`	Java-Object creation (from an OWL individual)
`DC`	Data Change.

By default logging of all message types is enabled. One can enable logging of certain message types only by calling `void activate(Logging.Semantics... semantics)`. Error and warning messages, however, are always enabled. For example,

```
logger.activate(Logging.Semantics.OCC,Logging.Semantics.OIC);
```

enables only OWL class creation and OWL individual creation messages (and error and warning messages).

Enabling or disabling all message types at once, *including* error and warning messages, is possible by calling `void activate(boolean active)`.

Enabling or disabling all message types at once, *excluding* error and warning messages, is possible by calling `void activateInfo(boolean active);`. `logger.activate(false);` disables all but error and warning messages at a very early state. This causes the least overhead by the logger. `logger.activate(true);` enables all message types, except those excluded by calling `void activate(Logging.Semantics... semantics)`.

Origin

The error and warning messages generated by the logger include a hint to the origin of the error or warning. The origin is indicated by one of the key words:

`USAGE`	something wrong with the way the application uses Java2OWL
`JAVA`	Java programming error
`ONTOLOGY`	something wrong with the ontology
`INDIVIDUAL`	something wrong with the OWL individuals
`REASONER`	something wrong with the reasoner
`UNCLEAR`	no idea where the problem came from.

In practice, however, it is not always clear what exactly went wrong. Therefore these hints may not always be precise.

Debug Mode

The logger can be put into debug mode by calling `setDebugMode(boolean flag)`, either separately for the logger, or globally for the J2OManager. In debug mode two things are different to normal mode:

1. all messages are immediately flushed.
2. an error message causes the system immediately to stop (by calling `System.exit(1)`). This may help detecting errors very early. Errors in normal mode usually cause an exception to be thrown. The exception handler may be able to circumvent the error, but this should not be a permanent solution.

Remarks

The `Logging` class is basically independent of the Java2OWL system. It can therefore in principle be used by other systems as well. Log messages are created by one of the following `public void` methods:

entering(Object... objects)
indicates entering a method and prints the objects.

exiting(Object... objects)
indicates exiting a method and prints the objects.

installation(Object object)
indicates installation of a module and prints the object.

error(String message, Origin origin,Object... objects)
generates an error message. In debug mode this method causes the system to stop.

warning(String message, Origin origin,Object... objects)
generates a warning message.

event(Semantics semantics, Object... objects)
generates an event message with a specific semantics.

correlation(Semantics semantics, Object... objects)
generates a message informing that a new correlation between the objects has been established.

The logger uses the `toString()` method to print the objects.

D Data Types

Java distinguishes primitive data types and object data types. The primitive data types are `boolean, byte, short, int, long, float` and `double`. A further distinguished non-primitive data type is `String`. All these data types have correspondences in OWL. Therefore it is no problem to map Java objects with attributes of these types to OWL. OWL, however, has a much richer set of built in data types. Most of them are inherited from XML-Schema[18]. It is therefore not so straight forward to map OWL individuals with their properties to Java. As a compromise, OWL data types are mapped in the following way to Java data types:

OWL data type	Java data type
OWL_RATIONAL	double
OWL_REAL	double
RDF_PLAIN_LITERAL	String
RDF_XML_LITERAL	String
XSD_ANY_URI	String
XSD_BASE_64_BINARY	int
XSD_BOOLEAN	boolean
XSD_BYTE	byte
XSD_DATE_TIME	String
XSD_DATE_TIME_STAMP	String
XSD_DECIMAL	int
XSD_DOUBLE	double
XSD_FLOAT	float
XSD_HEX_BINARY	int
XSD_INT	int
XSD_INTEGER	int
XSD_LANGUAGE	String
XSD_LONG	long
XSD_NAME	String
XSD_NCNAME	String
XSD_NEGATIVE_INTEGER	int
XSD_NMTOKEN	String
XSD_NON_NEGATIVE_INTEGER	int
XSD_NON_POSITIVE_INTEGER	int
XSD_NORMALIZED_STRING	String
XSD_POSITIVE_INTEGER	int
XSD_SHORT	short
XSD_STRING	String
XSD_TOKEN	String
XSD_UNSIGNED_BYTE	byte
XSD_UNSIGNED_INT	int
XSD_UNSIGNED_LONG	long
XSD_UNSIGNED_SHORT	short

[18] XML-Schema: http://www.w3.org/TR/xmlschema-2/

E Examples

The examples in this section illustrate already advanced features of the system.

E.1 Multiple Inheritance Example

The example in this section illustrates the treatment of the OWL multi-inheritance in the Java2OWL framework. Instances of an OWL class which is a subclass of more than one OWL class and which has no corresponding Java class are mapped to several Java objects. We use the notorious Ship/Surface-Vehicle/Amphibious-Vehicle example. At the Java side we define classes `TestVehicle`, `TestShip` and `TestSurfaceVehicle`. The example has no background ontology, but generates an extension ontology `Vehicle.owl`. This ontology is then extended by adding a common subclass `AmphibiousVehicle` and some individuals. The new ontology `AmphibiousVehicle.owl` is then loaded into a Java program and the OWL individuals are mapped to Java objects. The class `TestVehicle` has attributes `name`, `owners` (as `ArrayList<String>`) and `predecessors` (as `TestVehicle[]`). The attributes are inherited to the subclasses `TestShip` and `TestSurfaceVehicle`. The class `TestShip` has a further attribute `knots` (as `float`). The class `TestSurfaceVehicle` has a further attribute `enginePower` (as `double`).

The `main`-method of the `TestVehicle` class calls the J2OCompiler for translating the classes to OWL.

```
@J2OWLClass(name = "Vehicle")
public class TestVehicle {

    private String name;
    @J2OWLProperty(name = "hasName", setter="setName", total=true)
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}

    private ArrayList<String> owners = new ArrayList<String>();
    @J2OWLProperty(name = "hasOwner", adder="addOwner",
          remover = "removeOwner")
    public ArrayList<String> getOwners() {return owners;}

    public void addOwner(String owner) {owners.add(owner);}
    public void removeOwner(String owner) {owners.remove(owner);}

    private TestVehicle[] predecessors;
    @J2OWLProperty(name = "hasPredecessor", adder="addPredecessor")
    public TestVehicle[] getPredecessors() {return predecessors;}

    public void addPredecessor(TestVehicle vehicle) {
   if(predecessors == null)
       predecessors = new TestVehicle[]{vehicle};
   else {predecessors =
      Arrays.copyOf(predecessors, predecessors.length+1);
         predecessors[predecessors.length-1] = vehicle;}}


   @J2OWLClass(name = "Ship")
   public static class TestShip extends TestVehicle {

   private float knots;
   @J2OWLProperty(name = "speed", setter="setKnots")
```

```
public float getKnots() {return knots;}
public void setKnots(float knots) {this.knots = knots;}}


@J2OWLClass(name = "SurfaceVehicle")
public static class TestSurfaceVehicle extends TestVehicle {

private double enginePower;
public double getEnginePower() {return enginePower;}
public void setEnginePower(double enginePower) {
    this.enginePower = enginePower;}}


 public static void main(String[] args)  {
args = new String[]{
   "build/test/classes/TestPackage/TestVehicle.class",
   "exoi", "http://www.lmu.de/Vehicle.owl",
   "exof", "build/test/classes/TestPackage/Vehicle.owl",
   "sync","true","debug","true"};
J2OCompiler.main(args);}}
```

The generated ontology `Vehicle.owl` can be loaded into the OWL editor Protégé[19] [4] and exported as LaTeX file:

Classes

SurfaceVehicle

SurfaceVehicle $\sqsubseteq$ Vehicle

Ship

Ship $\sqsubseteq$ Vehicle

Vehicle

Vehicle $\sqsubseteq \exists$ hasName

Object properties

hasPredecessor

$\top \sqsubseteq \forall$ hasPredecessor Vehicle

Data properties

enginePower

$\top \sqsubseteq \leq 1$ enginePower

[19] Protégé: http://protege.stanford.edu/

hasName

$\top \sqsubseteq \leq 1$ hasName

hasOwner

speed

$\top \sqsubseteq \leq 1$ speed

Now we extend the ontology manually by adding a common subclass `AmphibiuosVehicle` and some individuals. The LaTeX dump generated by of Protégé is[20]:

Classes

AmphibiuosVehicle

AmphibiuosVehicle $\sqsubseteq$ Ship
AmphibiuosVehicle $\sqsubseteq$ SurfaceVehicle

SurfaceVehicle

Ship

Vehicle

Object properties

hasPredecessor

Data properties

enginePower

hasName

hasOwner

speed

Individuals

KingGeorge

KingGeorge : SurfaceVehicle
    hasName: "King George"

QueenMary

QueenMary : AmphibiuosVehicle
    hasName: "Queen Victoria"
    hasOwner: ("QueenElizabeth","PrinceCharles")
    enginePower: 200
    speed: 40.0f
    hasPredecessor(KingGeorge,QueenVictoria)

[20]Protégé's LaTeX dump is still incomplete. Therefore the text had to be modified slightly.

QueenVictoria

QueenVictoria : Vehicle
hasName: "Queen Victoria"

The ontology `AmphibiousVehicle.owl` can now be loaded into a Java programm and the OWL individuals can be imported into Java. As an example we list the JUnit code for testing the ontology.

```
static J2OManager manager;
static J2OOntologyManager omg;
static J2OClassManager clm;
static J2OIndividualManager img;

 // This is the initialisation code.
@BeforeClass
public static void setUpClass() throws Exception {
    manager = new J2OManager("TestManager",J2OManager.Status.RUN,true);
    manager.getLogger().activateInfo(false);

    AutoIRIMapper iriMapper =
   new AutoIRIMapper( new File("build/test/classes/TestPackage"),true);
    omg = manager.getOntologyManager();
    omg.addIRIMapper(iriMapper);

    clm = manager.getClassManager();
    String name = "build/test/classes/TestPackage/AmphibiousVehicle.owl";
    OWLOntology extOntology = omg.loadExtensionOntology(name);

    J2OCompiler comp = manager.getCompiler();
    comp.class2OWL(TestVehicle.class);
      //linking the Java classes to the OWL classes.
    comp.finishTranslation();

    img = manager.getIndvidualManager();}
```

Now the ontology `AmphibiousVehicle.owl` is loaded, linked, and ready to be used. The following statements illustrate how the ontology can be used.

```
@Test
public void testIndividual2Java() throws ParserException {

    Set<OWLNamedIndividual> individuals = omg.getInstances("Ship");
    OWLNamedIndividual QueenMary = individuals.iterator().next();
    //Notice, QueenMary's OWL class is AmphibousVehicle

    IndividualWrapper qm = img.individual2Java(QueenMary);

    //Now we access the 'ship aspect' of the AmphibousVehicle
    TestShip ship = (TestShip)qm.getObject(TestShip.class);
    //Ship is the lowest class above AmphibousVehicle
    //with a correspondence in Java.
```

```
  assertEquals("QueenMary",ship.getName());
  assertEquals(40.0,ship.getKnots(),0.0);
  assertEquals(2,ship.getOwners().size());
  assertEquals("[QueenElizabeth, PrinceCharles]",
     ship.getOwners().toString());

  TestVehicle[] predecessors = ship.getPredecessors();
  TestVehicle pd1 = predecessors[0];
  assertEquals("King George",pd1.getName());
  assertEquals("TestSurfaceVehicle",pd1.getClass().getSimpleName());
  TestVehicle pd2 = predecessors[1];
  assertEquals("Queen Victoria",pd2.getName());
  assertEquals("TestVehicle",pd2.getClass().getSimpleName());

  //Now we access the 'surface-vehicle aspect' of the AmphibousVehicle
  TestSurfaceVehicle surfaceVehicle =
 (TestSurfaceVehicle)qm.getObject(TestSurfaceVehicle.class);
  assertEquals("QueenMary",surfaceVehicle.getName());
  assertEquals(2000,surfaceVehicle.getEnginePower(),0.0);
  assertEquals("[QueenElizabeth, PrinceCharles]",
     surfaceVehicle.getOwners().toString());

  // Notice that the common attributes of the
  // 'ship aspect' and the 'surface-vehicle aspect' are pointer equal.
  assertTrue(ship.getOwners()       == surfaceVehicle.getOwners());
  assertTrue(ship.getName()     == surfaceVehicle.getName());
  assertTrue(ship.getPredecessors() == surfaceVehicle.getPredecessors());}
```

E.2 Reclassification Example

The next example illustrates the reclassification feature of the Java2OWL library. We define a class `SimpleStudent` with an attribute `semester` and a subclass `Freshman` of `SimpleStudent` where the semester is set to 1. The `Freshman` class has an extra attribute `Mentor`. A `SimpleStudent` instance with `semester = 2` whose semester is changed to 1 can be reclassified to become a `Freshman` instance. The Java class definitions are:

```
@J2OWLClass
public class SimpleStudent {

    private int semester;

    @J2OWLProperty(name = "semester", setter="setSemester")
    public int getSemester() {return semester;}
    public void setSemester(int semester) {this.semester = semester;}

    public SimpleStudent(){}
    public SimpleStudent(int semester) {this.semester = semester;}

    @J2OWLClass(EquivalentClass = "semester value 1")
    public static class Freshman extends SimpleStudent {
   @Override
   public int getSemester() {return 1;}

   String mentor;
   @J2OWLProperty(name = "hasMentor", setter="setMentor")
   public String getMentor() {return mentor;}
   public void setMentor(String mentor) {this.mentor = mentor;} }

    public static void main(String[] args) {
   args = new String[]{
      "build/test/classes/TestPackage/SimpleStudent.class",
      "exoi", "http://www.lmu.de/SimpleStudent.owl",
      "exof", "build/test/classes/TestPackage/SimpleStudent.owl",
      "sync","true","debug","true"};
   J2OCompiler.main(args); }}
```

Notice the annotation `EquivalentClass = "semester value 1"` for the class `Freshman`. Together with `class Freshman extends SimpleStudent` this causes the OWL class `Freshman` to become equivalent with `SimpleStudent and semester value 1`. Therefore a `SimpleStudent` OWL individual whose semester is set to 1 becomes an instance of the OWL class `Freshman`.

The following code snippet shows how the reclassification works. After some preparation steps we generate a Java `SimpleStudent` object `James` in the 4th semester and map it to the OWL side. Then we give it a property `hasMentor` = "Billy" by attaching it at the corresponding OWL individual. Afterwards the semester is set to 1, and the object is reclassified. As expected, the new object is an instance of the Java class `Freshman`, and we can call the `getMentor()` method for the new `Freshman` instance to retrieve the mentor "Billy".

```
J2OManager manager = new J2OManager("Manager",J2OManager.Status.RUN,true);
J2OOntologyManager omg = manager.getOntologyManager();
String name = "build/test/classes/TestPackage/SimpleStudent.owl";
OWLOntology extOntology = omg.loadExtensionOntology(name);
```

```
J2OCompiler comp = manager.getCompiler();
comp.class2OWL(SimpleStudent.class);
comp.finishTranslation();
J2OIndividualManager img = manager.getIndvidualManager();
img.activateSynchForClass(SimpleStudent.class); // do not forget this!

//The preparation is finished now.

SimpleStudent James = new SimpleStudent(4);
IndividualWrapper james = img.object2OWL(James);

OWLDataProperty hasMentor = omg.getOWLDataProperty("hasMentor");

James.setSemester(1);
james.setDataPropertyValue(hasMentor, "Billy");

james.reclassify();

//Now we test the reclassification
Object student = james.getFirstObject();
assertTrue(student instanceof Freshman);
Freshman freshman = (Freshman)student);
assertEquals("James",freshman.getName());
assertEquals("Billy",freshman.getMentor());
//Notice that the 'hasMentor' property of the OWL individual
//has been transferred to the new Freshman object.

String mentor = (String)(
      james.getDataPropertyValues(hasMentor).iterator().next());
//We can still access the 'hasMentor' property.
assertEquals("Billy",mentor);
assertTrue(freshman.getMentor() == mentor);
//The Java attribute and the OWL property are pointer equal.
```

Now the example gets extended a bit further. We define another class `Teacher` who teaches some students. The generated OWL class is made equivalent to `"teaches some SimpleStudent"`, such that every object which teaches some students becomes automatically an instance of the OWL class `Teacher`. The Java class `Teacher` itself is not related at all with `SimpleStudent` or `Freshman`.

```
@J2OWLClass(EquivalentClass = "teaches some SimpleStudent")
public class Teacher {
    Set<SimpleStudent> students = new HashSet<SimpleStudent>();

    @J2OWLProperty(name = "teaches", adder="addStudent")
    public Set<SimpleStudent> getStudents() {return students;}
    public void addStudent(SimpleStudent student) {
       students.add(student);}}
```

The tests are done in the same setting as above. There is the `SimpleStudent` James who became a `Freshman`. We ask him to teach *himself* by adding James as OWLObjectProperty `teaches`. Now he should still be a `Freshman`, but in addition he should also be a `Teacher`, and, as a Java object `Teacher`, he should have himself as a student.

```
OWLObjectProperty teaches = omg.getOWLObjectProperty("teaches");

james.setObjectPropertyValue(teaches, James);
james.reclassify(); // Now he should be a Freshman and a Teacher

Teacher teacher = (Teacher)james.getObject(Teacher.class);
assertTrue(freshman == teacher.getStudents().iterator().next());
assertTrue(freshman == james.getObject(Freshman.class));
```

Notice that after reclassification there are two Java objects for the OWL individual James, one in the `Freshman` class and one in the `Teacher` class. Only the `Freshman` instance, however, was inserted into the `students` attribute of the `Teacher` object. The `Teacher` instance has the wrong class to be inserted into the `students` attribute.

E.3 Reclassification Example with Relations with Properties

This example illustrates reclassification of objects with a reflexive-transitive relational attribute. The reflexive-transitive closure is computed by the reasoner and mapped back to the Java side. The Java class is

```
@J2OWLClass(name = "Item")
public class TestItem {
    Set<TestItem> components = new HashSet<TestItem>();

    @J2OWLProperty(name = "hasPart", adder = "addItem", clearer = "clearItems",
       transitive = true, reflexive = true)
    public Set<TestItem>  getItems() {return components;}

    public void addItem(TestItem item) {components.add(item);}

    public void clearItems() {components.clear();}}
```

The following code snippet demonstrates the effect:

```
TestItem item1 = new TestItem();
TestItem item2 = new TestItem();
IndividualWrapper it1 = img.object2OWL(item1);
IndividualWrapper it2 = img.object2OWL(item2);

item1.addItem(item2);
TestItem item3 = new TestItem();
IndividualWrapper it3 = img.object2OWL(item3);
item2.addItem(item3);

assertEquals(1,item1.getItems().size());
it1.reclassify(true);

assertEquals(3,item1.getItems().size());
```

After reclassification, `item1.getItems()` returns in fact all three items, and this is the reflexive-transitive closure of the `hasPart` relation.

E.4 Example with Map Types

The next example illustrates the treatment of Map-type attributes. Some of the accessor methods below, for example `setTexts`, have an argument of type `Object` instead of the expected generic type, in this case `HashMap<Locale,String>`. This circumvents a bug in the BCEL-5.2 library. The class files of compiled Java classes store information about generic local variables in a LocalVariableTypeTable. If BCEL-5.2 manipulates methods with such a LocalVariableTypeTable the resulting binary class file is no longer syntactically correct, even if the LocalVariableTypeTable is not touched at all. This bug is fixed in BCEL-5.3, but the version 5.3 is not yet released and still has other problems. The way to get around these problems is to avoid local variables with generic types in the accessor methods[21].

```
@J2OWLClass(name="Mapper",synchronise=true)
public class TestMapper {
    // First we test functional dataProperties

    HashMap<Locale,String> texts = new HashMap<Locale,String>();

    @J2OWLProperty(name = "hasText", setter = "setTexts",
      adder="addText,addTextLogged", remover="removeText,removeTextLogged",
      clearer="clearText", getKey="getKey")
    public HashMap<Locale,String> getTexts() {return texts;}

    public void setTexts(Object texts) {
        assert(texts instanceof HashMap);
        this.texts = (HashMap<Locale,String>)texts;}

    public String getText(Locale locale) {return texts.get(locale);}

    public String addText(Locale locale, String string) {...}
    public void addTextLogged(Locale locale,String string,boolean log) {...}

    public void removeText(Locale locale) {texts.remove(locale);}

    public void removeTextLogged(Locale locale, boolean log) {...}

    public void clearText() {texts.clear();}

    public Locale getKey(String s) {return new Locale(s);}

    // Now we test relational objectProperties of type Map

    HashMap<String,Set<TestMapper>> friends =
        new HashMap<String,Set<TestMapper>>();

    @J2OWLProperty(name = "hasFriends", setter = "setFriends",
        adder="addFriends,addFriend",
        remover="removeFriends,removeFriend", clearer="clearFriends")
    public HashMap<String,Set<TestMapper>> getFriends() {return friends;}

    public void setFriends(Object friends) {
        assert(friends instanceof HashMap);
```

[21] BCEL-5.3 has changed the treatment of RuntimeVisibleAttributes. Therefore the class J2OSynchroniser needs to be adapted to BCEL 5.3 once it is released.

```
        this.friends = (HashMap<String,Set<TestMapper>>)friends;}

    public Set<TestMapper> getFriends(String key) {return friends.get(key);}

    public void addFriends(String key, Object f1) {
    assert(f1 instanceof Set);
    friends.put(key,(Set<TestMapper>)f1);}

    public void addFriend(String key, TestMapper f) {...}

    public void removeFriends(String key) {friends.remove(key);}

    public void removeFriend(String key,TestMapper f) {...}

    public void clearFriends() {friends.clear();}

    // Now we test relation data properies of type Map

    HashMap<String,String[]> owners = new HashMap<String,String[]>();

    @J2OWLProperty(name = "hasOwner", setter = "setOwners",
        adder="addOwner,addOwners",
        remover="removeOwner,removeOwners", clearer="clearOwners")
    public HashMap<String,String[]> getOwners() {return owners;}

    public void setOwners(Object owners) {
        assert(owners instanceof Map);
        this.owners = (HashMap<String,String[]>)owners;}

    public void addOwner(String key, String owner) {...}

    public void addOwners(String key, String[] owns) {owners.put(key,owns);}

    public void removeOwner(String key, String owner) {...}

    public void removeOwners(String key) {owners.remove(key);}

    public void clearOwners() {owners.clear();}
```

The following code snippet from JUnit tests illustrates the usage of this class.

```
Locale loc_de = new Locale("de");
Locale loc_en = new Locale("en");
Locale loc_fr = new Locale("fr");

TestMapper mapper1 = new TestMapper();
img.object2OWL(mapper1);
IndividualWrapper mapperWr1 = img.getIWR(mapper1);
TestMapper mapper2 = new TestMapper();
img.object2OWL(mapper2);
IndividualWrapper mapperWr2 = img.getIWR(mapper2);
TestMapper mapper3 = new TestMapper();
img.object2OWL(mapper3);
```

```
IndividualWrapper mapperWr3 = img.getIWR(mapper3);

@Test
public void testTextAdd() throws ParserException {
    System.out.println("addText");
    mapper1.addText(loc_de, "Dies ist ein deutscher Text");
    mapper1.addText(loc_en, "This is an English Text");
    iwr = img.getInstances(
          "hasText_de value \"Dies ist ein deutscher Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));
    iwr = img.getInstances(
          "hasText_en value \"This is an English Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));
    mapper1.clearText();
    assertNull(omg.getFirstOWLDataProperty("hasText_en"));
    mapper1.addTextLogged(loc_de, "Dies ist ein neuer deutscher Text",false);
    iwr = img.getInstances(
          "hasText_de value \"Dies ist ein neuer deutscher Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));}


@Test
public void testTextSet() throws ParserException {
    mapper1.clearText();
    Map<Locale,String> strings = new HashMap<Locale,String>();
    strings.put(loc_de, "Dies ist ein Text");
    strings.put(loc_en, "This is a Text");
    mapper1.setTexts(strings);
    iwr = img.getInstances("hasText_de value \"Dies ist ein Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));
    iwr = img.getInstances("hasText_en value \"This is a Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));

    Map<Locale,String> newstrings = new HashMap<Locale,String>();
    newstrings.put(loc_de, "Dies ist ein neuer Text");
    newstrings.put(loc_en, "This is a new Text");
    mapper1.setTexts(newstrings);
    iwr = img.getInstances("hasText_de value \"Dies ist ein Text\"^^string");
    assertEquals(0,iwr.size());
    iwr = img.getInstances("hasText_en value \"This is a Text\"^^string");
    assertEquals(0,iwr.size());
    iwr = img.getInstances(
          "hasText_de value \"Dies ist ein neuer Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));
    iwr = img.getInstances(
          "hasText_en value \"This is a new Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));

@Test
public void testRemoveSet() throws ParserException {
    mapper1.clearText();
    Map<Locale,String> strings = new HashMap<Locale,String>();
    strings.put(loc_de, "Dies ist ein Text");
    strings.put(loc_en, "This is a Text");
```

```
    mapper1.setTexts(strings);
    mapper1.removeText(loc_de);
    assertNull(omg.getFirstOWLDataProperty("hasText_de"));
    iwr = img.getInstances("hasText_en value \"This is a Text\"^^string");
    assertEquals(mapperWr1, iwr.get(0));
    mapper1.removeTextLogged(loc_en,false);
    assertNull(omg.getFirstOWLDataProperty("hasText_en"));}

@Test
public void testFriendSet() throws ParserException {
    HashMap<String,Set<TestMapper>> friendsmap =
          new HashMap<String,Set<TestMapper>>();
    Set<TestMapper> friends = new HashSet<TestMapper>();
    friends.add(mapper2);
    friends.add(mapper3);
    friendsmap.put("goodFriends",friends);
    mapper1.setFriends(friendsmap);
    iwr = img.getInstances("hasFriends_goodFriends some {Mapper_1}");
    assertEquals(mapperWr1, iwr.get(0));
    iwr = img.getInstances("hasFriends_goodFriends some {Mapper_0}");
    assertEquals(0,iwr.size());

    HashMap<String,Set<TestMapper>> oldfriendsmap =
          new HashMap<String,Set<TestMapper>>();
    Set<TestMapper> oldfriends = new HashSet<TestMapper>();
    oldfriends.add(mapper1);
    oldfriendsmap.put("oldFriends",oldfriends);
    mapper1.setFriends(oldfriendsmap);
    assertNull(omg.getFirstOWLDataProperty("hasFriends_goodFriends"));
    iwr = img.getInstances("hasFriends_oldFriends some {Mapper_0}");
    assertEquals(mapperWr1, iwr.get(0));

    mapper1.clearFriends();
    assertNull(omg.getFirstOWLDataProperty("hasFriends_oldFriends"));}

@Test
public void testFriendAdd() throws ParserException {
    mapper1.clearFriends();
    mapper1.addFriend("goodFriend",mapper2);
    mapper1.addFriend("goodFriend",mapper3);
    iwr = img.getInstances("hasFriends_goodFriend some {Mapper_1}");
    assertEquals(mapperWr1, iwr.get(0));
    iwr = img.getInstances("hasFriends_goodFriend some Mapper");
    assertEquals(mapperWr1, iwr.get(0));

    Set<TestMapper> friends = new HashSet<TestMapper>();
    friends.add(mapper1);
    mapper1.addFriends("ownFriend",friends);
    iwr = img.getInstances("hasFriends_ownFriend some {Mapper_0}");
    assertEquals(mapperWr1, iwr.get(0));
     }

@Test
public void testFriendRemove() throws ParserException {
```

```
    mapper1.clearFriends();
    mapper1.addFriend("goodFriend",mapper2);
    mapper1.addFriend("goodFriend",mapper3);
    iwr = img.getInstances("hasFriends_goodFriend some {Mapper_1}");
    assertEquals(1,iwr.size());

    mapper1.removeFriend("goodFriend",mapper2);
    iwr = img.getInstances("hasFriends_goodFriend some {Mapper_1}");
    assertEquals(0,iwr.size());

    mapper1.removeFriends("goodFriend");
    assertNull(omg.getFirstOWLDataProperty("hasFriends_goodFriend"));}

@Test
    public void testOwnerSet() throws ParserException {
    HashMap<String,String[]> ownermap = new HashMap<String,String[]>();
    String[] owners = new String[]{"Miller","Jones"};
    ownermap.put("regularOwners",owners);
    mapper1.setOwners(ownermap);
    iwr = img.getInstances("hasOwner_regularOwners some {\"Jones\"^^string}");
    assertEquals(1,iwr.size());
    assertEquals(mapperWr1, iwr.get(0));
    iwr = img.getInstances("hasOwner_regularOwners some {\"Miller\"^^string}");
    assertEquals(1,iwr.size());
    assertEquals(mapperWr1, iwr.get(0));

    owners = new String[]{"Bush","Obama"};
    ownermap.put("secretOwners",owners);
    mapper1.setOwners(ownermap);
    iwr = img.getInstances("hasOwner_secretOwners some {\"Obama\"^^string}");
    assertEquals(1,iwr.size());
    assertEquals(mapperWr1, iwr.get(0));
    iwr = img.getInstances("hasOwner_regularOwners some {\"Miller\"^^string}");
    assertEquals(1,iwr.size());
    mapper1.clearOwners();}

@Test
public void testOwnerAdd() throws ParserException {
    mapper1.clearOwners();
    mapper1.addOwner("secretOwners","Charles");
    iwr = img.getInstances("hasOwner_secretOwners some {\"Charles\"^^string}");
    assertEquals(1,iwr.size());
    assertEquals(mapperWr1, iwr.get(0));

    String[] owners = new String[]{"Tom","Tim"};
    mapper1.addOwners("secretOwners",owners);
    iwr = img.getInstances("hasOwner_secretOwners some {\"Charles\"^^string}");
    assertEquals(0,iwr.size());
    iwr = img.getInstances("hasOwner_secretOwners some {\"Tom\"^^string}");
    assertEquals(mapperWr1, iwr.get(0));}

@Test
    public void testOwnerRemove() throws ParserException {
    mapper1.clearOwners();
```

```
    System.out.println("removeOwner");
    mapper1.addOwner("secretOwners","Charles");
    mapper1.addOwner("secretOwners","Jaque");
    iwr = img.getInstances("hasOwner_secretOwners_some_{\"Charles\"^^string}");
    mapper1.removeOwner("secretOwners","Charles");
    iwr = img.getInstances("hasOwner_secretOwners_some_{\"Charles\"^^string}");
    assertEquals(0,iwr.size());
    iwr = img.getInstances("hasOwner_secretOwners_some_{\"Jaque\"^^string}");
    assertEquals(1,iwr.size());
    mapper1.removeOwner("secretOwners","Jaque");
    assertNull(omg.getFirstOWLDataProperty("hasOwner_secretOwners"));}

// Now we test the OWL to Java direction.
// The ontology TestMapperExt.owl contains some individuals.
@Test
    public void testOWL2Java() throws ParserException {
    String name = "test/TestPackage/TestMapperExt.owl";
    omg.loadExtensionOntology(name);

    Set<OWLNamedIndividual> inds = omg.getInstances("Mapper");
    assertEquals(3,inds.size());
    Iterator<OWLNamedIndividual> it = inds.iterator();
    OWLNamedIndividual NewMapper1 = it.next();
    OWLNamedIndividual NewMapper2 = it.next();
    OWLNamedIndividual NewMapper3 = it.next();
    IndividualWrapper iwr1 = img.individual2Java(NewMapper1);
    TestMapper map1 = (TestMapper)(iwr1.getObjects()[0]);
    assertEquals("Hello",map1.getText(loc_en));
    assertEquals("Bonjour",map1.getText(loc_fr));
    Iterator<TestMapper> friends = map1.getFriends("old").iterator();
    assertEquals("Name:Mapper3",friends.next().getText(loc_en));
    assertEquals("Name:Mapper2",friends.next().getText(loc_en));}
```

References

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

[2] Birte Glimm and Ian Horrocks and Boris Motik and Giorgos Stoilos. Optimising ontology classification. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *Proc. of the 9th Int. Semantic Web Conf. (ISWC 2010)*, volume 6496 of *LNCS*, pages 225–240, Shanghai, China, November 7–11 2010. Springer.

[3] Markus Dahm. Byte code engineering. In *Proceedings JIT'99*, pages 267–277. Springer-Verlag, Springer-Verlag, 1999.

[4] John H Gennari, Mark A Musen, Ray W Fergerson, William E Grosso, Monica Crubézy, Henrik Eriksson, Natalya F Noy, and Samson W Tu. The evolution of Protégé: an environment for knowledge-based systems development. *International Journal of Human-Computer Studies*, 58(1):89 – 123, 2003.

[5] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(0):11–21, 2011.

[6] C. Lutz. Description Logics with Concrete Domains—a survey. In *Advances in Modal Logics Volume 4*. King's College Publications, 2003.

[7] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics*, 5(2):51–53, June 2007.

[8] D. Tsarkov and I. Horrocks. FaCT++ Description Logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.