

# RDFReactor – From Ontologies to Programmatic Data Access

Max Völkel  
Institute AIFB  
Karlsruhe, Germany  
voelkel@aifb.uni-karlsruhe.de

## ABSTRACT

Developers familiar with object oriented programming languages have to make a paradigm shift in order to produce and manage data usable on the Semantic Web (e.g. RDF). In this paper we describe the tool RDFReactor which transforms a given ontology in RDF Schema into a familiar, dynamic, object-oriented Java API – at the push of a button. Developers then are able to interact with java instances, thus allowing them to stay in their own world. The generated proxy objects contain no state and delegate all method calls to RDF model updates and queries. RDFReactor potentially turns every Java developer into a Semantic Web application developer and enables them to use RDF correctly, efficiently and effectively without even knowing it. It is downloadable (GNU LGPL) at <http://rdfreactor.ontoware.org>.

## 1. INTRODUCTION

A key promise of the Semantic Web is that of global interoperability, i. e. applications developed independent of each other will be able to read and use each others data. Ontologies are key enablers for the Semantic Web, they describe the semantics of data to enable ad-hoc interoperability. The Semantic Web is already rich in ontologies, but poor in applications that use semantic data. Why? Some evidence can be found by using Google queries which e. g. show millions of hits for “Java developer” and only hundred thousands of hits for queries like “ontology engineer”. This might indicate a shortage of *ontology engineers*, who currently can also be seen as *developers* for ontology based applications.

Reuse of existing ontologies is crucial for efficiently and effectively reaching semantic interoperability on a global scale. Unfortunately developers familiar with object oriented programming languages have to make a paradigm shift in order to produce data usable on the Semantic Web (e.g. RDF<sup>1</sup>). This means, all generated RDF instance data should be described by terms of an ontology. The task to make an existing Java application interoperable with the Semantic Web is a difficult task, as developers have to learn at the same time the RDF data model, RDF Schema syntax and semantics and an API for model manipulation.

The main contribution of our work is to leverage the power and quantity of Java developers and Java tools for the Se-

<sup>1</sup><http://www.w3.org/RDF/>

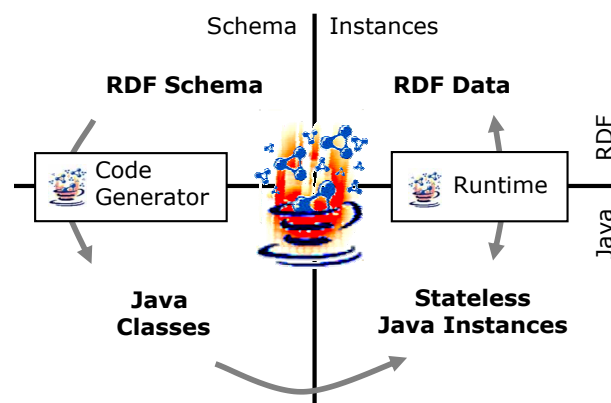


Figure 1: Mapping the two worlds

semantic Web by significantly reducing this burden. We introduce RDFReactor, a new open-source tool, which transforms a given RDF Schema ontology into an object-oriented Java API with domain-centric methods like `paper.setAuthor(Author a)` instead of `model.addTriple(...)`. This enables developers to interact with java proxy objects, thus allowing them to stay in their own world and at the same time to make use of the advantages RDF offers.

## 2. OVERVIEW

We distinguish two phases in application development: *API generation* followed by traditional *development*. An overview of the whole system can be found in Fig. 1: An RDF Schema (or an OWL ontology) is run once through the RDFReactor code generator which generates a number of type-safe, domain-specific Java classes. At runtime, instanced of these classes act as stateless proxies on the RDF model.

In the remainder of this section we will first explain the features of the generated API and explain how these features were realised. Then, we explain features of the code generator and explain how those were realised.

### 2.1 Runtime Features

In this section we explain the mapping from the object-oriented Java world to the triple-oriented RDF world. Basically, we map RDFS (or OWL) classes to Java classes and RDF properties to Java properties, accessed through `get()` and `set(value)` methods. This makes RDFReactor APIs most suitable to be used for instance manipulation,

e. g. in user interfaces on a desktop or web user interfaces. Other libraries, such as SemVersion<sup>2</sup>, also successfully used RDFReactor for internal RDF data model manipulation. Additionally, every class generated by RDFReactor offers the methods `Type[] getAll()`, `add( Type x)` and `remove( Type x)`, do deal with the multi-value nature of RDF properties. In fact, `get()` throws an exception if multiple values are present in the RDF model. The set-Method internally removes all values and then adds the given value.

Note that all API methods are type-safe. A developer using an RDFReactor generated API thus has full support from the Java compiler and it's IDE, e. g. auto-completion of method names.

At runtime, instances of the Java classes model resources in the RDF model. In the constructor, a reference to an RDF model and an identifier has to be supplied. We allow both for URIs or blank nodes to act as identifiers. There are no limitations with respect to blank node handling. For the RDF model manipulation, we use an abstraction layer, RDF<sub>2</sub>Go<sup>3</sup>, which has adaptors to Jena[5], Sesame, YARS and others.

One of the key advantages of using an RDFReactor generated API is synchronicity: Each method set or add requests is translated into an add statement operation, each get or getAll method call is executed as a query to the RDF model. All java instances are completely stateless, they act like proxies for RDF resources. Concurrent usage of the same data, e. g. access as a web portal, is also no problem.

Additionally, the API offers the following set of operations:

**SPARQL:** The `Bridge` class supplies a method to perform arbitrary SPARQL SELECT queries and return the result as RDFReactor instances, as requested.

**List support:** We offer methods to transform `rdf:List` constructs to Java lists and vice vers.

**Map support:** Each class generated by RDFReactor additionally implements the Java Collections Map interface, mapping properties to keys and the objects of RDF statements to Map entry values. Again, all calls are transparently translated into triple update and query operations, no state is duplicated at any time.

**Inverse properties:** As properties in RDF often hold references to other objects it makes a lot of sense to query also in the other direction. Instead of asking "whom do I know?" it is also interesting to find out "who knows me?" on the level of RDF resources. Thus the generated classes offer methods to get a list of backwards related entities. For each property  $p$  relating instances of class  $A$  with class  $B$ , the class  $A$  has a method to get, set, add and remove values as explained above. The class  $B$  has a method `Type[] getP_Inverse()` to get all instances of class  $A$  related to  $B$  via  $p$ .

<sup>2</sup>an RDF and ontology versioning system, see <http://semversion.ontoware.org>

<sup>3</sup><http://rdf2go.ontoware.org>

**Typing:** The Java keyword `instanceof` cannot reflect schema or type changes due to the static nature of the Java language. As it is a keyword it can not even be overwritten with advanced method such as dynamic proxies. Instead, the method `Class[] getAllType()` returns a list of all RDF or OWL classes this instance of *currently* type of.

**Cardinality constrains:** OWL cardinality constraints are reflected in the API: `add` raises a `CardinalityException` if more elements than allowed are added. Similarly, `remove` raises an exception if a min cardinality constrains would be violated. The cardinality constraints are modelled statically in the generate code and do not reflect schema changes. This allows to generate methods that raise no exceptions for properties without constraints. Such methods are easier to use for developers.

**Triple Store Independent.** All generated APIs access RDF models only through RDF<sub>2</sub>Go, thus the generated API can be ported without changes to another triple store.

## 2.2 Code Generator Features

RDFReactor translates an ontology into a number of domain-specific classes, which are explained in the previous sections. In this section, we explain features of the code generator.

The developer creates or, preferably, reuses an RDF Schema or OWL ontology as the backbone of the application. We map ontology classes to domain-specific Java classes and relations to properties, as explained in Sec. ?? and Sec. ?. The code generate employs a number of features to help an ordinary Java developer to become a semantic web developer (without actually having her to explain anything beyond URIs in most cases):

- The Java classes and methods are in most cases named according to usual Java coding conventions, even if no labels are present in the ontology.
- RDFReactor works both with RDF Schemas and OWL ontologies, respecting the ontology semantics (i. e. reasoning is used).
- After code generation, the API is ready to use. It's fully implemented and documented (JavaDoc).
- Multiple inheritance is resolved, a subclass tree is extracted to produce more readable code.
- Multiple domains and ranges are handled in an appropriate, type-safe manner.
- Changes to the generated source code are easy: Most methods are implemented with a single line of code. If frequent schema changes are expected, customized subclasses should be used, so that changes don't get lost when the code generation is triggered again.
- Cardinality constrains are reflected in the API and checked at runtime.
- The generated source code style is easy to change, as it is generated with a template language.

In the next section we explain *how* these features were designed.

### 3. RUNTIME DESIGN

A key of the runtime design is a subclass hierarchy modelling ontological layering in Java. The lowest layer mediates between RDF's graph model and object-oriented concepts. We now explain the layering in detail:

**OO-Graph mapping layer:** hand-written methods to translate from object-oriented method calls into triple-centric methods.

*ReactorBase* is the key to understand RDFReactor concepts. It acts as a generic RDF resource, without any RDF or RDFS semantics beyond the triple model. Each *ReactorBaseImpl* instance knows its URI (or blank node) and the RDFS Class it represents in the API.

Each generated Java class directly or indirectly inherits from *ReactorBaseImpl*, which is responsible for mapping the object oriented Java view to the triple-oriented model of RDF. *ReactorBase* instances map to RDF resources and offers methods to manipulate triples of the form (*i*, \*, \*), where *i* is then instance identifier, a blank node or a URI. E.g. when the user calls `setPhoneNumber( 123 )` on an instance of *Person*, this call is delegated to the *ReactorBase* method `set( PHONENUMBER, 123 )`, where *PHONENUMBER* is a URI. In this example, *ReactorBase* converts 123 into an RDF literal of type *xsd:int*.

Multiple *ReactorBase* instances looking at the same resource can have different RDFS Classes, of course, allowing for different perspectives on the same resource.

The RDF generated by such method calls is always compliant—or rather: described—by the ontology the API was generated from. This ensures that the RDF data is usable by other tools (even concurrently) as well.

*ReactorBase* also implements a `Map<URI, Object>` interface, allowing *all* *RDFReactor* instances to be used additionally in this generic fashion.

**Ontology language layer:** on this and lower layers, URIs have semantics. In our case, RDF, RDFS, OWL and XSD are considered. Classes like *Resource* and *Class* offer methods to access typical RDF(S) properties such as labels (`rdfs:label`), comments (`rdfs:comment`), types (`rdfs:type`) and subclasses (`rdfs:subClassOf`). These classes have been generated by applying the Code Generator to the RDF Schema definitions of RDFS and OWL. In fact, the auto-generated methods for the property *type* in the class *Resource* are used as a substitute to Java's `instanceof`, as explained above.

Note how *RDFReactor* can operate on RDFS's class and property structures without running into any meta-modelling issues. This simply due to the fact that all RDFS and OWL semantics are handled in the underlying store.

**Domain layer:** this layer contains the code generated from a particular RDFS or OWL ontology, e.g. *Person* or *Image*. They inherit from the class *Class* of the layer above.

In order to illustrate the layered design, we give now some examples. We could use the *ReactorBase* implementation directly and invoke it as follows:

```
1: URI FoafPhone =
    new URI("http://xmlns.com/foaf/0.1/phone");
2: URI FoafKnows =
    new URI("http://xmlns.com/foaf/0.1/knows");
3: ReactorBase heiko =
    new ReactorBaseImpl("urn://heiko");
4: ReactorBase max =
    new ReactorBaseImpl("urn://max");

5: heiko.add( FoafPhone, 123 );
6: heiko.add( FoafKnows, max );

7: int i = (int) heiko.get( FoafPhone, Integer.class );
8: Person[] heikosFriends =
    (Person[]) heiko.getAll( FoafKnows, Person.class );
```

The lines 5 and 6 immediately results in the triples { <urn://heiko> foaf:phone "123"(xsd:int)} and { <urn://heiko> foaf:knows <urn://max>} to be added to the model. This is nice, but *RDFReactor* can do better by generating type-safe domain-specific classes (which is assumed in line 8). Let's assume we generated a class *Person* from a FOAF-like ontology [1]. It will look as follows:

```
class Person extends ReactorBaseImpl

    public void addPhone( int value ) {
        add( FoafPhone, value );
    }

    /**
     * @throws RDFDataException
     * if multiple values found
     */
    public int getPhone() throws RDFDataException
        return (int) get( FoafPhone );
    }

    public void addKnows( Person p ) {
        add( FoafKnows, p );
    }

    public Person[] getAllKnows() {
        return (Person[]) getAll( FoafKnows, Person.class );
    }
```

As you can see, the domain specific classes are merely a customising *ReactorBaseImpl* by hiding some type conversions. This reflects well the strategy employed in the design of *RDFReactor*: We separate the concern of graph-OO-mapping from the concern of mapping loosely-typed-RDF from strictly-typed Java. The domain-specific class can be used by ordinary Java developers without any knowledge about RDF as follows:

```
Person p = new Person("urn://heiko");
p.setAge(29);
int i = p.getAge();
Person[] heikosFriends = heiko.getAllKnows();
```

We currently have support for the following datatypes as values in domain specific methods (`get`, `set`, ...): all *ReactorBase* instances, URI, URL, String, *DatatypeLiteral*, *LanguageTagLiteral*. Additionally with correct XSD mapping: Integer, Long.

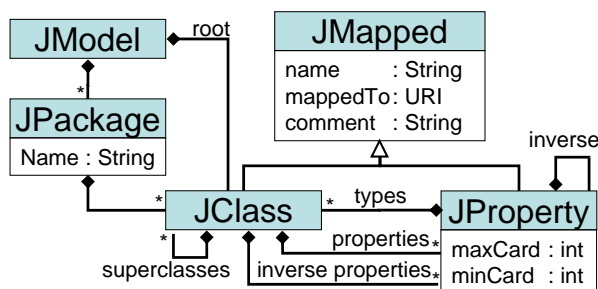


Figure 2: JModel - the internal ontology model

## 4. CODE GENERATOR DESIGN

First, we translate the ontology into an internal model (JModel). The internal model (c.f. Fig. 2) looks as follows:

**JModel** models a set of *JPackages*. It is mapped to an RDF model.

**JPackage** models a package in Java, e.g.

"org.ontoware.rdfreactor.foaf". Modelling this explicitly allows to map different RDF namespaces to different Java packages. A *JPackage* has a name and a list of classes.

**JClass** has a name (e.g. "Person"), a URI it is mapped to, a comment, a list of properties, a list of inverse properties and a list of superclasses.

**JProperty** has a name (e.g. "phone"), a URI it is mapped to, a comment, a list of types (*JClasses*), a minimum and maximum cardinality and an inverse *JProperty*.

The JModel is very expressive and allows many things that are illegal in Java, e.g. class cycles, multiple inheritance and properties with no type. Algorithms to make the JModel stricter are successively applied, allowing for reuse regardless of source ontology language. Finally, the JModel is strict enough to generate source code with the help of a template processor. In the remainder of this section we explain the code generation process in detail. In the future, we consider replacing the JModel with the Ecore framework from Eclipse or other works based on the *Ontology Definition Metamodel* [2].

**Load built-ins.** We initialise a JModel with the built-in mappings of URIs to the Java classes of the schema layer (c.f. Sec. 3). Of course, the default model is different for RDFS and OWL. A new *JPackage* for the API to be generated is created and added to the JModel.

**Inferencing.** RDFS and OWL inferencing is applied to the source ontology. All handling of ontology language semantics is thus delegated to existing implementations.

**Inspect Classes.** For each ontology class *oc*, we distinguish three cases. First, if it is an OWL restriction and thus not a *real* class: ignore it. Second, if the JModel already contains a *JClass* mapped to the URI of *oc*: ignore it as well. Otherwise we add a new *JClass* to the *JPackage*. In order to find a legal Java class name, a number of little things has to be done, see Sec. 4.1.

If the ontology class has an *rdfs:comment* it is copied over in the JModel. Later it is integrated with the auto-generated JavaDoc of the class.

**Multiple Inheritance.** Inheritance is tackled in two steps: First, each superclass relation in the ontology is mapped to a superclass relation in the JModel. Second, the inheritance hierarchy is flattened, to remove multiple inheritance, as described in Sec. 4.2.

**Inspect Properties.** For each property *op* in the ontology: In the JModel, we look up the *JClasses* mapped to the IDs (URI or blank node) of *op*'s domains. For each domain *d* we generate a *JProperty jp* and attach it to the respective

*JClass*. The *JProperty* is named and commented with the same strategy as classes (c.f. Sec. 4.1).

Then, for each range relation of the property *op*, we add the corresponding *JClass* as a type of the property *jp*. Multiple types for a property are handled in later steps of the code generation.

**Inverse Properties.** For each *JProperty jp* with domain *d* in the JModel, we add inverse properties: For each *JClass r* in the range of *jp* we add an inverse property '*jp*-Inverse' to *r* with the range *d*.

**Source Code Generation.** Finally the JModel is suitable for a template-based code generation. Emitting source code with a template language allows better control over the generated syntax, especially indentation. We initialise a Velocity<sup>4</sup> template and use it to generate the source code of a Java class for each *JClass* in the JModel. The template has about 300 lines of codes and is described in detail in Sec. 4.3.

The overall process of code generation can be summarised as transformation from source ontology language into an internal representation (JModel), stepwise simplification of the internal model until a template-based code generation approach can emit the final source code. In the remainder of this section we explain in detail issues regarding naming, multiple inheritance and the template.

### 4.1 Naming

In order to name a class or a property adequately, a number of strategies are used:

First, a raw name is created by looking at the *rdfs:label* in the schema (e.g. 'IM-account'). If no label was found, the part of the URI after the fragment identifier ('#') is used (e.g. 'imAccount'). If no such delimiter is present in the URI, the part after the last slash ('/') is used. This name is then transformed into a legal Java identifier as follows: Spaces and underscore are removed. Then all non-alphanumeric letters are converted into an underscore (e.g. 'IM-account'). If the resulting name *starts* with a number, the name is prefixed with 'a.'. Finally, the first letter is capitalised, to reflect common Java naming conventions (e.g. 'ImAccount').

The resulting name is compared with a list of taken names, to avoid name clashes. If such a conflict occurs, a longer name is created from the URI. If the part before the last number sign or slash is known, the correct namespace prefix is used (e.g. 'foaf', resulting in 'FoafImAccount'). Otherwise the last part of the URI consisting only of letters is used (e.g. from the URI 'http://xmlns.com/wordnet/1.6/' the prefix 'wordnet' would be extracted, as '1.6' is not consisting only of letters, resulting in 'WordnetImAccount'). If that name is *also* taken already in the schema at hand, a fallback to transform the full URI into a legal Java identifier is used. As it turns out, naming is less simple than one might think. For properties, removing leading parts such as 'is', 'has', 'is\_', 'has\_' leads to methods names better reflecting common Java API design.

### 4.2 Inheritance Flattening

For each class in the JModel the following cases have to be considered:

- If the class has no superclass, the root class of the JModel (either *rdfs:Class* or *owl:Class*) is set.
- Classes with exactly one superclass are left unchanged.
- If multiple superclasses are found, the superclass that has itself most superclasses is used. Note that setting the JModel root class for all classes as the superclass would result in no loss of functionality in the generated API. The whole point of modelling ontology subclass relations at least partially in Java is the desire to generate code that is simpler to change manually later on. There are other options to extract a strict tree from the inheritance graph in the ontology, though.

<sup>4</sup>Jakarta Velocity, a template engine. See <http://jakarta.apache.org/velocity/>

## 4.3 Template

The template has to take care of a number of things in order to produce a good source code. In particular the template generates the following:

**Minimal Header.** A package declaration, import statements for basic RDF classes and conditional imports, e.g. for exceptions. Only classes that are used are imported, to avoid having IDEs such as Eclipse to bug the user with warnings.

**Binding.** Each class is bound to an specific ontology class with a URI constant.

**Documentation.** An overview comment that explains which properties are handled in this class is also added to the JavaDoc. Each class and method are also documented.

**Constants.** All URIs that are used in the body of the class are defined as constants.

**Constructors.** A number of constructors allowing for instances to be created from a URI, URL, blank node or no identifier (a random URI will be used). Some protected constructors are also generated, to allow constructor delegation up to the top (ReactorBaseImpl).

**Properties.** For each property, methods are generated *according to cardinality constraints and number of given ranges*. If no cardinality constraints have been state or the maximum cardinality is exactly 1, a get and set methods are created. A remove method, encoding cardinality constraints in the implementation as needed (e.g. 'remove( value, 3 )', it a minimum of three elements must be guarded), is generated. If the maximum cardinality is greater than 1 or not set, add and getAll methods are generated.

Multiple ranges result in multiple set or add methods with the same name but different types, exploiting method overloading. For get or getAll methods, it is not possible to have the same method with different return types. Thus we disambiguate method names with a postfix of 'asATYPE' to indicate the desired return type.

**Lean code.** All generated methods are implemented (typically in single line of code, thanks to the inheritance), documented (in JavaDoc) and throw appropriate exceptions (as documented in the JavaDoc) if things go wrong.

**Queries.** A method to return all instances of the generated class for a given model.

**Inverse Properties.** For each inverse property of the class, get and add methods.

## 4.4 Implementation Details

First we wrote the ReactorBaseImpl class and hand-coded essential parts of the schema layer (c.f. Sec. 3). Then we wrote the code generator, making use of the schema layer classes (e.g. `getLabel()`). Then we ran the preliminary code generator on the RDF Schema of RDF Schema to get a better (more features, better documented) schema layer. The new schema layer classes were then used to enhance the code generator. This way of bootstrapping seems well suited for other programming languages as well.

As a second remark, the current handling of primitive data types works only thanks to the new feature autoboxing and unboxing introduced in Java 1.5. Autoboxing converts seamlessly from primitive types and their corresponding Classes, e.g. `int` and `Integer.class`.

## 5. MODEL MANIPULATION DESIGN

In order to stay independent of a specific triple store and be able to offer RDFReactor APIs on top of a number of stores, we developed the abstraction layer RDF2Go. RDF2Go is a simple wrapper over Java triple (and quad<sup>5</sup>) stores, similar in spirit of Jakarta Commons Logging. It allows developers to un-tangle

<sup>5</sup>Quad stores store quads, not only triples. This is also known as *Named Graphs*

their semantic web applications from a specific store and thus profit from using always the best store (performance, features). Writing new adaptors is easy: only 6 methods have to be implemented.

RDF2Go currently<sup>6</sup> comes with adaptors for Jena 2.2, Jena 2.3, Sesame 1, Sesame 2, YARS, NG4J and even an experimental adaptor for the *Java Content Repository API* (JSR170). The core features of RDF2Go are:

- add and remove triples with one-method-call, no costly object creation involved
- query for triple patterns
- SPARQL (although not all stores support it)
- full support for quad stores
- built-in URIs for RDF, RDFS, XSD
- NO support for parsing. Users should use the supplied parser from the triple stores instead. We don't re-implement everything.

As a brief introduction, RDF2Gomaps to Java as follows:

**uri:** `java.net.URI`

**blank node:** `org.ontoware.rdf2go.BlankNode`

**plain literal:** `java.lang.String`

**language tagged literal:** `org.ontoware.rdf2go.LanguageTagLiteral`

**datatyped literal:** `org.ontoware.rdf2go.LanguageTagLiteral`

This allows to add statements as simple as

```
// assume some URIs have been created
URI mike = new URI("urn://mike"); URI foafName = new
URI("http://foaf.com#name");
```

```
// add triple
model.addStatement( mike, foafName, "Mike Müller" );
```

There are a number of related approaches, but to our knowledge none is as simplistic and comes with such a large number of adapters. Related approaches are KPOntology<sup>7</sup>, Trippi<sup>8</sup> and of course the inventor of all these<sup>9</sup>.

Internally, RDFReactor relies on Jena 2.3 (accessed through RDF2Go) in the code generator, which is explained in the next sections.

## 6. RELATED WORK

There have been several projects yet, that attempted to map RDFS to Java. The predecessor of RDFReactor was *OntoJava* [3] from AIFB. OntoJava was an early approach and lacks basic features such as multiple inheritance, RDF resources with more than one RDF literal related to it or RDF properties with multiple domains.

*Rdf2Java*<sup>10</sup> overcomes the inability of RDF Schema to express *restrictions* by using the *Protégé* [6] annotations. RDF Schemas created with *Protégé* can therefore be used to generate constrained Java classes. *Rdf2Java* generates Java classes and stores all state at runtime in member variables. Although the tool allows import and export of RDF data at runtime, the user still has to invoke this periodically.

*OntoJava* and *rd2Java* both generate source code for Java classes and hold state at runtime in Java instances. This leads to divergences between the two models and consistency problems.

An approach to map OWL full to Java is described in [4]. Here the multiple inheritance problem, which applies to RDF Schema as well, is solved by using Java interfaces. In Java, a class can

<sup>6</sup>release downloads, including all libraries: [http://ontoware.org/frs/?group\\_id=37&release\\_id=173](http://ontoware.org/frs/?group_id=37&release_id=173)

<sup>7</sup><http://kpointology.isoco.com/download.html>

<sup>8</sup><http://trippi.sourceforge.net/>

<sup>9</sup><http://www-db.stanford.edu/~melnik/rdf/api.html>

<sup>10</sup><http://rdf2Java.opendfki.de>

have only one superclass, but may implement many interfaces. Unfortunately, the OWL type system is only present in the form of raised exceptions, as all properties are mapped to methods that get and set generic non-type-safe `Java.util.List` methods.

Summary: The current mapping approaches create custom Java classes in source code and give the user ways to load RDF into this object model and write it back as well. State is thus maintained in Java instances and not in the RDF model. This leads to divergences between the two models and consistency problems. Also some usability features such as a customisation option is currently missing from all tools.

## 7. EVALUATION

In order to demonstrate the usefulness of RDFReactor we show a small program which shows how the API is used. The sample program creates a FOAF-like file.

```
public static void main(String[] args) {
    // set up an RDF2Go model with no reasoning
    Model model = new ModelImplJena23(false);

    // use generated FOAF API
    Person max = new Person( model,
        new URI("http://www.xam.de/#i") );
    max.setFirstName("Max");
    max.setFamilyName("Völkel");
    max.setMBoxHashed("25ab1214...67700");
    max.setNick("xamde");
    max.setHomepage("http://www.xam.de");
    max.setPhone("+49-171-8359678");

    Person denny = new Person( model,
        new URI("http://example.com/#denny") );
    denny.setName("Denny Vrandecic");
    denny.setMBoxHashed("4789fb144...fea3d12");
    denny.setRdfSeeAlso(
        "http://www...english?id_db=2097");
    max.addKnows(denny);
}
```

## 8. CONCLUSION

In this paper we have shown how a domain-centric, usable Java API can be generated from an arbitrary RDF Schema. Our implementation, RDFReactor, is due to its dynamic nature always in-sync with the RDF data model. This inherently allows for concurrent access to the RDF data model. Each class inherits from `ReactorBase`, which allows the developer to manipulate arbitrary RDF properties directly (`set(URI prop, Object o)`). Thus we do not restrict the expressivity of RDF in any way. Additionally, code generated by RDFReactor is fully customisable, thus method names and URIs can be changed easily.

We help to make the ontology reuse promise a reality by enabling the average Java developer to consume and produce data conforming to existing ontologies through domain-specific Java APIs. The main advantage of our approach is that developers who use the generated API don't have to know RDF at all, but can make full advantage of its' capabilities. RDFReactor is downloadable (Open Source GNU LGPL) at <http://rdfreactor.ontoware.org>.

**Acknowledgements.** This research was partially supported by the European Commission under contract FP6-507482 (Knowledge Web<sup>11</sup>) and FP6-027705 (Nepomuk<sup>12</sup>). The expressed content is the view of the authors but not necessarily the view of the Knowledge Web Network of Excellence as a whole. Many thanks to Andreas Eberhart, Daniel Oberle, Sudhir Agarwal, Peter Haase and Heiko Haller.

<sup>11</sup><http://knowledgeweb.semanticweb.org>

<sup>12</sup><http://nepomuk.semanticdesktop.org>

## 9. REFERENCES

- [1] L. M. Dan Brickley. Foaf vocabulary specification, 04 2004.
- [2] I. S. S. DSTC, Gentleware. Ontology definition metamodel.
- [3] A. Eberhart. Automatic generation of java/sql based inference engines from rdf schema and ruleml. In *Lecture Notes in Computer Science*, volume 2342, 01 2002.
- [4] A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of owl ontologies into java. In *Proceedings of SEKE 2004*, Banff, Canada, June 2004.
- [5] B. McBride. Jena: Implementing the rdf model and syntax specification. *Processings of the Semantic Web Workshop at the WWW2001*, 2001.
- [6] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with protégé-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

## APPENDIX

### A. SAMPLE ONTOLOGY

```
@prefix rdf:
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:
  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix test:
  <http://purl.org/net/xamde/ns/test#> .

test:Agent a rdfs:Class.

test:Person a rdfs:Class
; rdfs:subClassOf test:Agent
.

test:Paper a rdfs:Class.

test:Conference a rdfs:Class .

test:Researcher rdfs:subClassOf test:Person
; rdfs:comment "They are almost like normal people"
.

test:email a rdf:Property
; rdfs:domain test:Agent
; rdfs:range rdfs:Literal
.

test:hasName a rdf:Property
; rdfs:domain test:Person
; rdfs:domain test:Paper
; rdfs:range rdfs:Literal
.

test:writes a rdf:Property
; rdfs:domain test:Researcher
; rdfs:range test:Paper
.

test:publishedAt a rdf:Property
; rdfs:domain test:Paper
; rdfs:range test:Conference
.

test:knows a rdf:Property
; rdfs:domain test:Person
; rdfs:range test:Person
.
```

## B. GENERATED RESULT

```
/**
 * generated by RDFReactor on 10.03.06 23:14
 */
package org.ontoware.example.foaf;

import java.net.URI;
import java.net.URL;

import org.ontoware.rdf2go.BlankNode;
import org.ontoware.rdf2go.Model;
import org.ontoware.rdf2go.impl.URIUtils;

import org.ontoware.rdfreactor.runtime.RDFDataException;

/**
 * This class was generated by <a href="http://RDFReactor.ontoware.org">RDFReactor</a> on 10.03.06 23:14
 * This class manages access to these properties:
 * <ul>
 * <li> Name </li>
 * <li> Knows </li>
 * </ul>
 */
public class Person extends Agent {

    /** http://purl.org/net/xamde/ns/test#Person */
    public static final URI RDFS_CLASS = URIUtils.createURI("http://purl.org/net/xamde/ns/test#Person");

    /** http://purl.org/net/xamde/ns/test#hasName */
    public static final URI NAME = URIUtils.createURI("http://purl.org/net/xamde/ns/test#hasName");

    /** http://purl.org/net/xamde/ns/test#knows */
    public static final URI KNOWS = URIUtils.createURI("http://purl.org/net/xamde/ns/test#knows");

    /**
     * Returns a Java wrapper over an RDF object, identified by URI.
     * Creating two wrappers for the same instanceURI is legal.
     * @param model RDF2GO Model implementation, see http://rdf2go.ontoware.org
     * @param classURI URI of RDFS class
     * @param instanceURI URI of this instance
     */
    protected Person ( Model model, URI classURI, URI instanceURI ) {
        super(model, classURI, instanceURI);
    }

    protected Person ( Model model, URI classURI, BlankNode bnode ) {
        super(model, classURI, bnode);
    }

    protected Person ( Model model, URI classURI, URL instanceURL ) {
        super(model, classURI, instanceURL);
    }

    /**
     * Returns a Java wrapper over an RDF object, identified by URI.
     * Creating two wrappers for the same instanceURI is legal.
     * @param model RDF2GO Model implementation, see http://rdf2go.ontoware.org
     * @param uri URI of this instance
     */
    public Person ( Model model, URI uri ) {
        super(model, RDFS_CLASS, uri);
    }

    /**
     * Returns a Java wrapper over an RDF object, identified by a URL.
     * Creating two wrappers for the same URL is legal.
     * @param model RDF2GO Model implementation, see http://rdf2go.ontoware.org
     * @param url URL of this instance
     */
    public Person ( Model model, URL url ) {
        super(model, RDFS_CLASS, url);
    }
}
```

```

}

/**
 * Returns a Java wrapper over an RDF object, identified by a blank node.
 * Creating two wrappers for the same blank node is legal.
 * @param model RDF2GO Model implementation, see http://rdf2go.ontoware.org
 * @param bnode BlankNode of this instance
 */
public Person ( Model model, BlankNode bnode ) {
    super(model, RDFS_CLASS, bnode);
}

/**
 * Returns a Java wrapper over an RDF object, identified by
 * a randomly generated URI.
 * Creating two wrappers results in different URIs.
 * @param model RDF2GO Model implementation, see http://rdf2go.ontoware.org
 */
public Person ( Model model ) {
    super(model, RDFS_CLASS, model.newRandomUniqueURI());
}

/**
 * @return all instances of this class
 */
public Person[] getAllInstances() {
    return (Person[]) getAllInstances(super.model, Person.class);
}

/**
 * @return all A's that have a relation 'Knows' to this Person instance
 */
public Person[] getAllKnows_Inverse() {
    return (Person[]) getAll_Inverse(Person.KNOWS, this.getID(), Person.class);
}

/**
 * add 'Knows'-Inverse
 */
public void addKnows_Inverse(Person value) {
    value.add( Person.KNOWS ,this);
}

/**
 * @return the only value. null if none is found
 * @throws RDFDataException, if the property has multiple values
 */
public java.lang.String getName() throws RDFDataException {
    return (java.lang.String) get(NAME, java.lang.String.class);
}

/**
 * removes all values and sets this one
 */
public void setName( java.lang.String value ) {
    set(NAME, value);
}

/**
 * removes a value
 */
public void removeName( java.lang.String value ) {
    remove(NAME, value);
}

/**
 * adds a value
 */
public void addName( java.lang.String value ) {
    add(NAME, value );
}

```



```

/**
 * returns all values
 */
public java.lang.String[] getAllName() {
    return (java.lang.String[]) getAll(NAME, java.lang.String.class);
}

/**
 * @return the only value. null if none is found
 * @throws RDFDataException, if the property has multiple values
 */
public Person getKnows() throws RDFDataException {
    return (Person) get(KNOWS, Person.class);
}

/**
 * removes all values and sets this one
 */
public void setKnows( Person value ) {
    set(KNOWS, value);
}

/**
 * removes a value
 */
public void removeKnows( Person value ) {
    remove(KNOWS, value);
}

/**
 * adds a value
 */
public void addKnows( Person value ) {
    add(KNOWS, value );
}

/**
 * returns all values
 */
public Person[] getAllKnows() {
    return (Person[]) getAll(KNOWS, Person.class);
}
}

```

## APPENDIX

## A. REVIEWS JUC2006

### A.1 Reviewer 7

#### A.1.0.1 Criteria.

An excellent paper, very suitable for the expected conference audience. The tool is well described both in terms of its features and implementation.

#### A.1.0.2 Overall assesment.

5

#### A.1.0.3 Comments to the authors.

### A.2 Reviewer 14

#### A.2.0.4 Overall assesment.

5

#### A.2.0.5 Comments to the authors.

##### Content

*E.g: Are the goals of the project or demo clear? Is the maturity of the work explained? Have key design choices been characterised and explored? Has the work been robustly assessed?:*

The paper describes an approach to map RDFS and OWL ontologies to Java classes, generating working code that can be used to interact with instances of the ontology. The Jena framework is used as a basis for that project (together with other RDF frameworks).

The work is mature and seems to be used in real projects. The argumentation is sound and logic, different common pitfalls (naming, ease of use, strong typing) are addressed. The work is repeatable and published as open-source, so this is a contribution.

What I miss: You argument that you work will "make the world a better place" in the introduction. Which is ok, but perhaps you can add a nice story about a programmer that already lives in this happy place (cut the evaluation section in favor for a nice war story?)

Logic Flaw: "Concurrent usage of the same data, e. g. access as a web portal, is also no problem." -> are you sure? As RDFReactor does not (from itself) provide transactions, there is the possibility of having non-valid RDF data in the store at some moments.

Correctly, RDFReactor does not solve that problem, so perhaps cut this.

Robustness and practibility: Clearly, RDFReactor is not usable in a client-server environment. So - as most Semantic Web applications are client-server - what is your answer? You might explain that the approach works best on in-mem models and does not support transactions, perhaps give a slight example how it can be used in a client-server example (without live access to the servers). Example: a server might not accept an instance that missed required properties, so you have to add all of them before "committing". RDFReactor does not support that, nor does any other RDF api :-| For example, the typical Java approaches to map relational databases to in-mem representations might help with that argument (DB people always think in transactions).

Flaw?: In the paper you mention that you implement inverse relations using `getP_inverse()`. In the demo down-

loadable at your website, the Person-example does not have these inverse. So - do you support inverse (was the demo generated with this option turned off?) The same can be said for the appended code examples.

You talk about "documentation in the javadoc" but there, you missed the chance to include the `rdfs:Description` of properties - or? (is the `rdfs:description` of classes and properties included in the javadoc? it would be good if it is.)

**Presentation** *E.g: Are the key ideas clearly and succinctly presented? Does the presentation assume too much or too little background knowledge? Is there too much or too little detail?:*

Sturdy scientific engineering work, no flaws there.

Side remark: Author does not conform to the submission format as text is two column. "Papers should be in 12 point Times font, single column per page, A4 or US Letter. PDF is preferred." I also accept it 2-column, it improves readability. Check with workshop organizers if this is a problem!

Abstract is a concise bounded description of RDFReactor, well done.

"Unfortunately developers familiar with object oriented programming languages have to make a paradigm shift in order to produce data usable on the Semantic Web (e.g. RDF1). This means, all generated RDF instance data should be described by terms of an ontology." -> these two sentences sound not correct (this means, I cannot determine that means from the first sentence). Let check by native english speaker!

Typo: "to domain-specific Java classes and relations to properties, as explained in Sec. ?? and Sec. ??." -> check your latex there.

Typo: "The code generate employs a number" generator?

Typo: "to become a semantic web adeveloper" -> we alla ara adeveloper :-)

Style: In section 3. RUNTIME DESIGN the use of enumerations/indent looks weird (indenting a whole column) perhaps switch to subsubsection ?

Content: Section "OO-Graph mapping layer:" Too much detail... please cut the details of "ReactorBase instances map to RDF resources and offers methods to manipulate triples of the form (i, \*, \*), where i is then instance identifier, a blank node or a URI." thats just too much. If I want to see the code, I can (thanks to your open-source policy) download it.

"The RDF generated by such method calls is always compliant- or rather: described-by the ontology the API was generated from. This ensures that the RDF data is usable by other tools (even concurrently) as well." What do you want to say? Is this paragraph needed?

Understanding: "This reflects well the strategy employed in the design of RDFReactor: We separate the concern of graph-OO-mapping from the concern of mapping loosely-typed-RDF from strictly-typed Java." I don't really understand the second sentence, but thats ok with me (the rest is crystal-clear).

The section about naming is good! This problem is under-evaluated but always an issue for developers.

Typo: "As a brief introduction, RDF2Gomaps to Java as follows:" ...RDF2Go maps...

Section "Related Work" You cite related work just above the section header and below, inside the section, that is awkward to read.

Just above "Related Work": "Internally, RDFReactor relies on Jena 2.3 (accessed through RDF2Go) in the code generator, which is explained in the next sections." – uhm - you meant in the above sections? (the next section does not contain that, or?)

"Related Work" "Rdf2Java generates Java classes and stores all state at runtime in member variables." That was so years ago, the latest Rdf2Java does store the data in Jena Models (background) and does it like RDFReactor (Setting/getting directly). So give credit there. On the other hand, RDF2Java does not do the inverse like you did and does not use Velocity nor a templating engine (but thats nitpicking), and its not cross-RDF framework compatible like RDF2Go.

"Evaluation" That is not an Evaluation in the scientific sense (to my knowledge) You might us all spare the trouble of discussing it by just removing this section. Also, after reading the paper, I can easily guess that the code would look like that.

You work is good, you don't have to say this using this evaluation arumentation. Comparing the lines-of-code of direct graph interaction (addStatement(), removeStatement(), max.addProperty(FOAF.name, "MAx")) to rdfreactor might be an argument, though. Sure, evaluating RDFReactor is not easy and I understand that you did better than the state-of-the-art a slight tip may be: <http://www-cse.ucsd.edu/wgg/CSE210/howtoread.html>

"What is the evaluation of the proposed solution? An idea alone is usually not adequate for publication of a research paper. This is the concrete engagement of the research question. What argument, implementation, and/or experiment makes the case for the value of the ideas? What benefits or problems are identified? Are they convincing? For work that has practical implications, you also want to ask: Is this really going to work, who would want it, what it will take to give it to them, and when might it become a reality?" So - you fulfill most of these arguments, perhaps just say that.

"Code & Ontology appendices" For a paper, this is not needed. You explained all above and this is (for the final version) a waste of trees. Please cut it, your work is fine without it.

btw: you got a broken link on your homepage: <http://rdfreactor.ontoware.org/> to <http://rdfreactor.ontoware.org/javadoc-example>

btw: good to include acknowledgements and good to include Heiko, whom you used in your beginning examples :-)

### A.3 Reviewer 21

#### A.3.0.6 Overall assesment.

4

#### A.3.0.7 Comments to the authors.

**Content** *E.g. Are the goals of the project or demo clear? Is the maturity of the work explained? Have key design choices been characterised and explored? Has the work been robustly assessed?:*

The system is epxlained well and clearly, and its goals made clear. I would have liked to know more about whether this system has been deployed. How widely is it used? Has it been used on any realistic applications?