RDFReactor – From Ontologies to Programmatic Data Access *

Max Voelkel and York Sure
Institute AIFB, University of Karlsruhe, Germany
{mvo,ysu}@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 proxy objects, thus allowing them to stay in their own world. The proxy objects are implemented as dynamic proxies and map method calls on the fly to 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 available for download 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[1]).

*This research was partially supported by the European Commission under contract FP6-507482. The expressed content is the view of the authors but not necessarily the view of the Knowledge Web Network of Excellence as a whole.

[1] `http://www.w3.org/RDF/`

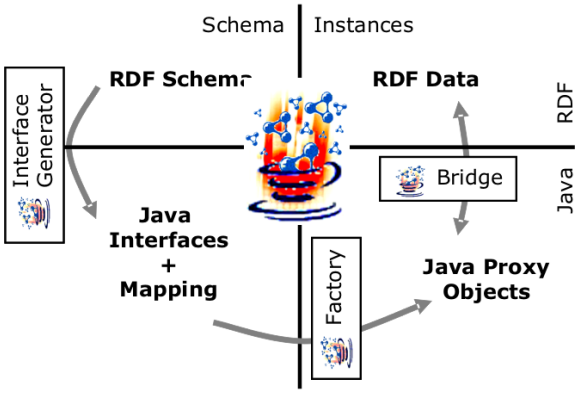


Figure 1: Mapping the two worlds

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 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 Design

We distinguish two phases in application development: *mapping* and *development*. The architecture reflects this by consisting of two independent components. An overview of the system can be found in Fig. 1.

2.1 Interface Generator

At mapping time, the developer creates or, preferably, reuses an RDF Schema as the backbone of her application. The *InterfaceGenerator* transforms an RDF Schema into a set of

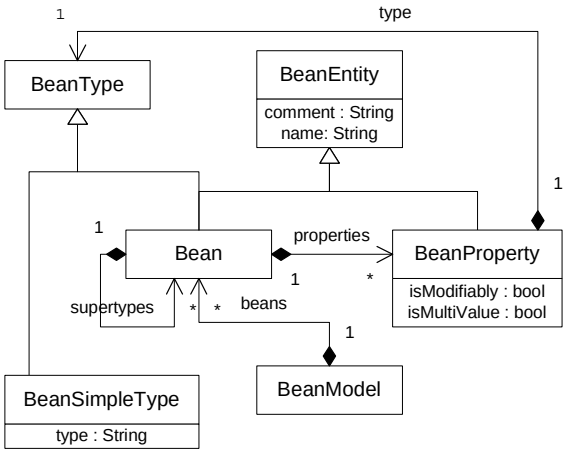


Figure 2: The Bean Model

Java interfaces and a mapping ϕ, which links generated Java interfaces and their methods to RDF URIs. The mapping is explained more detailed below.

Java Bean Properties and the Bean Model. The Java programming language models states and activities, but RDF only models states. Therefore we have to define a suitable data-manipulation-centric subset of possible Java-based APIs. In our work we choose to use the common notion of a *Java Bean Property*, that is a typed field in a class, which can only be accessed through methods, typically `get` and `set`. By abstracting this subset of Java, we created the Java Bean Metamodel (c. f. Fig. 2). Beans map to Java interfaces. `BeanProperty` elements are mapped to a set of type-safe Java methods, according to their `isModifiable` and `isMultiValue` attributes. These methods give access to the set of stored RDF properties either as Java primitive types, e.g. `String getName()`, or as Java class types, e.g. `Person[] getAllKnows()`.

Mapping RDF Schema to the Bean Model. Overall, we map RDF instances to Java instances, RDFS classes to Java interfaces and RDFS Properties to Java Bean properties. Multiple inheritance is legal for Java interfaces. Cycles in the `subClassOf`-graph are checked. RDF-resources can have multiple types, which causes no problem as the dynamic Java proxy instances (`java.lang.reflect.Proxy`) can implement an arbitrary number of interfaces. Dynamic proxies were introduced in the JDK 1.3 and work internally by creating class files on the fly which are then instantly loaded into the virtual machine.

The Full Mapping. Multiple Java artefacts (interfaces, bean properties) can be mapped to the same RDF artefact (class, property). Calling different methods with the same mapping then simple has the same effect. The mapping is stored as a Java property file. Generated Interfaces and the property file can be altered in any way, as long as the naming convention for method names is fulfilled: `T get`X, `T getAll`X, `set`X`(T x)`, `add`X`(T x)`, `remove`X`(T x)`, where `T` is the type of the Java property. Additionally, RDFReactor requires every Java class and all of its bean properties to appear in the mapping.

2.2 Bridge and Factory

At runtime, implementations are provided by a type-safe *ReactorFactory*. All method invocations on Java instances are channelled through the "Bridge" which analyses method names and types, looks up the URIs of classes and properties in the mapping and translates the method call into RDF operations. Changes in the RDF data model are thus reflected back instantly in the Java instances without any effort of the developer. Concurrent usage of the same data is also no problem. Internally, RDFReactor uses Jena[2] for processing of RDF/S.

3 Related Work & Conclusion

Current approaches like *OntoJava* [1], *Rdf2Java* [3], and *OWL2Java* [2] generate source code for Java classes and hold state at runtime in Java instances. They 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.

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 it's dynamic nature always in-sync with the RDF data model. This inherently allows for concurrent access to the RDF data model. Each interface inherits from `RDFBase`, which allows the developer to manipulate arbitrary RDF properties directly (`setProp( URI propURI, ... value )`. Thus we do not restrict the expressivity of RDF in any way. Additionally, RDFReactor's mapping is fully customisable, thus method names can be chosen manually, if desired.

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 available for download at `http://rdfreactor.ontoware.org`.

References

[1] A. Eberhart. Automatic generation of java/sql based inference engines from rdf schema and ruleml. In *Proc. of ISWC2002*, volume 2342 of *LNCS*, 01 2002.

[2] A. Kalyanpur et al. Automatic mapping of OWL ontologies into Java. In *Proc. of SEKE2004*, 2004.

[2] `http://jena.sourceforge.net/`
[3] `http://rdf2Java.opendfki.de`

A Demo

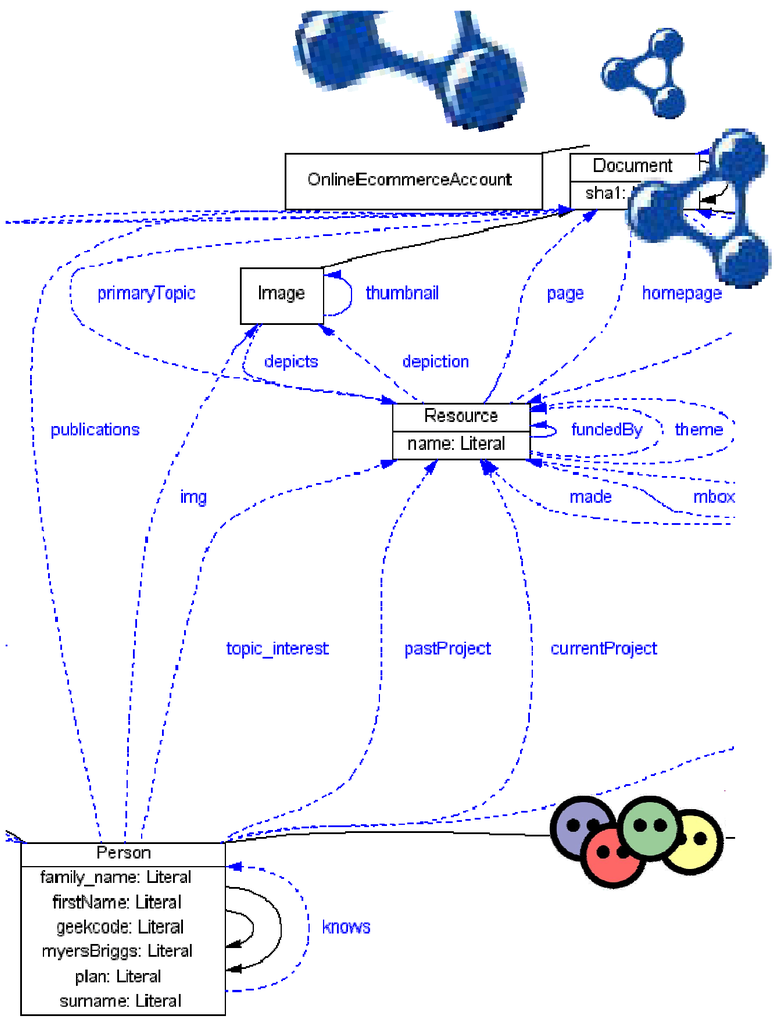
In the demo we will first show how the FOAF[4] RDF Schema is transformed into interfaces on the fly. We then customise the interfaces to our needs. Then we write a little example program against the generated API. After execution of the program we show the generated RDF data.

Additionally, we opt to transform any RDF schema suggested by the audience ad-hoc into an API. Writing a minimal example program also takes only 2 minutes.

For reference we add an example program in the following.

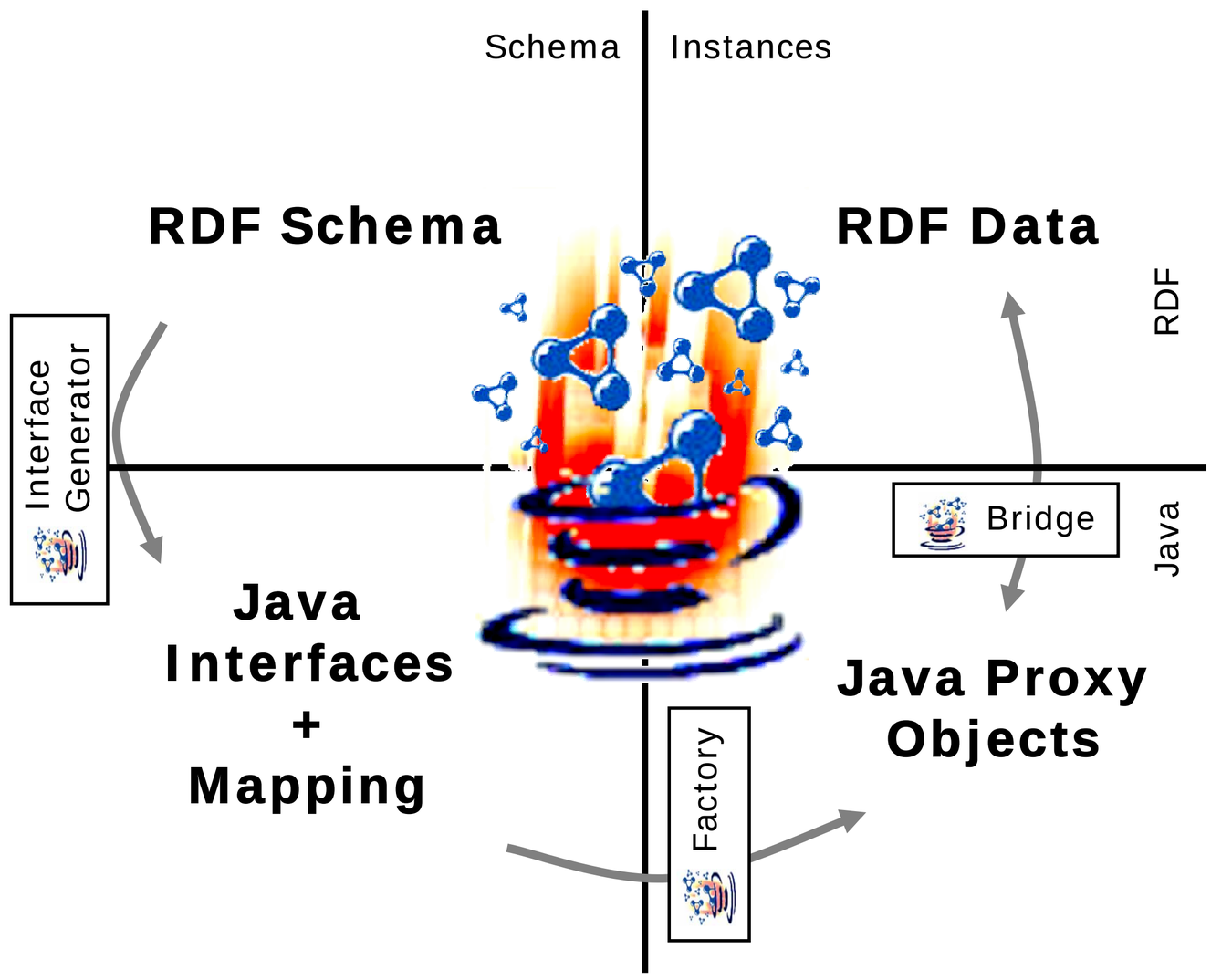
```
public static void main(String[] args) {

    // set up
    Model model = ModelFactory.createDefaultModel();
    ReactorFactory factory = new ReactorFactory(model);

    // demo: use generated API
    Person max = factory.createPerson();
    max.setFirstName("Max");
    max.setFamily_name("Voelkel");
    max.setMBoxHashed("25ab1214....67700");
    max.setNick("xamde");
    max.setHomepage("http://www.xam.de");
    max.setPhone("tel:+49-171-8359678");

    Person denny = factory.createPerson();
    denny.setName("Denny Vrandecic");
    denny.setMBoxHashed("4789fb144...fea3d12");
    denny
            .setSeeAlso("http://www.aifb.uni-karlsruhe.de/
            Personen/viewPersonenglish?id_db=2097");
    max.addKnows(denny);

    Person pascal = factory.createPerson();
    pascal.setName("Dr. Pascal Hitzler");
    max.addKnows(pascal);

    //output RDF data
    model.write(System.out,"N3");

}
```

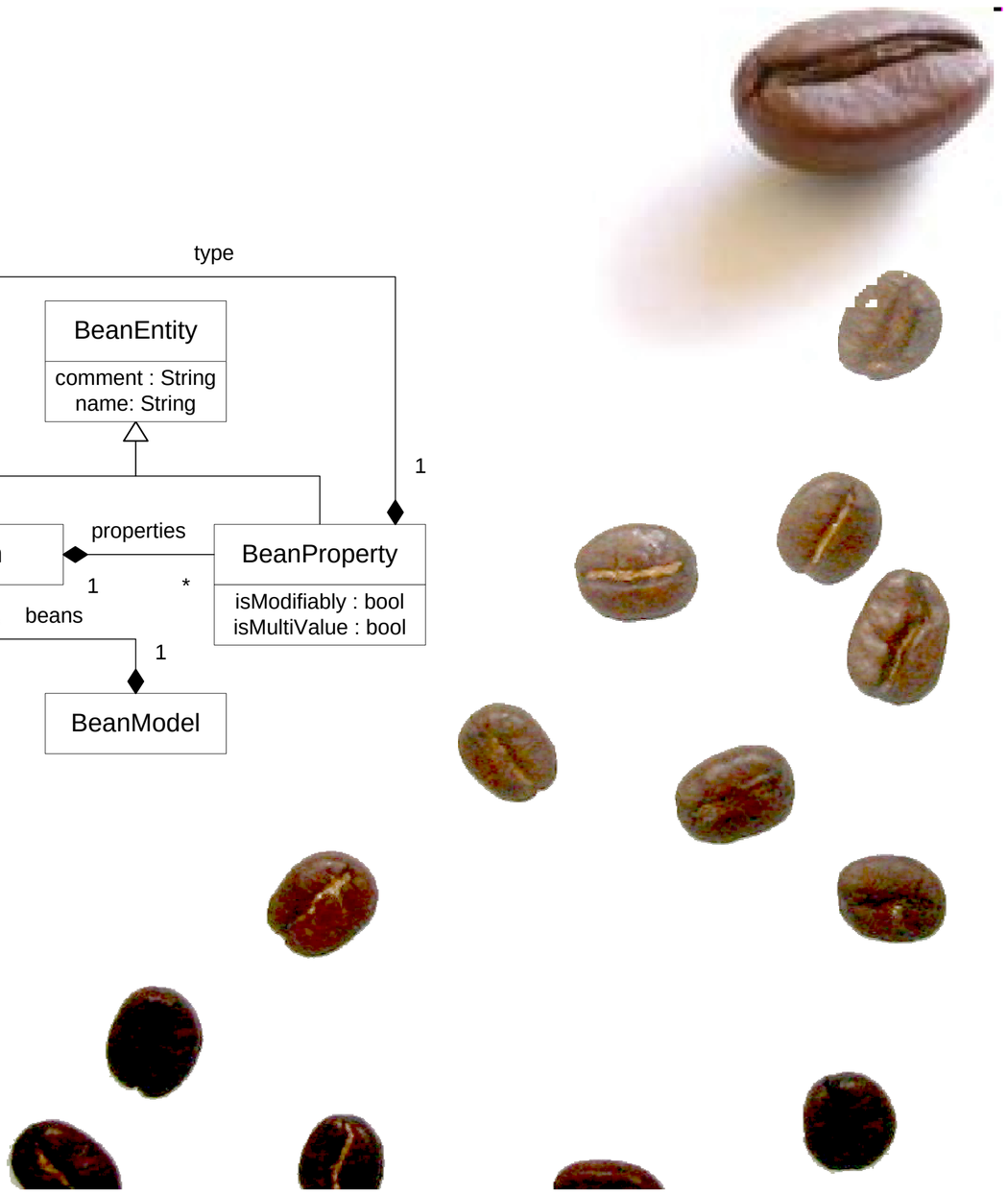
[4] http://www.foaf-project.org/



RDFReactor - it‘s that simple

A nive Java API for any RDF Schema at the push of a button…





```
<rdf:RDF
      xmlns:rdf="http://w                    rdf-syn
      xmlns:rdfs="http://                    df-schem
      xmlns:foaf="http://
      xmlns:admin="http:/
<foaf:PersonalProfileDocu
  <foaf:maker rdf:nodeID=
  <foaf:primaryTopic rdf:
  <admin:generatorAgent r                    www.ldoc
  <admin:errorReportsTo r                    leigh@lc
</foaf:PersonalProfileDocument>
<foaf:Person rdf:nodeID="me">
<foaf:name>Max Völkel</foaf:name>
<foaf:givenname>Max</foaf:givenname>
<foaf:family_name>Völkel</foaf:family_na
<foaf:nick>xamde</foaf:nick>
<foaf:mbox_sha1sum>25ab1214189e5dd8c7bc64beede8
a1sum>
<foaf:homepage rdf:resource="http://www.
<foaf:phone rdf:resource="tel:+49-171-83
<foaf:knows><foaf:Person>
<foaf:name>Denny Vrandecic</foaf:name>
<foaf:mbox_sha1sum>4789fb144a4c139875bc066a6b7a
a1sum>
<rdfs:seeAlso rdf:resource="http://www.aifb.
karlsruhe.de/Personen/viewPersonenglish?id_db=2097"/
</foaf:Person></foaf:knows>
  <foaf:knows><foaf:Person>
    <foaf:name>Dr. Pascal Hitzler</foaf:name
      <foaf:mbox_sha1sum>45c8cbd51ea3e90f3296a5e8dc6
        <rdfs:seeAlso rdf:resource="http://www.aifb.
          Personen/viewPersonenglish?id_db=2084"/
            n></foaf:knows>
              >
```

```
public static void main(String[] args) {
  Model model =
      ModelFactory.createDefaultModel();
  ReactorFactory factory =
      new ReactorFactory(model);
  Person max = factory.createPerson();
  max.setFirstName("Max");
  max.setFamily_name("Völkel");
  max.setMBoxHashed("25ab1214....67700");
  max.setNick("xamde");
  max.setHomepage("http://www.xam.de");
  max.setPhone("tel:+49-171-8359678");

  Person denny = factory.createPerson();
  denny.setName("Denny Vrandecic");
  denny.setMBoxHashed("4789fb144...fea3d12");
  denny.setSeeAlso("http://www.aifb.uni-
karlsruhe.de/Personen/viewPersonenglish?id_db=2
097");
  max.addKnows(denny);

  Person pascal = factory.createPerson();
  pascal.setName("Dr. Pascal Hitzler");
  max.addKnows(pascal);
}
```