Ontology-Driven Application Architectures with KOMMA

Ken Wenzel

Fraunhofer-Institute for Machine Tools and Forming Technology IWU,
Department for Production Planning and Resource Management,
Reichenhainer Str. 88, 09126 Chemnitz, Germany
ken.wenzel@iwu.fraunhofer.de

Abstract. The Knowledge Modeling and Management Architecture is an application framework for Java software systems based on Semantic Web technologies. KOMMA supports all application layers by providing solutions for persistence with object triple mapping, management of ontologies using named graphs as well as domain-specific visualization and editing of RDF data. This paper gives a short introduction to KOMMA's architecture and illustrates its usage with examples.

Keywords: application framework, eclipse, rdf, semantic web, object triple mapping, model-driven architecture

1 Introduction

Semantic Web technologies have gained popularity in the fields of information integration and semantic search. Therefore research and development activities mainly focused on ontologies, ontology editors, storage and inference solutions for RDF data as well as on generic solutions for presenting, navigating and editing RDF data sets.

We argue that the wide adoption of Semantic Web technologies can be accelerated by providing sophisticated application frameworks that lower the barriers for the development of RDF and OWL-based software systems.

Existing frameworks for model-driven engineering (MDE) like the Eclipse Modeling Framework (EMF)[1] simplify the creation of model-based applications by providing tools that support the required transformation tasks. The EMF project shows that there is large industrial request for model-based application frameworks. Over recent years this project was able to acquire a large community that actively uses and develops a variety of solutions around the EMF core components.

Because knowledge representation languages like RDFS or OWL can also be regarded as some type of modeling language in the sense of MDE, we've started the development of KOMMA by taking EMF as an example.

[1] http://www.eclipse.org/modeling/emf/

The goal of KOMMA is to combine RDF persistence providers with a sophisticated solution for object triple mapping and EMF-like flexible visualization and editing capabilities into a unified application framework for RDF- and ontology-based software systems.

2 Architecture and object model

KOMMA uses a layered architecture as depicted in Figure 1 to improve the reusability of its components.

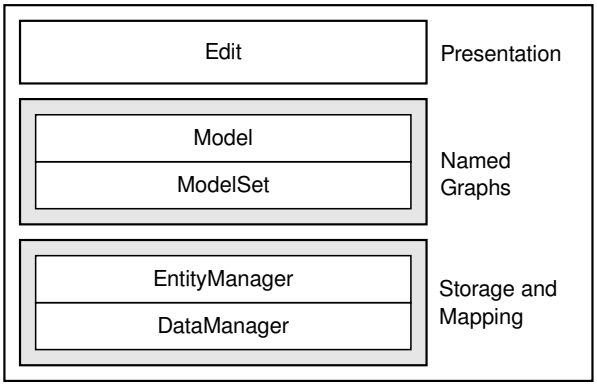

Fig. 1. Architecture overview.

The storage and mapping layer (Section 3) defines a small interface for pluggable persistence providers and implements an advanced solution for the mapping between Java objects and RDF triples. It is ontology language agnostic to ensure its reusability for different types of RDF based systems no matter which kind of ontology language, if any, is used. Currently, a persistence provider plugin for OpenRDF[2] Sesame is bundled with KOMMA but other triple stores can be easily added by implementing the `IDataManager` interface.

The next layer on the stack deals with RDF named graphs (Section 4). It provides models to manage OWL ontologies along with their mutual dependencies and associated software components. In its current implementation the data of each ontology is mapped to one model that itself is contained in a set of interrelated models. Consequently, a model set contains these models that constitute the imports closure of all contained ontologies.

The last layer implements mechanisms for the presentation and editing of RDF resources. It uses the adapter pattern to create views and editors for RDF resources and already provides many generic implementations and base classes to quickly create editable lists, tables or tree views.

[2] http://www.openrdf.org/

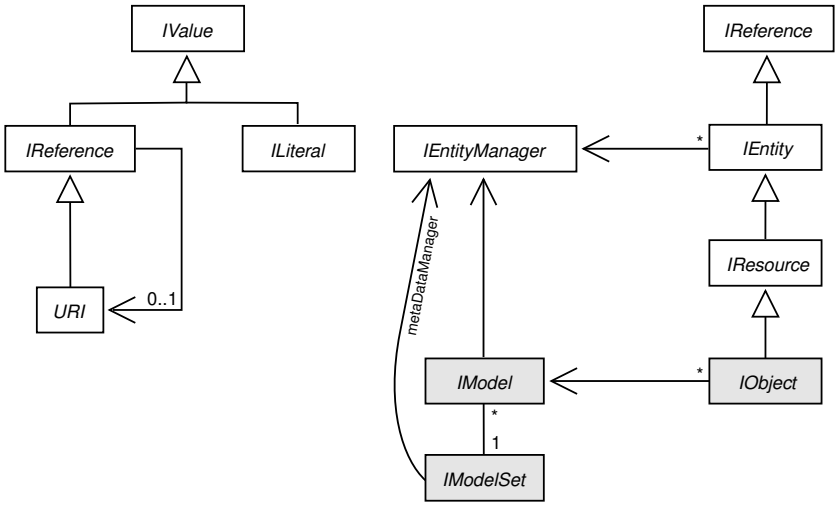

Fig. 2. Core object model.

Figure 2 shows KOMMA's core object model. All objects representing RDF nodes implement the interface `IValue`. RDF resources are modeled by objects of type `IReference` while `ILiteral` is used denote RDF literals. In the case of named nodes an `IReference` is associated to a corresponding `URI`, for blank nodes this `URI` is `null`.

The interfaces `URI` and `IEntity` are defined as specializations of `IReference`. This creates a unified class hierarchy where mapped objects of type `IEntity` can be used interchangeably with basic data-level references (`URI` for named nodes, `IReference` for blank nodes) to RDF nodes.

Java objects representing RDF resources are managed by an `IEntityManager` which is responsible for object triple mapping as described in Section 3.

`IResource` is a subject-oriented interface for reflective access to properties of RDF nodes. Among other things, it can be used for determining the direct RDF types of a resource or for adding and removing of property values.

The shaded classes in Figure 2 are core components of KOMMA's subsystem for the management of RDF named graphs and OWL ontologies which is covered by Section 4.

3 Object triple mapping

KOMMA implements object triple mapping by using a role-based composition approach that is partially based on the work done for the OpenRDF projects Elmo and AliBaba. Figure 3 shows the main components of a KOMMA application where modules are used to group related roles (interfaces and behaviour classes) for object triple mapping.

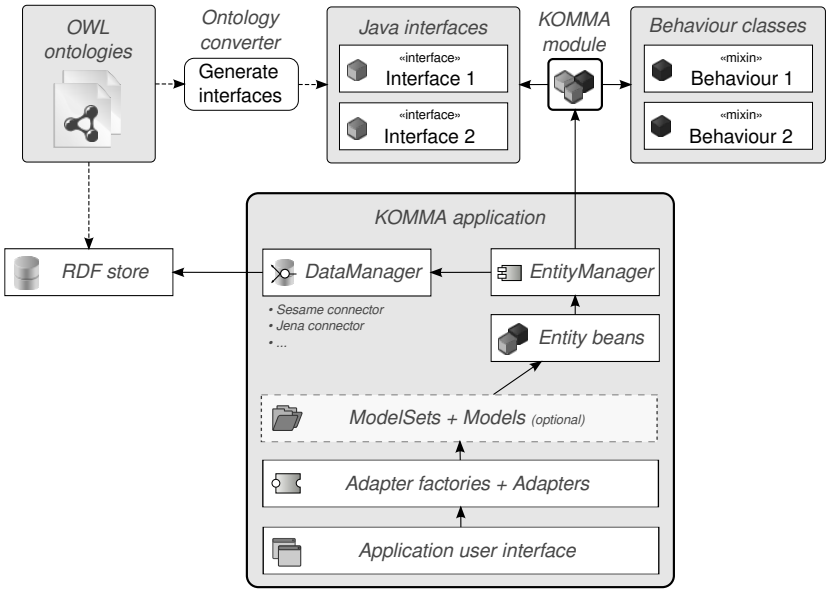

Fig. 3. Development workflow and basic application structure.

3.1 Mapping of RDF types to Java classes

RDF resources may have multiple associated types while Java supports only single inheritance. This fact imposes problems for the mapping of RDF resources to Java objects. One solution to circumvent this issue is by using the adapter pattern to create multiple views for a Java object that reflect the different types of the underlying RDF resource. This approach is, for example, implemented by Jena's[3] ontology API.

KOMMA follows another approach by using an engine that is able to combine multiple interfaces and behaviour classes into one class representing the union of multiple RDF types. These bean classes are generated dynamically at runtime of the application.

Interfaces and classes can be mapped to RDF types by using the `@iri` annotation that expects an absolute IRI to associate its target with an RDF resource.

When KOMMA needs to create a new Java class to represent an RDF resource as Java object, it retrieves its RDF types and determines all associated Java interfaces and classes. These are then combined into one class which embodies the resource's types.

Figure 4 shows an example of mappings for the class `Pet` and its subtypes `Cat` and `Dog`. Each pet has the ability to `yell`. Since cats and dogs usually make different sounds, the method `yell` is implemented according to Figure 5 for cats by `CatSupport` and for dogs by `DogSupport`.

[3] http://jena.sourceforge.net/

```
@iri("http://example.org/pets#Pet")
public interface Pet {
  @iri("http://example.org/pets#name")
  String getName();
  void setName(String name);

  @iri("http://example.org/pets#parent")
  Set<Pet> getParents();
  void setParents(Set<Pet> parents);

  void yell();
}

@iri("http://example.org/pets#Cat")
public interface Cat extends Pet {}

@iri("http://example.org/pets#Dog")
public interface Dog extends Pet {}
```

Fig. 4. Mapping of RDF types to Java interfaces.

A Java class representing an RDF resource with both types `Cat` and `Dog` (e.g. the famous CatDog[4]) would get a `yell` method that chains both implementations of `CatSupport` and `DogSupport`. The order of chained calls can be defined by establishing a hierarchy between behaviour classes by using the annotation `@precedes`.

```
abstract public class CatSupport implements Cat {
  public void yell() {
    System.out.println(getName() + " says MEOW");
  }
}

@precedes(CatSupport.class)
abstract public class DogSupport implements Dog {
  public void yell() {
    System.out.println(getName() + " says WUFF WUFF");
  }
}
```

Fig. 5. Extending mapped objects with user-defined behaviour.

[4] http://en.wikipedia.org/wiki/CatDog

3.2 Mapping of RDF properties to object methods

The example in Figure 4 contains mappings for the properties `name` and `parent` of the type `Pet`. Both mappings are defined by annotating the corresponding getter methods with an `@iri` annotation. If a setter method is also present then the property is treated as writable, else as read-only.

The mapping of properties is implemented by using `PropertySet` objects that encapsulate the required logic to convert between their RDF and Java representation. KOMMA contains a bytecode generator to implement interface methods annotated with `@iri`. This generator uses a pluggable factory for property sets to enable the usage of various storage back ends.

4 Named graphs and models

RDF named graphs are a means to structure RDF data sets. With named graphs it becomes possible to assign an IRI to a related set of RDF statements that can then be directly selected by its name within SPARQL queries.

KOMMA provides models and model sets to manage data contained in RDF named graphs in combination with associated software components in form of Java interfaces and behaviour classes.

Figure 6 shows two related models. Each model has its own module with interfaces and behaviour classes as well as an own entity manager. This manager is initialized with roles of the model's module and of all other modules contained in the model's imports closure. Hence, besides having read-only access to data from imported models, it is also able to use the imported roles for transformations of RDF nodes to Java objects. This enables the creation of ontology-driven architectures where composition of ontologies is accompanied by composition of related software components.

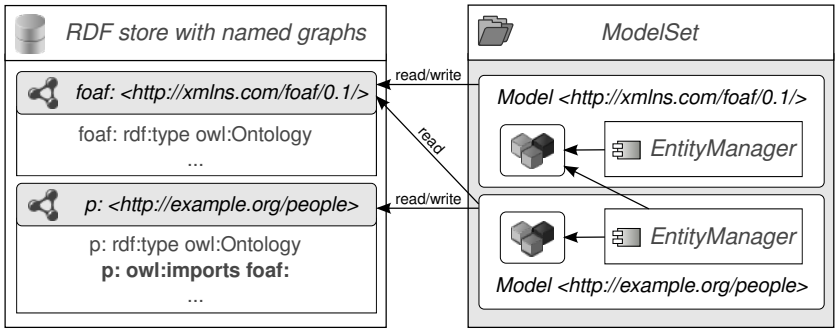

Fig. 6. Managing named graphs and associated software components with models.

Models implement the interface `IModel` that provides methods for loading the model's contents into a repository, for saving them to an external resource

and for deleting them from a repository. A model does also provide methods to access information about modification state or about diagnostics like warnings and errors.

Loading and saving models is done with the help of a so-called URI converter that uses a set of rules for resolving model URIs to physical locations in the local file system or on a remote server. After resolving the correct location, a specialized URI handler for the encoded scheme (e.g. http, ftp or file) is responsible for accessing the corresponding resource.

An `IModelSet` is a group of multiple interrelated models that is backed by an RDF repository where each model is stored in its own context (named graph).

Figure 2 reveals that each `IModelSet` has an associated `metaDataManager`. The reason is that KOMMA implements models and model sets by using its own object triple mapping mechanism. This results in the ability to handle these objects in the same way as all other RDF resources and enables the usage of SPARQL to access metadata about models and model sets.

5 Editing framework and adapter factories

KOMMA includes an Eclipse framework comprising generic reusable classes for building editors for RDF data models. This framework is strongly modeled after EMF.Edit, the editing framework for EMF.

Likewise, it simplifies the creation of GUI editors by providing

- content and label providers for JFace viewers,
- a command framework, including a set of generic commands for common editing operations as well as
- automatic undo and redo support for changes to models.

Adapters are used to implement tailored presentation and editing behaviour for different model elements. These adapters are created by factories that may support different adapter types. In contrast to EMF where each model object keeps track of its associated adapters, KOMMA transfers this obligation to the adapter factories. This decision was made to decouple the domain model from transient applications objects. The reason is that KOMMA's domain objects are rather lightweight references to RDF resources and may be instantiated multiple times while EMF's domain objects usually exist only once in memory.

6 Using and supporting KOMMA

KOMMA is developed as open source software, licensed under the terms of the Eclipse Public License (EPL)[5]. The public community site for KOMMA is hosted at `komma.sourceforge.net`. This is the entry point to public forums, a bug tracker as well as KOMMA's source code repositories.

Our code review system at `git.iwu.fraunhofer.de` enables users to participate in KOMMA's development by submitting patches for bugs or features.

[5] http://www.opensource.org/licenses/eclipse-1.0.php

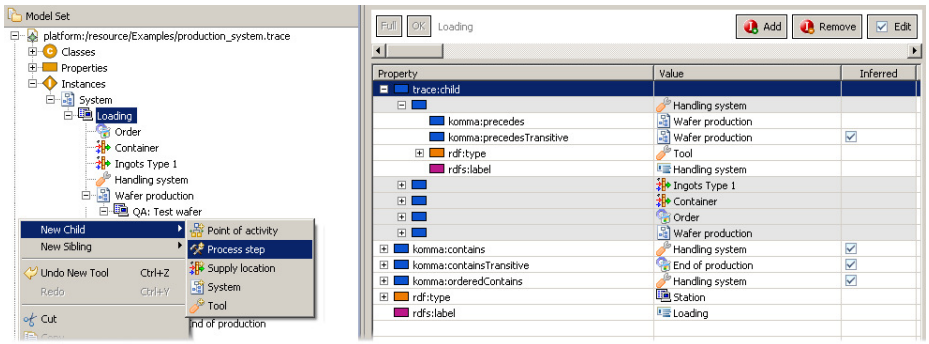

Fig. 7. Editing framework in action.

7 Conclusions and future work

KOMMA complements existing model-driven application frameworks by providing an integrated architecture for RDF and ontology-based software systems. Therefore, it is able to reduce the effort for the creation of semantic desktop and internet applications.

Future developments of KOMMA include the extension of its editing framework towards more advanced domain-specific graphical and textual presentations of RDF data as well as seamless support of rich client and rich internet applications. We hope that these activities will improve KOMMA's applicability to a wide range of real-world usage scenarios.

KOMMA already implements required core features to support innovative software design paradigms like *Data, Context and Interaction (DCI)*[6]. Further research could investigate how KOMMA's role-based object triple mapping in combination with its notion of models can be used to implement context-based policies for user interaction to accomplish goals of the DCI principle.

8 Acknowledgements

The work presented in this paper is co-funded within the Cluster of Excellence *Energy-Efficient Product and Process Innovation in Production Engineering* (eniPROD®) by the European Union (European Regional Development Fund) and the Free State of Saxony.

[6] http://www.artima.com/articles/dci_vision.html