openArchitectureWare 4.1
An introduction

Markus Voelter, www.voelter.de,

openArchitectureWare (oAW) is a suite of tools and components assisting with model driven software development, more precisely it is a "tool for building MDSD/MDA tools". It is built upon a modular MDSD generator framework and is implemented in Java. The suite supports arbitrary import (model) formats, meta models, and output (code) formats, especially Eclipse EMF.

The tooling (such as editors and model browsers) is based on the Eclipse platform. The core of oAW comprises of a workflow engine that allows the user to define transformation workflows as well as a number of prebuilt workflow components that can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code.

openArchitectureWare [oAW] is open source and part of the Eclipse GMT [GMT] project.

What is Model-Driven Software Development

This section is a *very* brief introduction to Model-Driven Software Development (MDSD). You can find more details for example in [MDSD].

MDSD is about making software development more domain-related as opposed to computing-related. It is also about making software development in a certain domain more efficient. The idea is to use domain-specific languages (DSLs) to represent domain knowledge in a precise and concise form. These representations are “sentences” written in the DSL and are typically called models. In order to make these models useful, they are typically transformed into executable artefacts, often source code for a particular platform.

There are a number of reasons why you would want to use MDSD in a development project:

- You want to provide a way for your domain experts to formally specify their knowledge, and to provide a way for your technology people to define how this is implemented (using model transformations and code generation).
- You might want to provide different implementations for the same model, perhaps because you want to run it on different platforms (.NET, Java, CORBA).
- You may want to capture knowledge about the domain, the technology, and their mapping in a clear, uncluttered format.

- In general, you don't want to bother with implementation details when specifying your functionality.
- Another reason for using MDSD: You are working in the context of product lines and software system families and need to develop domain-specific assets.
- Finally, a very good reason for using MDSD is to scale architecture to larger development teams (see also [MV05]).

You'll gain the biggest benefits from MDSD is you don't just use some kind of generator that you downloaded from the web, but rather, if you build your own generator. Now, what do I mean by this? What I recommend is the following:

- You should understand the domain for which you want to generate code, you should build your own meta models that accurately represent your domain in a formal manner
- You should build your own transformations and code generation templates in order to "capture" expert knowledge wrt. to the target platform and to understand how the generated code works and how manually written aspects must be included

Of course, you should not start from scratch. openArchitectureWare is a framework/toolset that provides many basic features that can help you achieve the above tasks. Let's take a look at some of the details of openArchitectureWare by implementing a simple example.

Example Overview

The purpose of this tutorial is to illustrate code generation and some related features with openArchitectureWare. The process we're going to go through will start by defining a meta model (using EMF tooling), coming up with some example data, writing code generation templates, running the generator and finally adding some constraint checks and extensions (and you'll know what these terms mean as you read on).

The actual content of the example is rather trivial – we will generate Java classes following the Java Beans conventions. The model will contain entities (such as *Person* or *Vehicle*) including some attributes and relationships among them – a rather typical data model. From these entities in the model we want to generate the Beans for implementation in Java. In a real setting, we might also want to generate persistence mappings, etc.

Please note that this article will show only a subset of the features of openArchitectureWare using an example that is so trivial that it's almost useless. At the end of this article we'll briefly mention some of the other features of oAW and point you to

additional things you can read. You can find the code for the this article in http://www.eclipse.org/gmt/oaw/samples/oaw41SamplesEMF.zip

Defining an EMF metamodel

EMF is short for Eclipse Modeling Framework and is the basis for all of the modeling tools on the Eclipse platform. It's main contribution is a meta meta model, i.e. a facility for defining domain-specific meta models (which then, in turn, serve as the type system for the models from which you'll then generate code).

To illustrate the example meta model before we deal with the intricacies of EMF figure 1 meta model nicely rendered in UML:

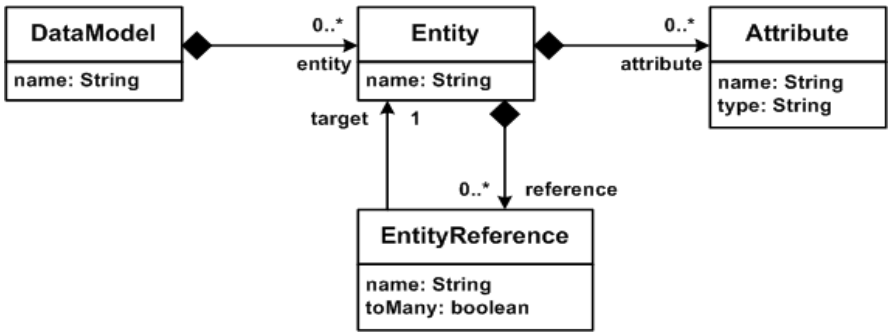

Figure 1: The meta model for our example rendered in UML

We'll now illustrate some of the steps necessary to implement the meta model in EMF. Note that we will not show each an every detail in this article. The reader is referred to the oAW documentation at [oAWDoc].

We start by defining an Eclipse project that contains the meta model. For EMF to work properly, it is best to make this an *Empty EMF project*. In this project, we'll now create an Ecore model using the respective wizard. Figure 2 shows the EMF-supplied tree editor that contains the tree representation of the meta model given above. Note that some of the relevant properties (such as the multiplicities of the references) are not visible, they can be changed via the *EMF Properties* view which is not shown here.

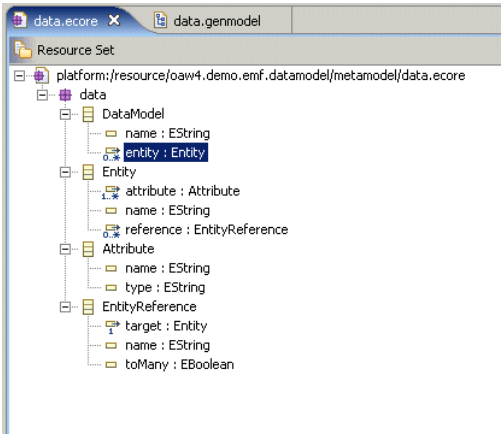

Figure 2: The meta model in the EMF tree view

Modeling an Example

The next natural step is to define an example model – technically, an instance of the meta model we've just defined. This raises the issue of how to actually model instances of that meta model: which concrete syntax do we want to use? And wich editor tool? There are a number of options here:

- We can use the tree editors which EMF provides out of the box. These are suitable for simple examples, but don't scale to real world scenarios.
- We can use a UML tool and define a profile to represent the concepts defined in the meta model (class diagrams obviously are a good candidate here). A model-to-model transformation would transform the profiled UML model into an instance of our meta model.
- We can generate a graphical editor (probably also using some kinds of boxes and lines) based on the Eclipse GMF framework.
- Finally, we can also come up with a textual notation for the models, and build nice editos using oAW's own Xtext framework.

To keep this example simple, we'll use the EMF provided tree editors. We need to press a couple of buttons in Eclipse EMF to generate those tree editors (it's really just five steps in a wizard, and you can see the details in the oAW docs, if you're interested). Once we did this, we have a (more or less) nice tree-based editor available for building an instance of the meta model. Figure 3 shows the sample model.

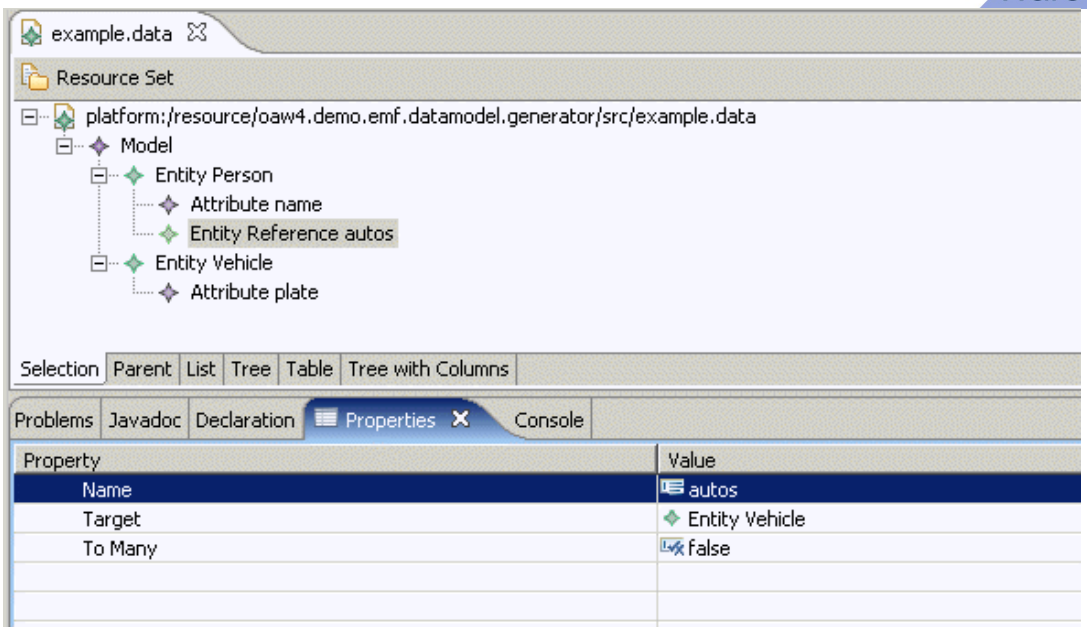

Figure 3: Example Model in the default Tree Editor

All the stuff you have seen up to now is Eclipse EMF and not openArchitectureWare. Let's now look at how to generate code from that model with oAW.

Generating Code From the Example Model

To run the openArchitectureWare generator you have to define a workflow. It controls which steps the generator executes (loading models, checking them, generating code). Let's create a workflow that only loads the example model we created above (or any other model that is an instance of the meta model defined above). The *workflow.oaw* file looks as follows:

```
<workflow>
	<component id="xmiParser"
			class="org.openarchitectureware.emf.XmiReader">
		<modelFile value="example.data"/>
		<outputSlot value="model"/>
		<firstElementOnly value="true"/>
	</component>
</workflow>
```

This workflow consists of only one workflow component, an *XmiReader*. This instantiates the example model and stores it in the workflow context in a slot named *model* (the workflow context is basically a hashmap available to all workflow components, the slots being the keys into the hashmap). Note that in real project you would property files to make the model file name configurable.

You can run the workflow using the *Run As ...* menu in Eclipse (alternatively you can run workflows from your own custom Java code, ant or a simple batch file, without Eclipse!). Of course, nothing spectacular happens except for some log output. We didn't write any code generation templates yet. Let's do that now.

Figure 4 shows the oAW template editor. OAW comes with its own template language – called *xPand* – including editors that provide code completion, syntax highlighting and static error checking.

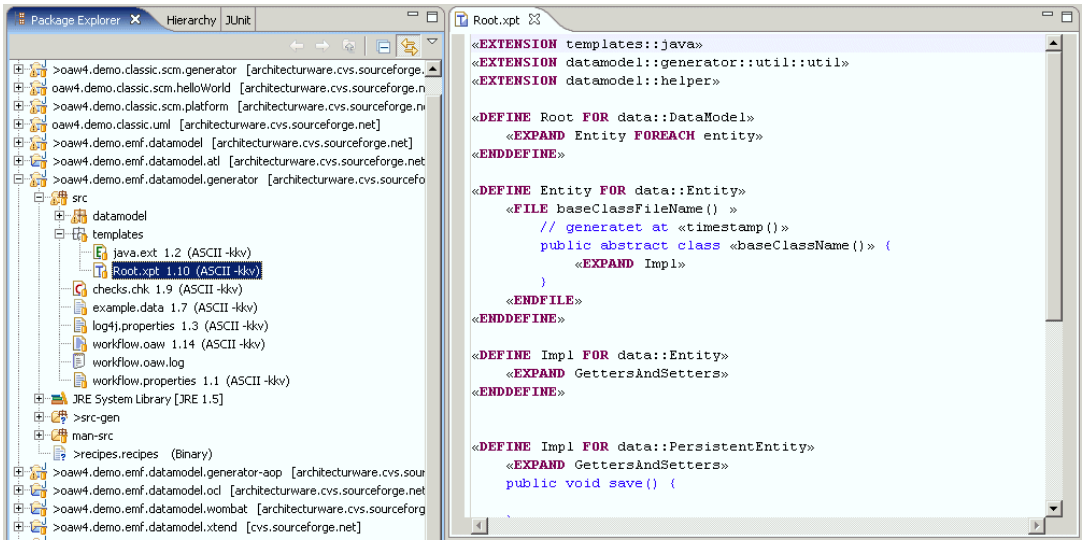

Figure 4: oAW's template editor

Let's look a bit more closely at the templates. We start by defining a template called *Root* for the metaclass *DataModel* (as defined in the *data* metamodel above). Inside that template we iterate over all entities defined as part of that *DataModel* instance (technically, we iterate over the EReference *entity* that we defined in the meta model) and then call another template called *Entity* for each of the entities.

```
«DEFINE Root FOR data::DataModel»
	«EXPAND Entity FOREACH entity»
«ENDDEFINE»
```

In that other template, we open a file that is named just like the entity (with *.java* appended). The file contains a class declaration (again, named just like the entity); the class itself contains a private attribute for each of the Entity's attributes.

```
«DEFINE Entity FOR data::Entity»
	«FILE name+".java"»
		public class «name» {
			«FOREACH attribute AS a»
				private «a.type» «a.name»;
			«ENDFOREACH»
```

```
            }
      «ENDFILE»
«ENDDEFINE»
```

In order to actually generate code using these templates, we have to add an additional workflow component to the workflow definition, a *Generator* component.

```
<component id="generator"
            class="org.openarchitectureware.xpand2.Generator">
      <metaModel id="mm"
             class="org.openarchitectureware.type.emf.EmfMetaModel">
         <metaModelFile value="data.ecore"/>
       </metaModel>
      <expand value="templates::Root::Root FOR model"/>
      <genPath value="src-gen/"/>
      <beautifier
            class="org.openarchitectu...put.JavaBeautifier"/>

</component>
```

Let's look at that component declaration a little bit more closely. First of all, you have to specify the meta meta model. In our case we use the *EmfMetaModel* since we want to work with EMF models. Also, you have to specific the ecore file name of the meta model we want to use. Then you have to define the entry expression for the template engine. Knowing that the *model* slot contains an instance of *data::DataModel* (the *XMIReader* had put the first element of the model into that slot, and we know from the data that it's a *DataModel*), we can write the following statement. Notice that *model* refers to a slot name here!

We then specify where the generator should put the generated code (*genPath*). Finally, we specify a beautifier, to pretty-print the generated code:

If you now run the generator again, you should get a file generated that looks like this:

```
public class Person {
      public String lastName;
}
```

A similar file should be generated for the *Vehicle* model element in our model.

Checking Constraints

Constraints are important. A meta model captures only the structral aspects of a model, but there are almost always additional constraints that must hold. An example could be that the names of the attributes of an *Entity* must be distinct. You cannot express this with the structural means of an EMF meta model. It is, however, important that these constraints hold before you attempt to process the model. For example, there's no point in generating Java code as shown above if the name-uniqueness constraint does not hold, since the generated code will not compile.

So let's start by defining a simple constraint. We create new file called *checks.chk* in the *src* folder of our project. The file has the following content (note that oAW also provide syntax highlighting, code completion and static error checking for constraint check files) The language used for specifying the constraints is oAW's expression language which is similar to OCL in many respects.

```
import data;
context Attribute ERROR
   "Names must be more than one character long" :
   name.length > 1;
```

This constraint says that for the metaclass *data::Attribute*, we require that the name be longer than one character. If this expression evaluates to *false*, the error message given before the colon will be reported. A checks file can contain any number of such constraints. They will be evaluated for all instances of the respective metaclass. To show a somewhat more involved constraint example, this one ensures that the names of the attributes have to be unique:

```
context Entity ERROR "Names of Entity attributes must be unique":
      attribute.forAll(a1| attribute.notExists(a2| a1 != a2 &&
a1.name == a2.name ) );
```

To actually check the constraints, you have to add an additional step to the workflow. After reading the model, we add an additional component, namely a *CheckComponent*.

```
<component
          class="org.openarchitectureware.check.CheckComponent">
      <metaModel id="mm"
class="org.openarchitectureware.type.emf.EmfMetaModel">
        <metaModelFile value="data.ecore"/>
      </metaModel>
      <checkFile value="checks"/>
      <emfAllChildrenSlot value="model"/>
</component>
```

As with the code generator, we have to explain to the checker what meta meta model and which meta model we use. We then have to provide the checks file. The component tries to load the file by appending *.chk* to the name and searching the classpath. Finally, we have to tell the engine on which (part of) the model the checks should work. In general, you can use the <*expression value="..."*/> element to define an arbitrary expression on slot contents. For our purpose, where we want to use the complete EMF data structure in the *model* slot, we can use the shortcut *emfAllChildrenSlot* property, which returns the complete subtree below a specific slot's content element, including the slot content element itself.

Running the workflow produces an error in case the length of the name is not greater than one. Note that subsequent workflow components will not be executed if there are errors in the workflow – so no code generation will be attempted.

Extensions

In order to generate certain artifacts, you'll often need to access additional properties in the templates; these properties should not be added to the metaclasses directly, since they are often specific to the specific code generation target and thus should not „pollute“ the metamodel. It is possible to define such extensions external to the metaclasses using so-called extensions. Assume we wanted to change the *Attributes*-Part of the template as follows:

```
«FOREACH attribute AS a»
	private «a.type» «a.name»;

	public void «a.setterName()»( «a.type» value ) {
		this.«a.name» = value;
	}

	public «a.type» «a.getterName()»() {
		return this.«a.name»;
	}
«ENDFOREACH»
```

To make this work, we need to define the *setterName()* and *getterName()* operations. We do this by writing an extension file; we call it *java.ext*. It must have the *.ext* suffix to be recognized by oAW; the *java* name is because it contains Java-generation specific extensions. We put this file directly into the *templates* directory under *src*, i.e. next to the *Root.xpt* file. The extension file looks as follows.

```
import data;

String setterName(Attribute this) :
   'set'+name.toFirstUpper();

String getterName(Attribute this) :
   'get'+name.toFirstUpper();
```

First we have to import the *data* metamodel; otherwise we'd not be able to use the *Attribute* metaclass. We can then define the two new operations *setterName* and *getterName*. Note that they take the type on which they're called as their first parameter, a kind of „explicit this“. After the colon we use an expression that defines the result of the extension.

To be able to use the extension in our template, we have to add the following line to the beginning of the *Root.xpt* template file:

```
«EXTENSION templates::java»
```

In case you cannot express the „business logic“ for the expression with the oAW expression language you can fall back to so-called Java extensions, where you can escape to Java code to implement the expression.

This concludes our simple introduction of openArchitectureWare. Below you'll find links to further reading.

More features of oAW

We'd like to at least mention some of the advanced features that distinguish openArchitectureWare from some of its competitors, hence this short, incomplete feature list.

With a suitable instantiator, oAW can read any model. Currently, we provide out-of-the-box support for EMF, various UML tools (MagicDraw, Poseidon, Enterprise Architect, Rose, XDE, Innovator, …), the Eclipse UML2 project, textual models (using JavaCC or antlr parsers), XML, Visio as well as pure::variants variant configuration models.

oAW can read several different models during one generator run. These models can use different concrete syntaxes (one could be UML, another one pure EMF and a third one could be a pure::variants feature model). The generator then "weaves" these models together to form a comprehensive, all-encompassing model. Constraints can be checked over model boundaries; references between models can be established.

There are a number of additional important features in the template language that have not been illustrated in this article. Xpand supports template polymorphism, template aspects and many other advanced features that are necessary for building non-trivial code generators.

For realistically-sized meta models, you can build the meta model using a UML2-capable UML tool and create the Ecore meta model from that. This allows you to use the UML tool's model management features to keep an overview over your meta model. The transformer from UML2 to Ecore adds a number of additional features to the Ecore meta model, such as namespaces and qualified names.

Model-to-model transformations can be implemented using the xTend language, a textual, functional transformation language. This language has a number of very important characteristics. First, it has a very concise and powerful syntax. Second, it can be used to transform between the various meta meta models, e.g., you can transform a model defined with the oAW-classic metametamodel into an EMF model! Thirdly, a transformational function is only evaluated once for each unique combination of parameters. Thus, you can call the same function with the same number of arguments multiple times, and it will only be evaluated the first time! This is an indispensible feature when working with graph transformations, especially, if they contain circular references.

The Recipe framework allows you to define validation rules for artefacts created outside of the generator (such as manually written subclasses). During code generation, these rules can be instantiated; later, the Eclipse IDE will read these rules and verify them against the

code base. This will help to guide developers beyond the modelling/generation stage, when integrating generated and manually written code.

oAW also comes with an integration into the Eclipse GMF framework. Specifically, you can verify oAW constraints interactively in a GMF editor.

Finally, oAW provides a framework for building textual editors: xText. It provides a means of generating textual Eclipse editors from a BNF-like syntax description. The editor generator generates a parser component that can be used with the oAW workflow engine as well as an Ecore file that serves as the metamodel for the models edited with the editor. Also, an Eclipse editor is generated, that can be used to edit instances of the textual DSL. It also supports syntax highlighting and constraints checking.

Further Reading

There are two online resources you might want to check out: *From Frontend To Code* [FFTC] is a rather long article at Eclipse.org that shows many of the advanced features of oAW, including the integration with GMF. Another one, *The Pragmatic Generator Programmer* [TPGP] is an article that looks at implementing a simple textual DSL. Of course, if you want to learn about the details of oAW, you should take a look at the documentation page at [oAWDoc]... and play with oAW. Have fun!

References

FFTC	Voelter, Efftinge, Kolb, Haase, *From Frontend to Code*, http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html
GMT	The Generative Modeling Tools project, http://eclipse.org/gmt
MDSD	Stahl, Voelter, Model-Driven Software Development, Wiley, 2006
oAW	openArchitectureWare, http://www.eclipse.org/gmt/oaw
oawDoc	oAW Docs, http://www.eclipse.org/gmt/oaw/doc
TPGP	Voelter, Efftinge, Kolb, Haase, *The Pragmatic Generator Programmer*, http://www.theserverside.com/tt/articles/article.tss?l=PragmaticGen

About the Author

Markus Völter works as an independent consultant and coach for software technology and engineering. He focuses on software architecture, middleware as well as model-driven software development. Markus is the author of several magazine articles, patterns and books on middleware and model-driven software development. He is a regular speaker at

conferences world wide. He is also a core developer of openArchitectureWare. Markus can be reached at voelter@acm.org via or www.voelter.de