

openArchitectureWare 4.1 XTend Example

Markus Voelter, voelter@acm.org, www.voelter.de

Table of Contents

INSTALLING THE PRE-BUILT TUTORIAL.....	2
INTRODUCTION.....	2
WORKFLOW.....	2
THE TRANSFORMATION.....	3

Installing the pre-built tutorial

You need to have openArchitectureWare 4.1 installed. Please consider <http://www.eclipse.org/gmt/oaw/download> for details.

You can also install the code for the tutorial. It can be downloaded from the URL above, it is part of the the EMF samples ZIP file. Installing the demos is easy: Just add the projects to your workspace. Note that in the openArchitectureWare preferences (either globally for the workspace, or specific for the sample projects, you have to select *EMF metamodels* for these examples to work.

Note that there are more examples of using Xtend as a transformation language:

- One is in the Eclipse.org article *From Frontend To Code* at <http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html>
- The other one can be found in the EMF State Machine example

Introduction

This example uses Eclipse EMF as the basis for model-to-model transformations. It builds on the emfExample documented elsewhere. Please read and install the emfExample first.

The idea in this example is to transform the data model introduced in the EMF example into itself. This might seem boring, but the example is in fact quite illustrative.

Workflow

By now you should know the role and structure of workflow files. Therefore, the interesting aspect of the workflow file below is the *XtendComponent*.

```
<workflow>
  <property file="workflow.properties"/>
  ...
  <component class="oaw.xtend.XtendComponent">
    <metaModel class="oaw.type.emf.EmfMetaModel">
      <metaModelPackage value="data.DataPackage"/>
    </metaModel>
    <invoke value="test::Trafo::duplicate(rootElement)"/>
    <outputSlot value="newModel"/>
  </component>
```

...

</workflow>

As usual, we have to define the metamodel that should be used, and since we want to transform a data model into a data model, we need to specify only the *data.DataPackage* as the metamodel.

We then specify which function to invoke for the transformation. The statement *test::Trafo::duplicate(rootElement)* means to invoke

- the *duplicate* function taking the contents of the *rootElement* slot as a parameter
- the function can be found in the *Trafo.ext* file
- and that in turn is in the classpath, in the *test* package.

The Transformation

The transformation, as mentioned above, can be found in the *Trafo.ext* file in the *test* package in the *src* folder. Let's walk through the file.

So, first we import the metamodel.

```
import data;
```

The next function is a so-called *create* extension. Create extensions, as a sideeffect when called, create an instance of the type given after the *create* keyword. In our case, the *duplicate* function creates an instance of *DataModel*. This newly created object can be referred to in the transformation by *this* (which is why *this* is specified behind the type). Since *this* can be omitted, we don't have to mention it explicitly in the transformation.

The function also takes an instance of *DataModel* as its only parameter. That object is referred to in the transformation as *s*. So, this function sets the name of the newly created *DataModel* to be the name of the original one, and then adds duplicates of all entities of the original one to the new one. To create the duplicates of the entities, the *duplicate()* operation is called for each *Entity*. That is the next function in the transformation.

```
create DataModel this duplicate(DataModel s):  
    entity.addAll( s.entity.duplicate() ) ->  
    setName(s.name);
```

The duplication function for Entities is also a create Extension, this time it creates a new Entity for each old Entity passed in. Again, it copies the name and adds duplicates of the attributes and references to the new one.

```
create Entity this duplicate(Entity old):  
    attribute.addAll( old.attribute.duplicate() ) ->  
    reference.addAll( old.reference.duplicate() ) ->  
    setName( old.name );
```

The function that copies the attribute is rather straight forward, but ...

```
create Attribute this duplicate(Attribute old):  
    setName( old.name ) ->  
    setType( old.type );
```

... the one for the references is more interesting. Note that a reference, while being owned by some Entity, also references another Entity as its target. So how do you make sure you don't duplicate the target twice? Xtend provides explicit support for this kind of situation. *Create extensions are only executed once per tuple of parameters!* So if, for example, the Entity behind the *target* reference had already been duplicated by calling the duplicate function with the respective parameter, the next time it will be called *the exact same object will be returned*. This is very useful for graph transformations.

```
create EntityReference this duplicate(EntityReference old):  
    setName( old.name ) ->  
    setTarget( old.target.duplicate() );
```

For more information about the Xtend language please see the Xtend Reference documentation.