openArchitectureWare 4.1
EMF Example

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

Table of Contents

Introduction

This example uses Eclipse EMF as the basis for code generation. One of the essential new features of oAW4 is EMF support. While not all aspects of EMF as good and nice to use as one would wish, the large amount of available 3rd party tools makes EMF a good basis. Specifically, better tools for building EMF metamodels are on the horizon already. To get a deeper understanding of EMF, we recommend that you first read the EMF tutorial at

- http://www-128.ibm.com/developerworks/library/os-ecemf1/
- http://www-128.ibm.com/developerworks/library/os-ecemf2/
- http://www-128.ibm.com/developerworks/library/os-ecemf3/

You can also run the tutorial without completely understanding EMF, but the tutorial might "feel" unnecessarily complex.

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 downlaoded 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.

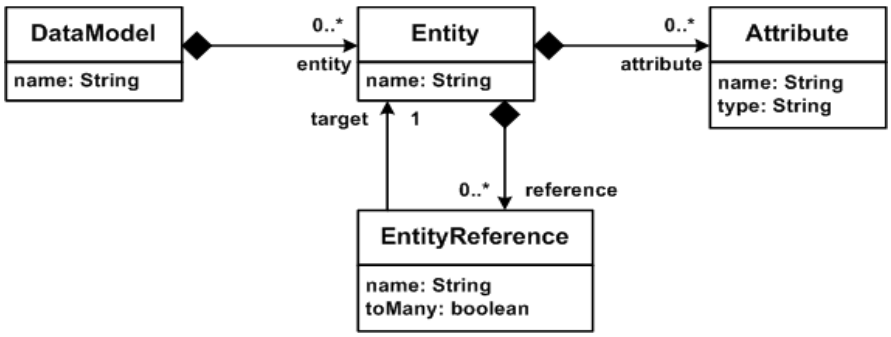
Example Overview

The purpose of this tutorial is to illustrate code generation with openArchitectureWare from EMF models. 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.

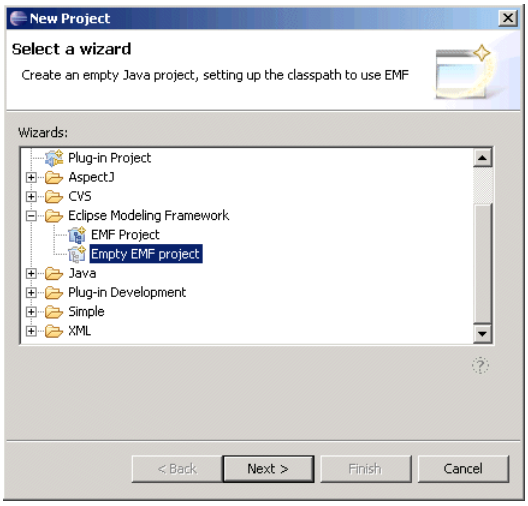
The actual content of the example is rather trivial – we will generate Java classes following the Java Beans convetions. The model will contain entities (such as *Person* or *Vehicle*) including some attributes and relationships among them – a tather typical data model. From these entities in the model we want to generate the Beans for implementation in Java. In a real setting, we might also want to generate persistence mappings, etc. We will not to this for this simple introduction.

Defining an EMF metamodel

To illustrate the metamodel before we deal with the intricacies of EMF here is the metamodel in UML:



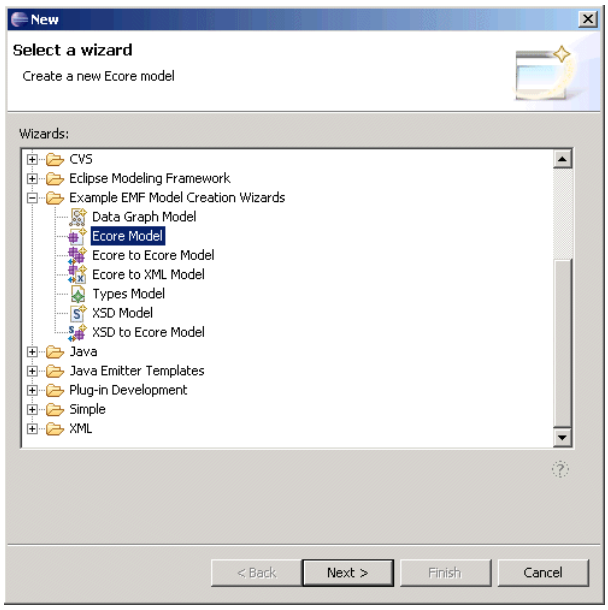
Create an EMF project



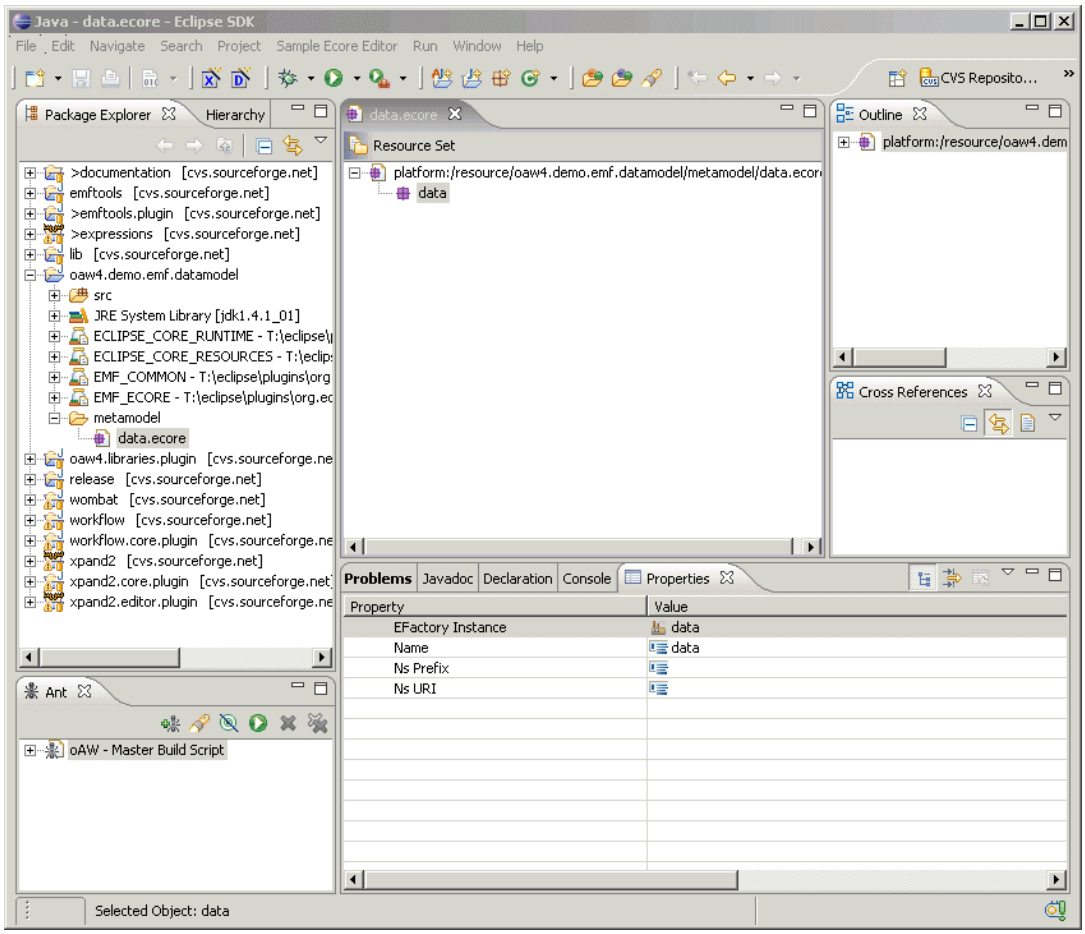
It is important that you create an EMF project, not just a simple or a Java project. name it *oaw4.demo.emf.datamodel*

Define the (meta)model

Create a folder *metamodel* in that project. Create a new Ecore model in that folder named *data.ecore*. Use *EPackage* as the model object.



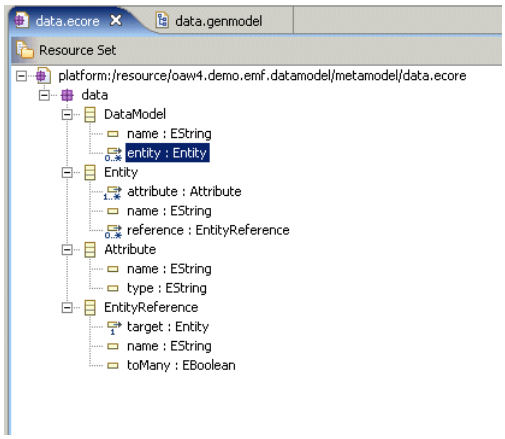
This opens the Ecore Editor. Name the package *data*.



Create the following Ecore model. Make sure you set the following properties as described next: (note: there are a couple of -1's ... don't miss the minus!)

- **DataModel::entity**: containment->true, LowerBound->0, UpperBound->-1
- **Entity**::attribute: containment->true, LowerBound->1, UpperBound->-1
- **Entity::reference**: containment->true, LowerBound->0, UpperBound->-1
- **EntityReference::target:** containment->false, LowerBound->1, UpperBound->1

To add children, right-click on the element to which you want to add children and select the type of the child from the list. To configure the properties, open the properties dialog by selecting *Show Properties View* at the bottom of any of the context menues. Note that this is not an EMF tutorial. For more details on how to build EMF (meta-)modes please refere to the EMF documentation.



Define the Ns Prefix to be *data* and the Ns URI to be *http://www.openarchitectureware.org/oaw4.demo.emf.datamodel*. It has to be defined on the toplevel model element (here: the *data* EPackage).

EMF saves the model we created above in its own dialect of XMI. To avoid any ambiguities, here is the complete XMI source for the metamodel. It goes into the file *data.ecore:*

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="data"
    nsURI="http://www.openarchitectureware.org/oaw4.demo.emf.datamodel" nsPrefix="data">
  <eClassifiers xsi:type="ecore:EClass" name="DataModel">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="entity" upperBound="-1"
        eType="#//Entity" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Entity">
    <eStructuralFeatures xsi:type="ecore:EReference" name="attribute" lowerBound="1"
        upperBound="-1" eType="#//Attribute" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="reference" upperBound="-1"
        eType="#//EntityReference" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Attribute">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

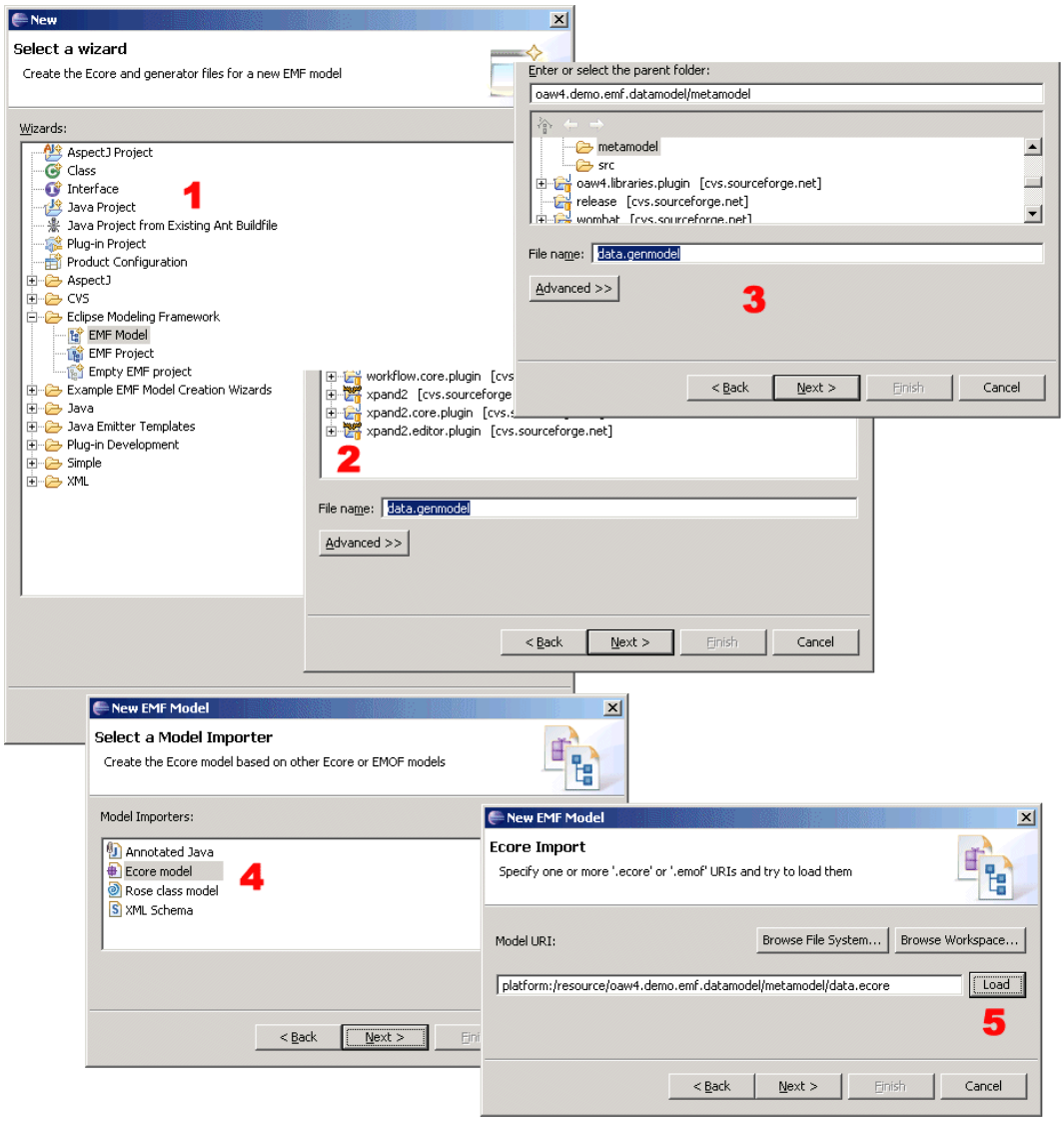
```
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="EntityReference">
    <eStructuralFeatures xsi:type="ecore:EReference" name="target" lowerBound="1"
        eType="#//Entity"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="toMany" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
  </eClassifiers>
</ecore:EPackage>
```

Generating the EMF tooling

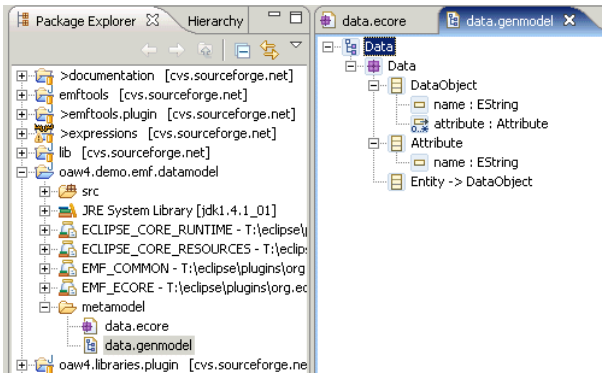
In addition to providing the Ecore meta meta model, EMF also comes with support for building (more or less usable) editors. These are generated automatically from the metamodel we just defined. In order to define example models (which we'll do below) we have to generate these editors. Also, we have to generate the implementation classes for our metamodel. To generate all these things, we have to define a markup model that contains a number of specifics to control the generation of the various artifacts. This markup model is called *genmodel*.

So we have to define the genmodel first. Select the *data.ecore* model in the explorer and right mouse click to *New->Other, Eclipse Modelling Framework::EMF Model*. Follow the following five steps; note that they are also illustrated in the next figure.

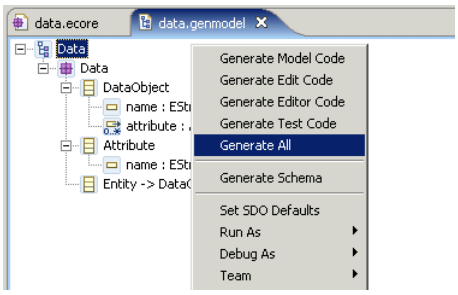
1. select EMF model
2. define the name
3. select the folder
4. select ecore model as source
5. press the load button and then finish



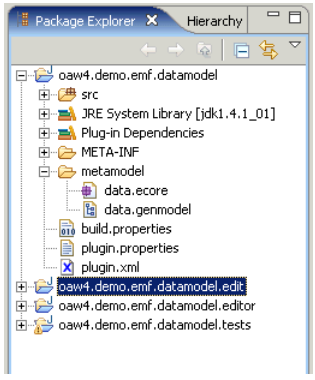
As a consequence you'll get the finished EMF genmodel. It is a kind of "wrapper" around the original metamodel; thus it has the same structure, but the model elements have different properties. As of now, you don't have to change any of these.



You can now generate the other projects



You now have all the generated additional projects.

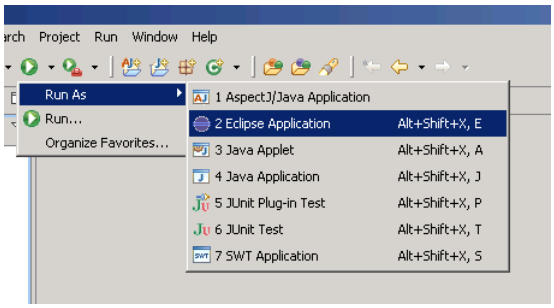


We won't look any deepter at these additional projects for now. However, there is one important thing to point out: The generator also generated the implementation classes for the metamodel. If you take a look into *oaw4.demo.emf.datamodel/src* folder you can find classes (actually, interfaces at the top level) that represent the concepts defined in your metamodel. These can be used to access the model. For some more details on how to use the EMF model APIs as well as the reflective cousins, take a look at

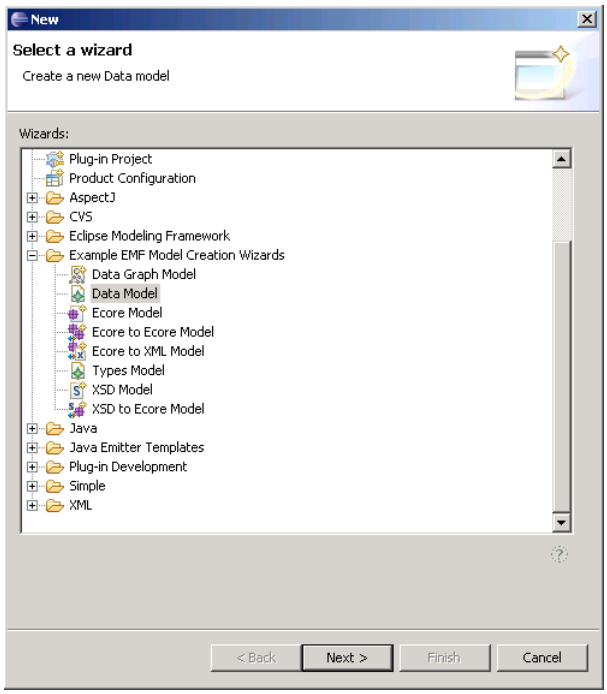
http://voelterblog.blogspot.com/2005/12/codeblogck-emf_10.html

Defining an Example Data Model

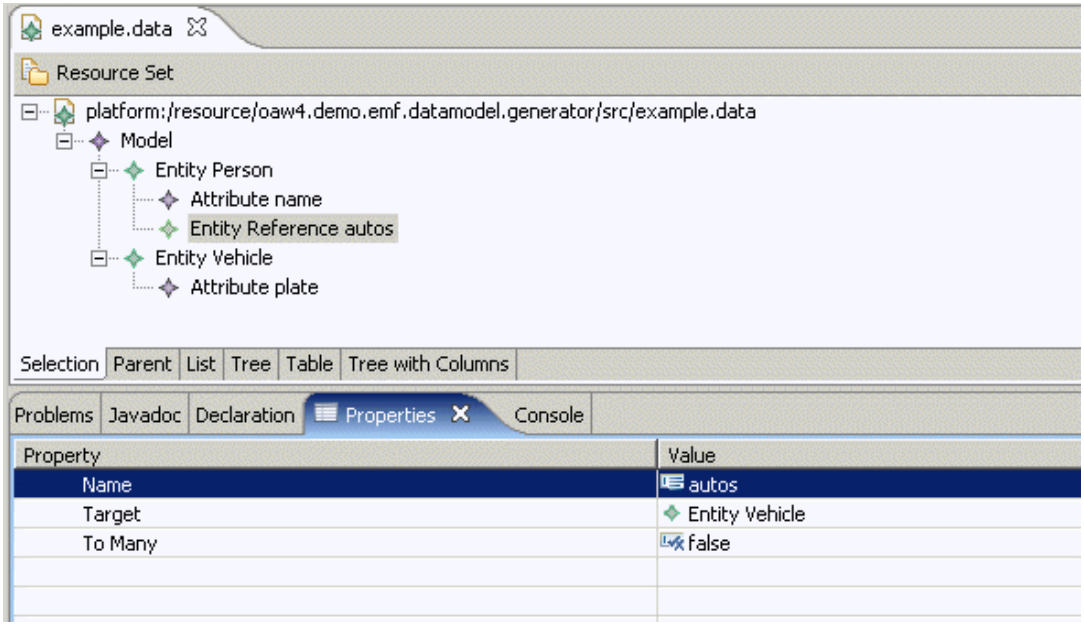
To make working with Eclipse EMF a bit less painless (we'd have to export the plugins, restart Eclipse, etc. etc.) , we start another Eclipse in the IDE. This instance is called the *Runtime Workbench*.



Create a new Java project called *oaw4.demo.emf.datamodel.generator*. Create a source folder named *src*. Create a new *data* model, call it *example.data*. It should be located in the source folder. As the model object, select the *Model*.



Then populate that Model with content as shown in the following. Please note that in the case of attributes you have to define a type as well (i.e. *String*), not just a name.



Again, to make sure you don't include typos, here is the XMI for *example.data*:

```
<?xml version="1.0" encoding="UTF-8"?>
<data:DataModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:data="http://www.openarchitectureware.org/oaw4.demo.emf.datamodel">
  <entity name="Person">
    <attribute name="name" type="String"/>
    <reference target="//@entity.1" name="autos"/>
  </entity>
  <entity name="Vehicle">
    <attribute name="plate" type="String"/>
  </entity>
</data:DataModel>
```

Using Dynamic EMF

Instead of generating editors and metaclasses, you can also use dynamic EMF. This works by selecting, in the opened metamodel, the root class of the model you want to create (here: *DataModel*) and then selecting *Create Dynamic Instance* from the context menu. This opens an editor that can dynamically edit the respective instance. The created file by default has an *.xmi* extension.

Note that openArchitectureWare can work completely with dynamic models, there's no reason to generate code. However, id you want to programmatically work with the model,

the generated metaclasses (not the editors!) are really helpful. There's one other think you'll have to keep in mind: in subsequent parts of the tutorial, you'll specify the metaModelPackage in various component configurations in the workflow file, like so:

```
<metaModel id="mm"
        class="org.openarchitectureware.type.emf.EmfMetaModel">
    <metaModelPackage value="data.DataPackage"/>
</metaModel>
```

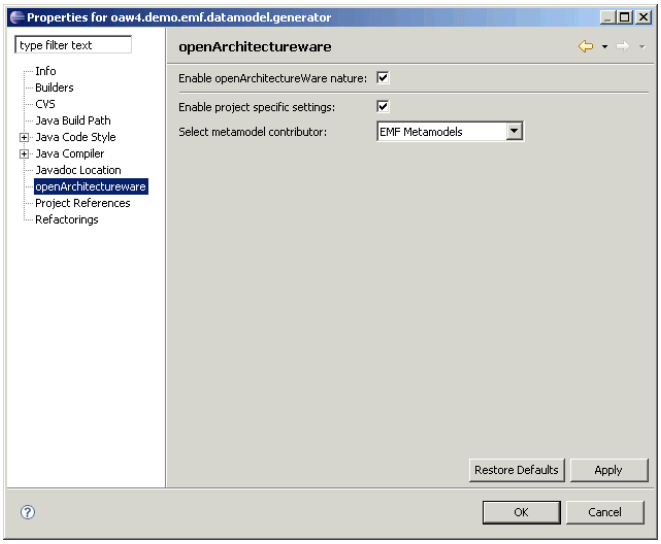
In case of dynamic EMF, there has no metamodel package been generated. So, you have to specify the meta model file instead, that is, the *.ecore* file you just created. Note that the *.ecore* file has to be in the classpath to make this work.

```
<metaModel id="mm"
        class="org.openarchitectureware.type.emf.EmfMetaModel">
    <metaModelFile value="data.ecore"/>
</metaModel>
```

Generating Code From the Example Model

Setting up Eclipse

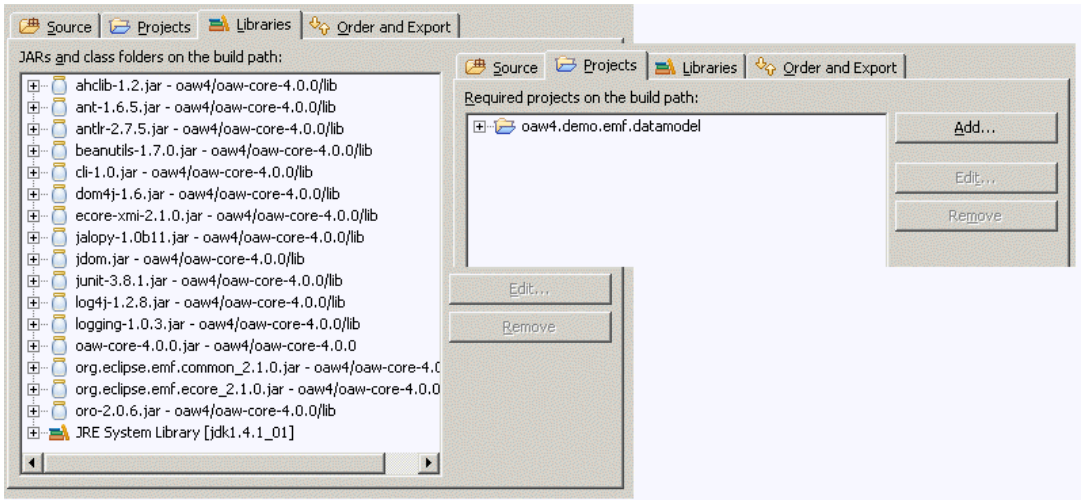
Since we want to use oAW with EMF, you have to set the following preference:



Setting up the project dependencies

Your Eclipse Application Runtime Workspace needs to import the *oaw4* project in your original workspace. This will make all the oAW-Libs available to the runtime workspace. You should also import the *oaw4.demo.emf.datamodel* project. Note that importing the project does not physically move the files; so you can have the project be part of both workspaces at the same time.

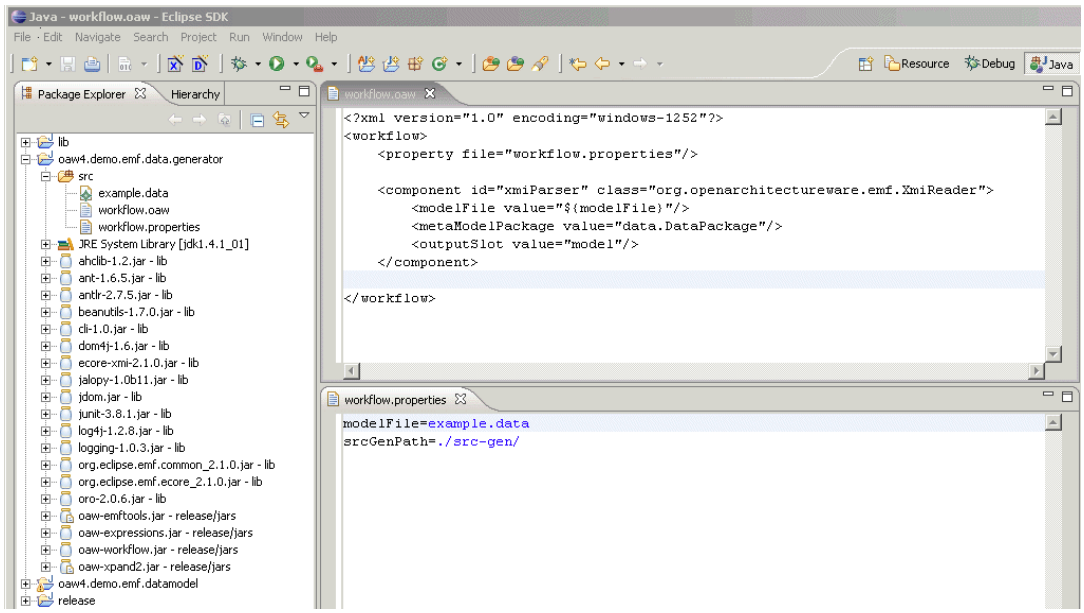
You then have to add dependencies from the *oaw4.demo.emf.datamodel.generator* project to all of the jars inside the *oaw4/oaw-core-4.x.x* directory, as well as those inside the *lib* subdirectory. You also have to add a dependency to *oaw4/oaw-recipe-4.x.x*. You might want to use the same Classpath-Variable based approach as explained above. Finally, you also have to add a project dependency to the *oaw4.demo.emf.datamodel* project.



The workflow definition

To run the openArchitectureWare generator you have to define a workflow. It controls which steps (loading models, checking them, generating code) the generator executes. For details on how workflow files work, please take a look at the *Workflow Reference Documentation*.

Create a *workflow.oaw* and a *workflow.properties* file as shown in the illustration.



The *workflow.oaw* looks as follows:

```
<workflow>
    <property file="workflow.properties"/>

       <component id="xmiParser"
                  class="org.openarchitectureware.emf.XmiReader">
             <modelFile value="${modelFile}"/>
             <metaModelPackage value="data.DataPackage"/>
             <outputSlot value="model"/>
             <firstElementOnly value="true"/>
       </component>
</workflow>
```

The workflow tries to load stuff from the classpath; so, for example, the *data.DataPackage* class is resolved from the classpath, as is the model file specified in the properties:

```
modelFile=example.data
```

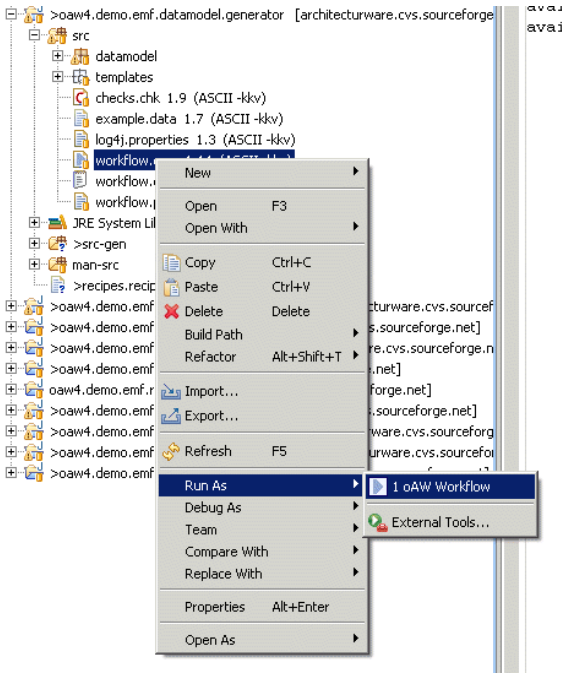
This instantiates the example model and stores in in a workflow slot named *model*. Note that in the *metamodelPackage* slot, you have to specify the EMF package object (here: *data.DataPackage*), not the Java package (which would be *data* here).

Running the workflow

Before you actually run the workflow, make sure you have a log4j configuration in the classapth; for example, you can put the following log4j.properties file directly into your source folder:

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=INFO, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r %-5p - %m%n
```

You can then run the workflow from within Eclipse:

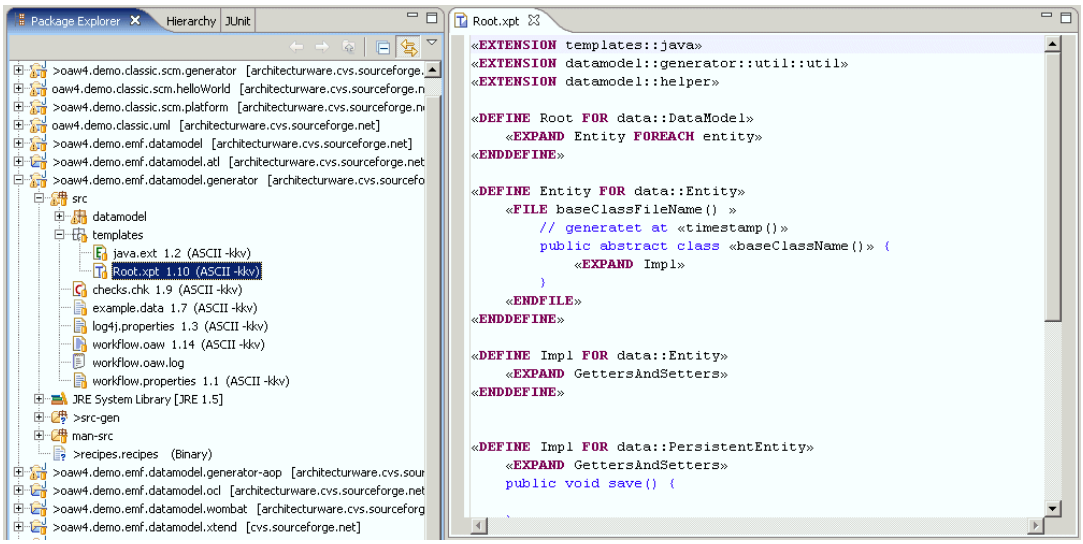


The following should be the output:

```
25.12.2005 12:58:09 - [INFO] openArchitectureWare v4 -
          (c) 2005 openarchitectureware.org and contributors
25.12.2005 12:58:09 - [INFO]
----------------------------------------
25.12.2005 12:58:09 - [INFO] running workflow:
           L:/runtime... datamodel.generator/src/workflow.oaw
25.12.2005 12:58:09 - [INFO] workflow completed!
```

Templates

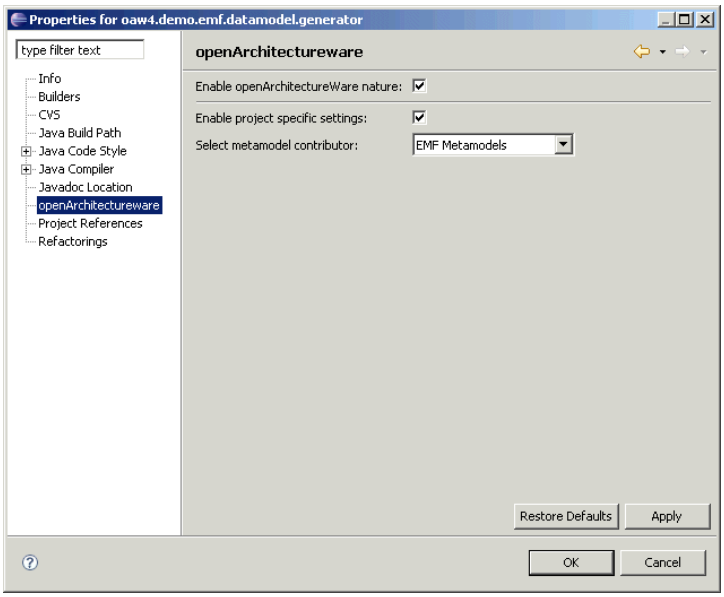
No code is generated yet. This is not surprising, since we did not yet add any templates. Let's change this:



The *Root.tpl* looks as follows. By the way, if you need to type the guillemots (« and »), the editor provides keyboard shortcuts with *Ctrl*+< and *Ctrl*+>.

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

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

In order to see potential errors in the template, we have to add the *oAW4* nature. In the properties dialog for the project, you can add the nature by clicking the respectivce check box on the openArchitectureWare page.



We have to extend the *workflow.oaw* file, in order to use the template just written:

```
<?xml version="1.0" encoding="windows-1252"?>
<workflow>
    <property file="workflow.properties"/>

      <component id="xmiParser"
            class="org.openarchitectureware.emf.XmiReader">
            ...
      </component>
```

First, we clean up the directory where we want to put the generated code.

```
    <component id="dirCleaner"
       class="org.openarchitecturew...ectoryCleaner" >
            <directories value="${srcGenPath}"/>
    </component>
```

Then we start the generator component. It's configuration is slightly involved.

```
   <component id="generator"
          class="org.openarchitectureware.xpand2.Generator">
```

First of all, you have to define the meta model. In our case we use the *EmfMetaModel* since we want to work with EMF models. Also, you have to specific the class name of the EMF package that represents that metamodel. It must be in the classpath!

```
      <metaModel id="mm"
             class="org.openarchitectureware.type.emf.EmfMetaModel">
         <metaModelPackage value="data.DataPackage"/>
```

```
        </metaModel>
```

Then you have to define the „entry statement“ for expand. 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. Again, notice that *model* refers to a slot name here!

```
        <expand value="templates::Root::Root FOR model"/>
```

We then specify where the generator should put the generated code (*genPath*), and where it should put the files that are generated with the ONCE keyword (*genPath*, see *Xpand Language Reference* to learn about the ONCE feature):

```
        <genPath value="${srcGenPath}/"/>
        <srcPath value="${srcGenPath}/"/>
```

Finally, we run a beautifier over the generated code:

```
        <beautifier
              class="org.openarchitectu...put.JavaBeautifier"/>
        </component>

</workflow>
```

You also need to add the *srcGenPath* to the *workflow.properties* file.

```
modelFile=example.data
srcGenPath=src-gen
```

Running the generator again

So, if you restart the generator now, you should get a file generated that looks like this:

```
public class Person {

        // bad practice
        public String lastName;

}
```

Checking Constraints with the Checks Language

An alternative to checking constraints with pure Java is the declarative constraint checking language Check. For details of this language take a look at the *Check Language Reference.* We will provide a simple example here.

Defining the contraint

We start by defining the constraint itself. We create new file called *checks.chk* in the *src* folder of our project. It is important that this file lies in the classpath! The file has the following content (note the syntax highlighting from the custom editor):

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

This constraint says that for the metaclass *data::Attribute*, we require that the name be more than one characters long. 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 ) );
```

Integration into the workflow file

The following piece of XML is the workflow file we'd already used above.

```
<?xml version="1.0" encoding="windows-1252"?>
<workflow>
    <property file="workflow.properties"/>

      <component id="xmiParser"
class="org.openarchitectureware.emf.XmiReader">
            ...
      </component>
```

After the reading of the model, we add an additional component, namely the *CheckComponent*.

```
      <component
           class="org.openarchitectureware.check.CheckComponent">
```

As with the code generator, we have to explain to the checker what meta meta model and which meta model we use.

```
            <metaModel id="mm"
class="org.openarchitectureware.type.emf.EmfMetaModel">
              <metaModelPackage value="data.DataPackage"/>
            </metaModel>
```

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 – that's why it has to be located in the classpath.

```
            <checkFile value="checks"/>
```

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.

```
            <emfAllChildrenSlot value="model"/>
        </component>
```

Running the workflow produces an error in case the length of the name is not greater than one.Again, it makes sense to add the *skipOnError="true"* to those subsequent component invocations that need to be skipped in case the constraint check found errors (typically code generators or transformers).

Extensions

It is often the case that you need 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. For details see the *Extend Language Documentation*, we provide an simple example here.

Expression-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 a so-called 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 properties. We put this file directly into the *templates* directory under *src*, i.e. directly next to the *Root.xpt* file. The extension file looks as follows.

First we have to import the *data* metamodel; otherwise we'd not be able to use the *Attribute* metaclass.

```
import data;
```

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 returns the to-be-defined value.

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

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

To make these extensions work, we have to add the following line to the beginning of the *Root.xpt* template file:

```
«EXTENSION templates::java»
```

Java Extensions

In case you cannot express the „business logic“ for the expression with the expression language, you can fall back to Java. Take a loot at the following extension definition file. It's called *util.ext* and is located in *src/datamodel/generator/util*:

```
String timestamp() :
   JAVA datamodel.generator.util.TemplateUtils.timestamp();
```

Here we define an extension that is independent of a specific model element, since it does not have a formal parameter! The implementation of the extension is delegated to a static operation of a Java class. Here is it's implementation:

```
public class TemplateUtils {
	public static String timestamp() {
		return String.valueOf( System.currentTimeMillis() );
	}
}
```

This element can be used independent of any model element – it's available globally.

Sometimes it's necessary to access extensions not just from templates (and Wombat scripts) but also from Java code. The following example is of this kind: we want to define properties that derive the name of the implementation class from the entity name itself; we'll need that property in the next section, the one on recipes. The best practice for this use case is to implement the derived property as a Java method, as above. The following piece of code declares properties for *Entity*:

```
package datamodel;

import data.Entity;

public class EntityHelper {

	public static String className( Entity e ) {
		return e.getName()+"Implementation";
```

```
	}

	public static String classFileName( Entity e ) {
		return className(e)+".java";
	}

}
```

In addition, to access the properties from the template files, we define an extension that uses the helper methods. The *helper.ext* file is located right next to the helper class shown above, i.e. in the *datamodel* package:

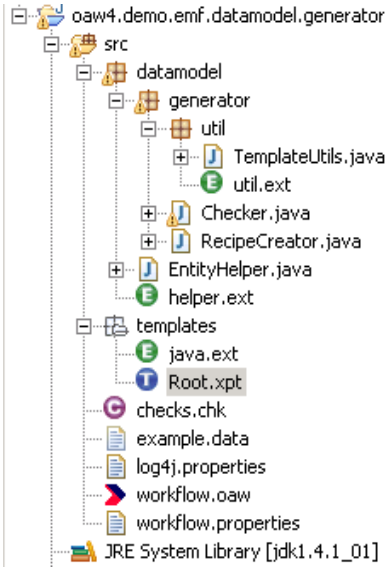
```
import data;

String className( Entity e ) :
   JAVA datamodel.EntityHelper.className(data.Entity);

String classFileName( Entity e ) :
   JAVA datamodel.EntityHelper.classFileName(data.Entity);
```

In addition to these new properties being accessible from Java code by invoking *EntityHelper.className(someEntity)*, we can now write the following template:

```
«EXTENSION templates::java»
«EXTENSION datamodel::generator::util::util»
«EXTENSION datamodel::helper»

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

«DEFINE Entity FOR data::Entity»
	«FILE classFileName()»
	    // generatet at «timestamp()»
	    public abstract class «className()» {

	       «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»
	    }
	«ENDFILE»
```

```
«ENDDEFINE»
```

For completeness, the following illustration shows the resulting directory and file structure.



Integrating Recipes

Let's assume we wanted to allow developers to add their own business logic to the entites, maybe adding a couple of derived properties. In that case, we have to integrate the generated code with manually written fragments. Let's further assume that you – just like me – don't like protected regions because the end up in versioning chaos. In such case you might want to let the generator create a base class that contains all generated aspects and developers have to inherit from this class to add their own logic. Let's first change the generator accordingly.

Adapting the existing generator

Let's first look at the template. Here we have to change the name of the generated class, and we have to make it *abstract*:

```
«DEFINE Entity FOR data::Entity»
      «FILE baseClassFileName()»
          // generatet at «timestamp()»
          public abstract class «baseClassName()» {

                    «FOREACH attribute AS a»
                        ...
                    «ENDFOREACH»
```

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

To make this work, our extensions must be adapted; we now need *baseClassName* and *baseClassFileName*.

```
import data;

String baseClassName( Entity e ) :
   JAVA datamodel.EntityHelper.baseClassName(data.Entity);

String baseClassFileName( Entity e ) :
   JAVA datamodel.EntityHelper.baseClassFileName(data.Entity);
```

The implementation helper class must be adapted, too:

```
package datamodel;

import data.Entity;

public class EntityHelper {

      public static String baseClassName( Entity e ) {
            return e.getName()+"ImplBase";
      }

      public static String baseClassFileName( Entity e ) {
            return baseClassName(e)+".java";
      }

      public static String implementationClassName( Entity e ) {
            return e.getName();
      }

}
```

Note the additional property *implementationClassName*. This is the name of the class that developers have to write manually. While we expect that the generated code goes into the *src-gen* directory, we want the manually written code in *man-src*. Here is the generated base class for the *Person* entity:

```
// generatet at 1138622360609
public abstract class PersonImplBase {
    private String name;

    public void setName(String value) {
        this.name = value;
    }

    public String getName() {
```

```
        return this.name;
    }
}
```

The manually written subclass could look as follows:

```
public class Person extends PersonImplBase {

}
```

Now, here's the issue: how do you make sure that developers actually write this class, that it has the right name and that it actually extends the generated base class? This is where the recipe framework comes into play. We want to define rules that allow Eclipse to verfy that these „programming guidelines“ have been met by developers.

Implementing the Recipes

As of now there is no specific language to implement those recipe checks, you have to write a bunch of Java code. In summary, you have to implement a workflow component that produces the checks. Let's look at what you need to do.

In order to simplify life, your recipe creation component should the *RecipeCreationComponent* base class.

```
public class RecipeCreator extends RecipeCreationComponent {
```

You then have to override the *createRecipes* operation.

```
  protected Collection createRecipes(Object modelSlotContent,
                String appProject, String srcPath) {
```

We now create a list that we use to collect all the checks we want to pass back to the framework.

```
      List checks = new ArrayList();
```

Since we need to implement such a check for each *Entity* in the model, we have to find all entities and iterate over them.

```
      Collection entities = EmfUtil2.findAllByType(
           ((DataModel)modelSlotContent).eAllContents(),
               Entity.class );
      for (Iterator iter = entities.iterator(); iter.hasNext();) {
      Entity e = (Entity) iter.next();
```

We then create a composite check whose purpose is to act as a container for the more specific checks that follow. It will show as the root of a tree in the Recipe Framework view.

```
        ElementCompositeCheck ecc = new ElementCompositeCheck(e,
          "manual implementation of entity");
```

Then we add a check that verifies the exsitence of a certain class in a given project in a certain directory. The name of the class it needs to check for can be obtained from our *EntityHelper*!

```
        JavaClassExistenceCheck javaClassExistenceCheck =
            new JavaClassExistenceCheck(
                 "you have to provide an implementation class.",
                 appProject, srcPath,
                 EntityHelper.implementationClassName(e)
               );
```

We then define a second check that checks that the class whose existence has been verified with the check above actually inherits from the generated base class. Again we use the *EntityHelper* to come up with the names. Note that they will be consistent with the names used in the code generation templates because both use the same *EntityHelper* implementation.

```
        JavaSupertypeCheck javaSuperclassCheck =
            new JavaSupertypeCheck(
               "the implementation class has to extend the "+
               "generated base class", appProject,
               EntityHelper.implementationClassName(e),
               EntityHelper.baseClassName(e)
            );
```

We then add the two specific checks to the composite check...

```
        ecc.addChild( javaClassExistenceCheck );
        ecc.addChild( javaSuperclassCheck );
```

... add the composite check to the list of checks we return to the framework, ...

```
        checks.add( ecc );
      }
```

... and return all the created checks to the framework after we finish iteration over *Entities*.

```
      return checks;
   }
}
```

Workflow Integration

Here's the modified workflow file. We integrate our new component as the last step in the workflow.

```
<?xml version="1.0" encoding="windows-1252"?>
<workflow>
    <property file="workflow.properties"/>

      <component id="xmiParser"
                 class="org.openarchitectureware.emf.XmiReader">
```

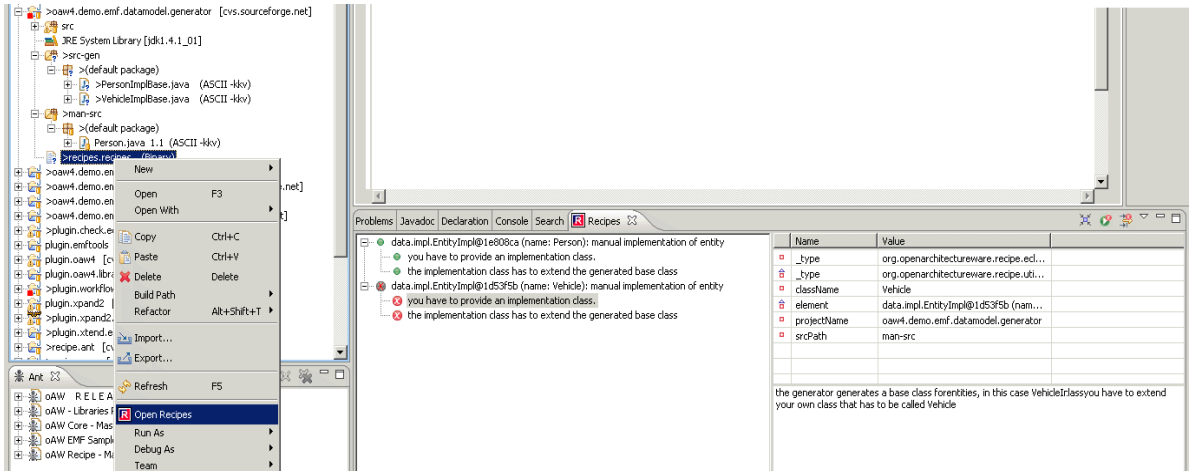
```
            <modelFile value="${modelFile}"/>
            <metaModelPackage value="data.DataPackage"/>
            <outputSlot value="model"/>
            <firstElementOnly value="true"/>
        </component>

        <!-- all the stuff from before -->
```

The parameters we pass should be self-explanatory. The *recipeFile* parameter is where the checks will written to – it must have the *recipes* extension.

```
        <component id="recipe"
                    class="datamodel.generator.RecipeCreator">
            <appProject value="oaw4.demo.emf.datamodel.generator"/>
            <srcPath value="man-src"/>
            <modelSlot value="model"/>
            <recipeFile value="recipes.recipes"/>
        </component>

</workflow>
```

Running the Workflow and seeing the Effect

We can now run the workflow. After running it, you should see a *recipes.recipes* file in the root of your project. Right-clicking on it reveals the *Open Recipes* button. Since the manual implementation of the *Vehicle Entity* is missing, we get the respective error.



We can now implement the class manually, in the *man-src* folder:

```
public class Vehicle extends VehicleImplBase {
}
```

After doing that, the remaining errors in the recipe view should go away automatically.

Transforming Models

It is often necessary to transform models before generating code from it. There are actually two forms of transformations:

- An actual model *transformations* generates a completely new model – usually based on a different metamodel – from an input model. The transformation has no sideeffects wrt. to the input model.
- A model *modification* completes/extends/finishes/modifies a model. No additional model is created.

Please take a look at the *xTend Example* tutotial to understand model transformations with the *xTend* language.

Model Modifications in Java

One way of doing modifications is to use Java. Take a look at the following piece of Java code. We extend from a class called *SimpleJavaTransformerComponent.* Instead of implementing directly the *WorkflowComponent* interface, we inherit from a more comfortable base class and implement the *doModification* operation.

```
public class Transformer extends SimpleJavaModificationComponent {

  protected void doModification(WorkflowContext ctx, ProgressMonitor
            monitor, Issues issues, Object model) {
```

We know that we have a *DataModel* object in the model slot (you can see in a moment where the model comes from).

```
      DataModel dm = (DataModel)model;
```

We then get us the factory to create new model elements (what this code does exactly you should learn from the EMF docs).

```
      DataFactory f = DataPackage.eINSTANCE.getDataFactory();
```

We then iterate over all entities.

```
      for (Iterator iter = dm.getEntity().iterator();
                          iter.hasNext();) {
            Entity e = (Entity) iter.next();
            handleEntity(e, f);
```

```
        }
    }
```

For each *Entity*...

```
    private void handleEntity(Entity e, DataFactory f) {
        for (Iterator iter =
                    EcoreUtil2.clone( e.getAttribute() ).iterator();
                            iter.hasNext();) {
                Attribute a = (Attribute) iter.next();
```

We create a new attribute with the same type, and a name with a „2“ postfixed. We then add this new attribute to the entity.

```
                Attribute a2 = f.createAttribute();
                a2.setName( a.getName()+"2" );
                a2.setType( a.getType() );
                e.getAttribute().add(a2);
        }
    }
}
```

To add this component to the workflow, we just have to add it to the workflow:

```
<component class="datamodel.generator.Transformer">
        <modelSlot value="model"/>
</component>
```

We have to specify the model slot. The super class (*SimpleJavaTransformerComponent*) provides the slot-setter and passes the object from that slot to the *doTransform* operation.

About our Sponsors

itemis GmbH & Co. KG is an independent IT service company with an emphasis on consulting, coaching, and software development. Every single itemis expert provides many years of project experience and widespread knowledge about all object oriented and component based software development issues - especially in the field of model driven software development.

b+m is the founder of the openArchitectureWare project. The software was originally developed within the scope of many successful projects. b+m opened the software to the community in late 2003. All of the paradigms of Model-Driven Software Development including Product Line Engineering and not only the generator framework have become a key concept for product and customer specific development at b+m. b+m customers can make use of long time expirience and substantial know-how in that field. Located at the company headquarters in Melsdorf/Kiel and at its subsidiaries in Berlin, Cottbus, Hamburg, Hanover and Kiel the b+m staff of 205 provides practical solutions for customized business applications, business process optimization and comprehensive architecture, project and quality management.

oose Innovative Informatik GmbH offers coaching, consulting and training in all themes about software engineering. The main focus of their activities are software architecture, requirements engineering and project-management. oose have first-hand information and experience, because our staff take actively part with others in actual trends, standards and innovations. Our staff support this and pass their know-how regularly on by writing and publishing books or being speaker at conferences, etc. Within the OMG oose collaborate actively on the specifications of the UML and also the SysML.

MID Enterprise Software Solutions GmbH is a leading supplier of optimized tool environments for standardsbased and model-centric software development as well as business process modeling. This includes professional tool consulting and tool components to build a complete tool environment using the best techniques and tool modules available - Architectural and Operational Excellence. With innovatorAOX, MID provides a holistic standard tool environment for object- and function-oriented software development as well as business process and data modeling to help its customers establish highly efficient processes and tool environments for software production, The unique and seamless integration of business process modeling into the development process ensures an unprecedented level of convergence of business requirements and implemented IT systems. Project members from all departments speak the same language and all requirements are clearly described.