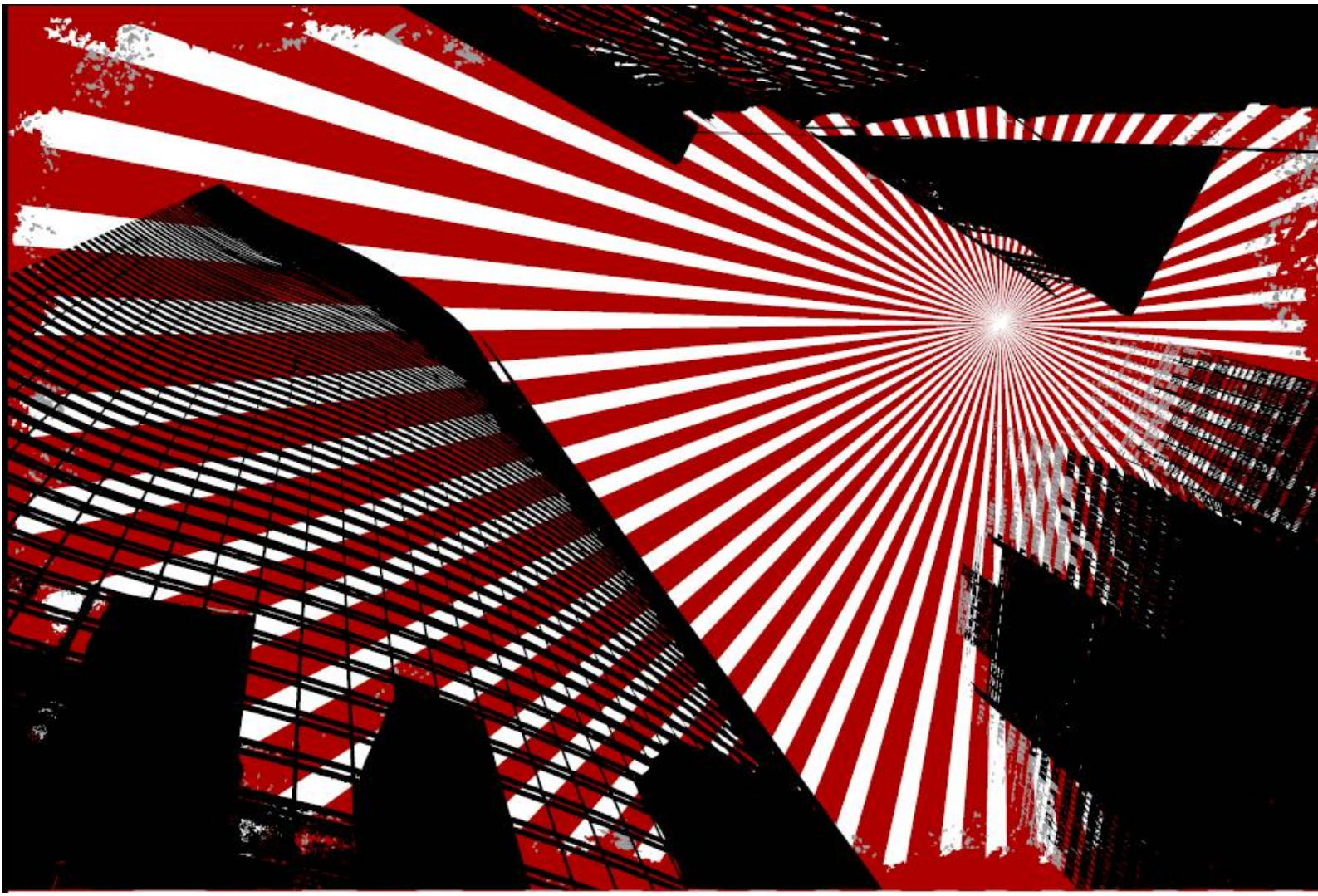


openArchitectureWare User Guide

Version 4.3

openArchitectureWare

2008-04-11

Table of Contents

List of Figures

List of Tables

Part I. Getting Started

Chapter 1. oAW Tutorial

This example uses Eclipse EMF as the basis for code generation. One of the essential new features of openArchitectureWare 4 is EMF support. While not all aspects of EMF as good and nice to use as one would wish, the large amount of available third party tools makes EMF a good basis. Specifically, better tools for building EMF metamodels are available already (Xtext, GMF, etc.). 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 seem unnecessarily complex to you.

1.1. Installing the pre-built tutorial

You need to have openArchitectureWare 4.3 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 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.

1.2. Tutorial overview

The purpose of this tutorial is to illustrate code generation with openArchitectureWare from EMF models. The process, we are going to go through, will start by defining a metamodel (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 JavaBean 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. We will not to this for this simple introduction.

1.3. Defining an EMF metamodel

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

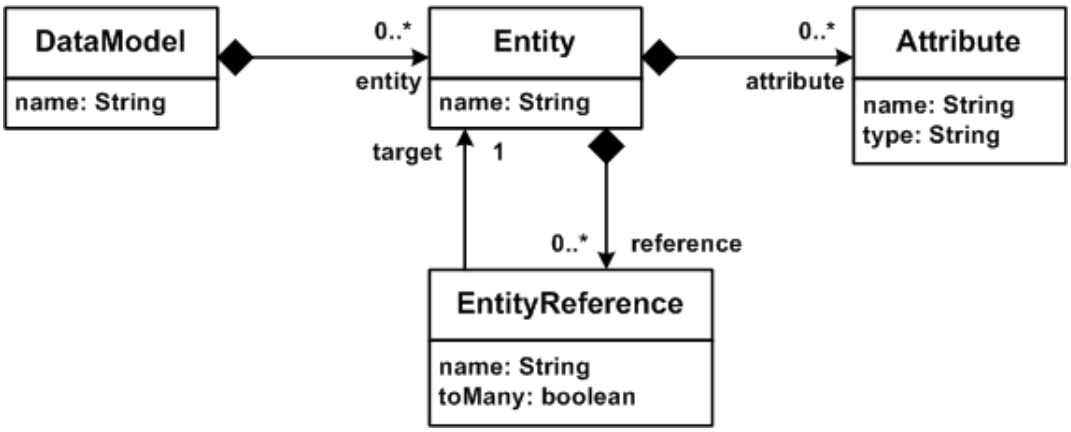


Figure 1.1. Sample metamodel

1.3.1. Creating an EMF project

Create an EMF project as depicted below:

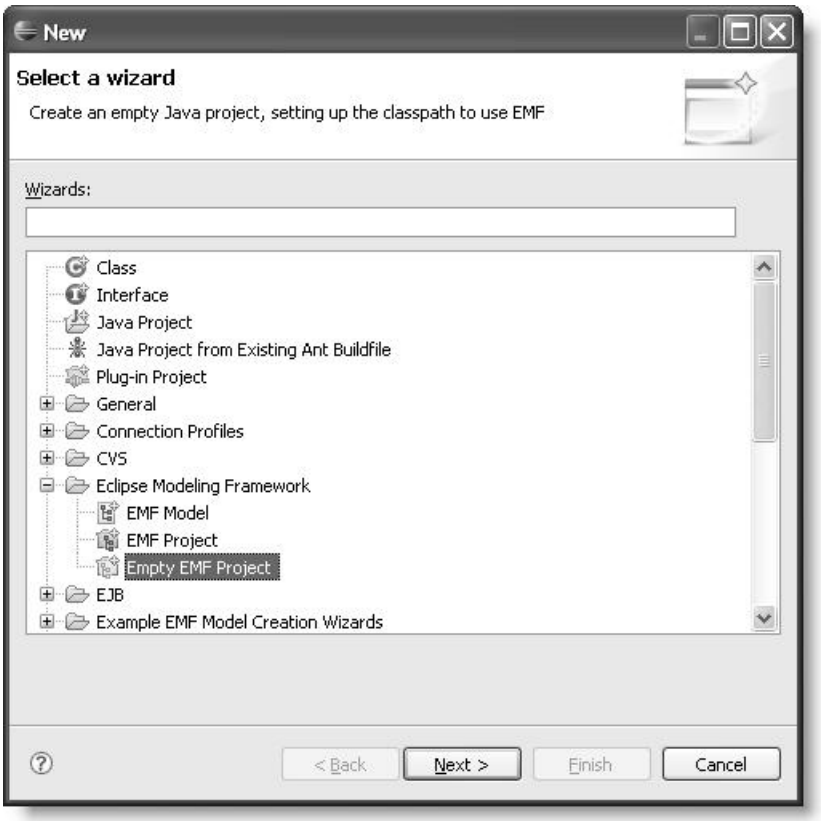


Figure 1.2. Create EMF project

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

1.3.2. Defining the (meta)model

Create a new source folder `metamodel` in that project. Then, create a new Ecore model in that source folder named `data.ecore`. Use EPackage as the model object.

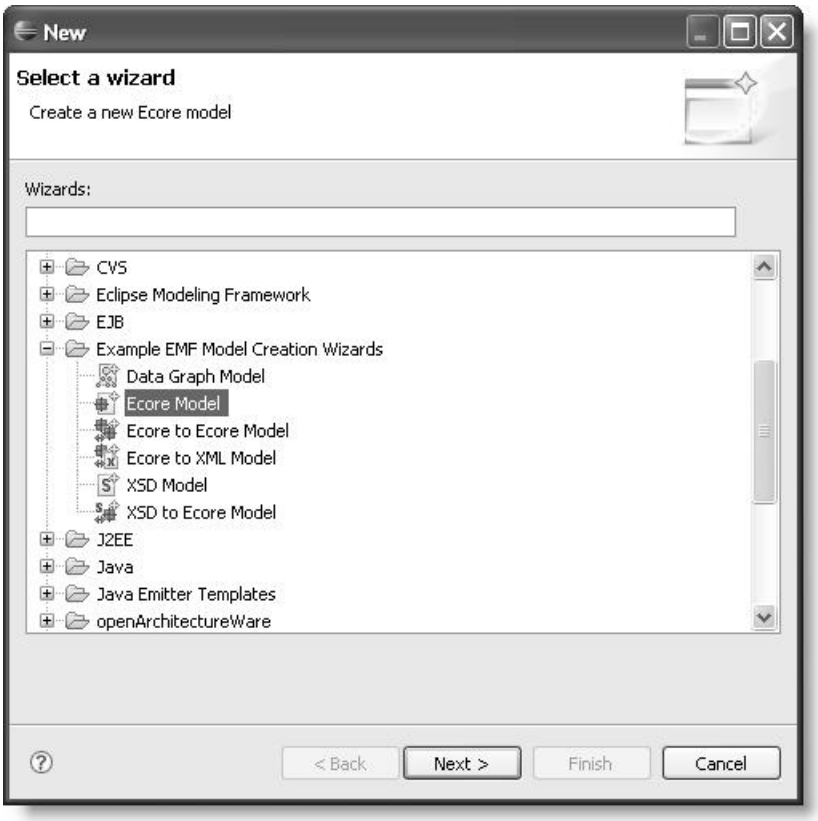


Figure 1.3. Create new Ecore model

This opens the Ecore Editor. You will see a root package with name `null`. Open the Properties View (context menu). Set the following properties for the package:

- Name: data
- Ns prefix: data
- Ns URI: http://www.openarchitectureware.org/oaw4.demo.emf.datamodel

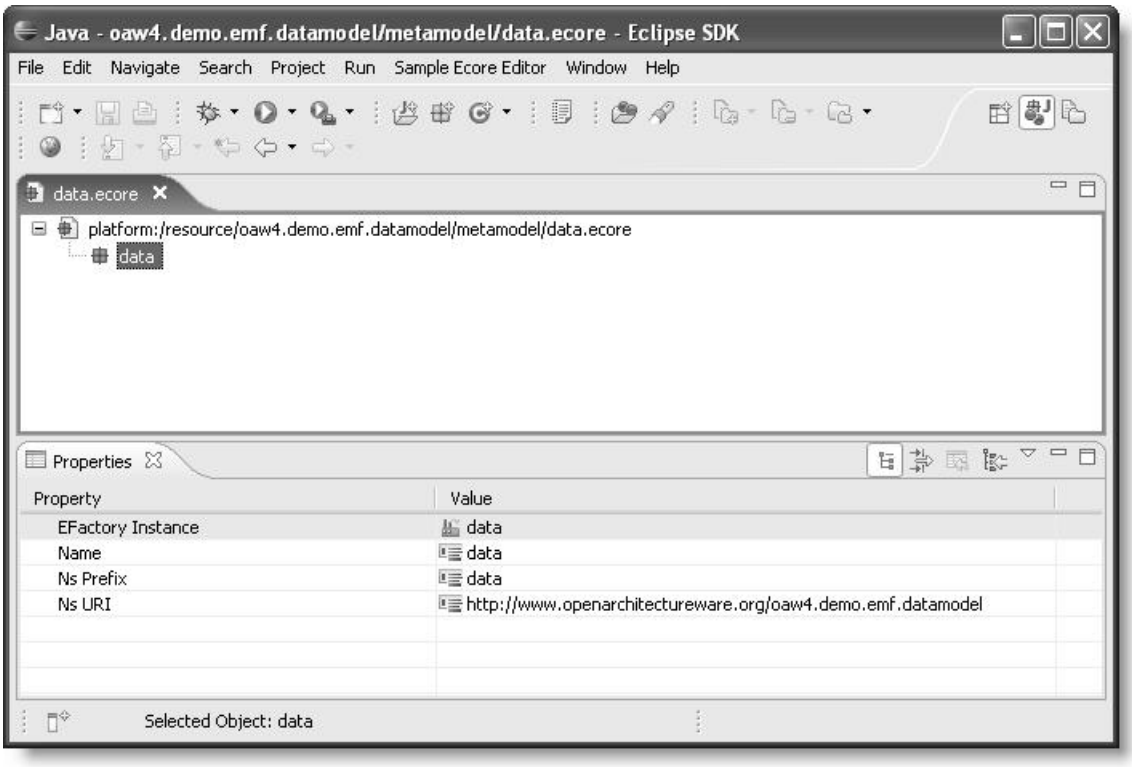


Figure 1.4. Adjust namespace settings

Create the following Ecore model.[1] Make sure you set the following properties *exactly* as described next:

Within the data package, create these `EClass` elements with their attributes:[2]

EClass name	EAttribute name	EAttribute EType
DataModel		
	name	EString
Entity		
	name	EString
Attribute		
	name	EString
	type	EString
EntityReference		
	name	EString
	toMany	EBoolean

Now, it is time to create references between the model elements. Add children of type `EReferences` as follows:[3]

EClass	EReference name	EReference attribute name	EReference attribute value

[1]To add children, right-click on the element to which you want to add these 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 menus. Note that this is not an EMF tutorial. For more details on how to build EMF (meta-)models, please refer to the EMF documentation.

[2]Attributes are children of type `EAttribute`. Please fill in the Name and the EType properties.

[3]Note: there are a couple of *-1's* ... don't miss the minus! Also, the containment flag is essential. If containment is `true` you will be able to create children of the referenced type, otherwise you can only reference them.

EClass	EReference name	EReference attribute name	EReference attribute value
DataModel			
	entity		
		EType	Entity
		containment	true
		Lowerbound	0
		Upperbound	-1
Entity			
	attribute		
		EType	Attribute
		containment	true
		Lowerbound	1
		Upperbound	-1
Entity			
	reference		
		EType	EntityReference
		containment	true
		Lowerbound	0
		Upperbound	-1
EntityReference			
	target		
		EType	Entity
		containment	false
		Lowerbound	1
		Upperbound	1

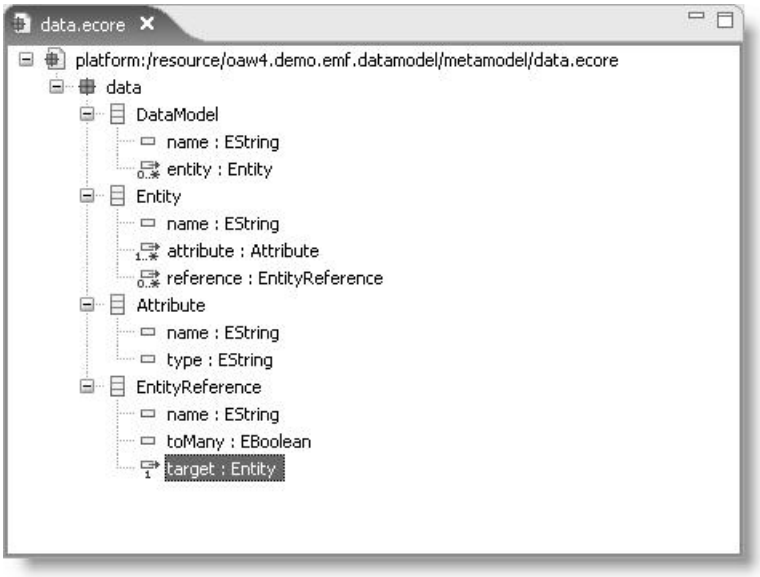


Figure 1.5. Metamodel structure

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:EAttribute" name="name"
   eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="attribute" lowerBound="1"
    upperBound="-1" eType="#//Attribute" containment="true"/>
  <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: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"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="target" lowerBound="1"
    eType="#//Entity"/>
 </eClassifiers>
</ecore:EPackage>
```

1.4. 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 will 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*

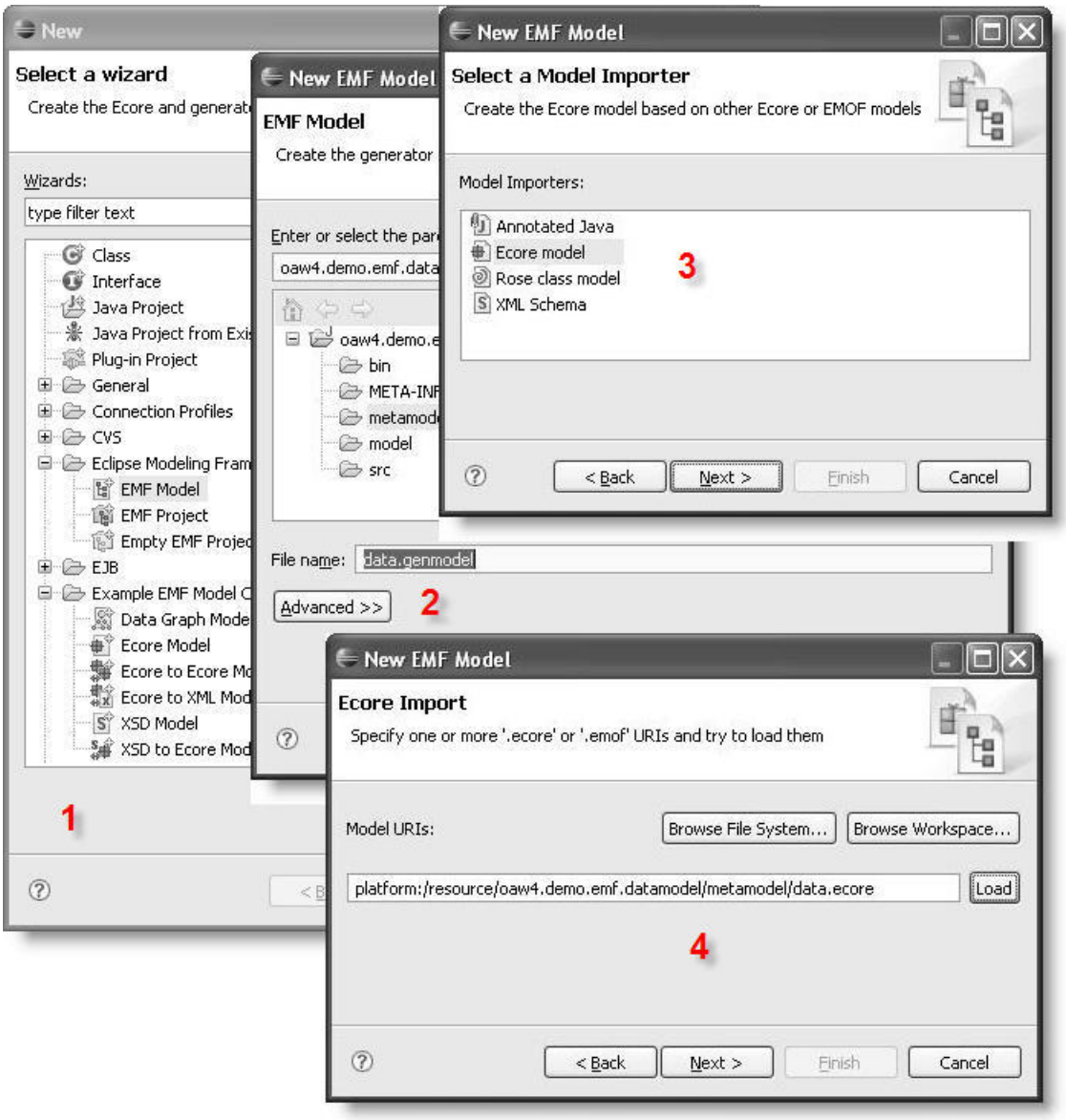


Figure 1.6. Creating the genmodel

As a consequence, you will 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 do not have to change any of these.

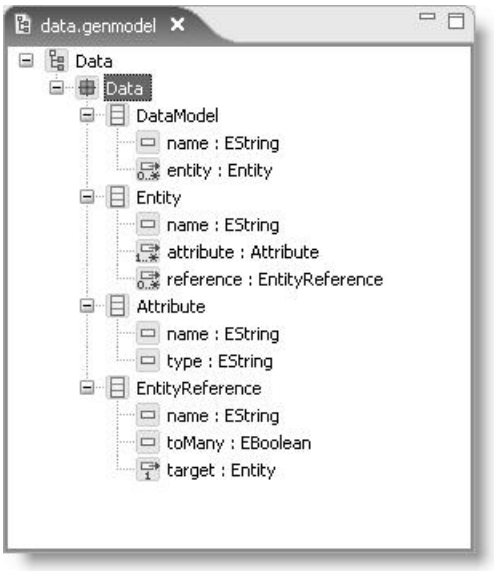


Figure 1.7. Structure of the genmodel

You can now generate the other projects.

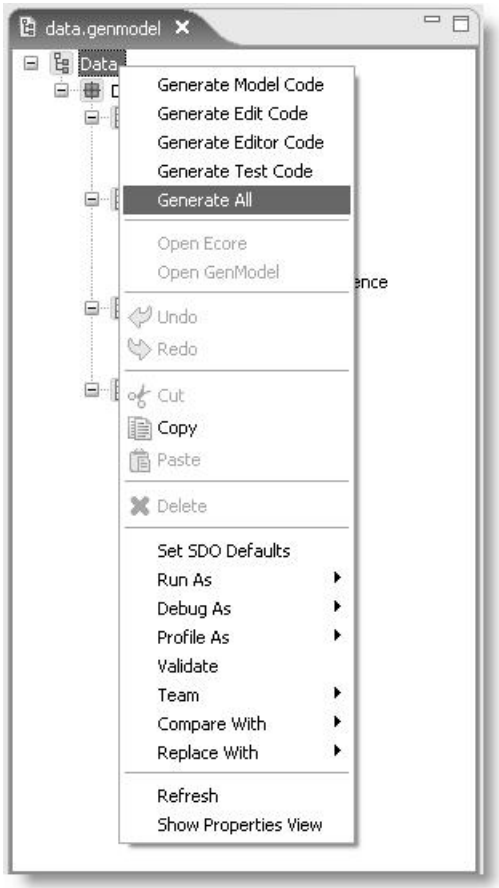


Figure 1.8. Generate editing projects

You now have all the generated additional projects.

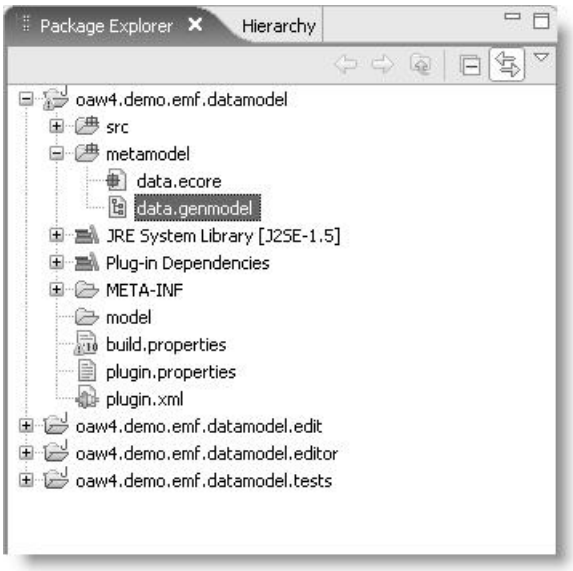


Figure 1.9. Generated projects

We will not look any deeper 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.

1.5. Setting up the generator project

In order to make it a bit less painless to work with Eclipse EMF (we would have to export the plugins, restart Eclipse, etc. etc.), we start another Eclipse in the IDE. This instance is called the *Runtime Workbench*. Therefore select the `oaw4.demo.emf.datamodel.edit` project and choose from the context menu Run As -> Eclipse Application.

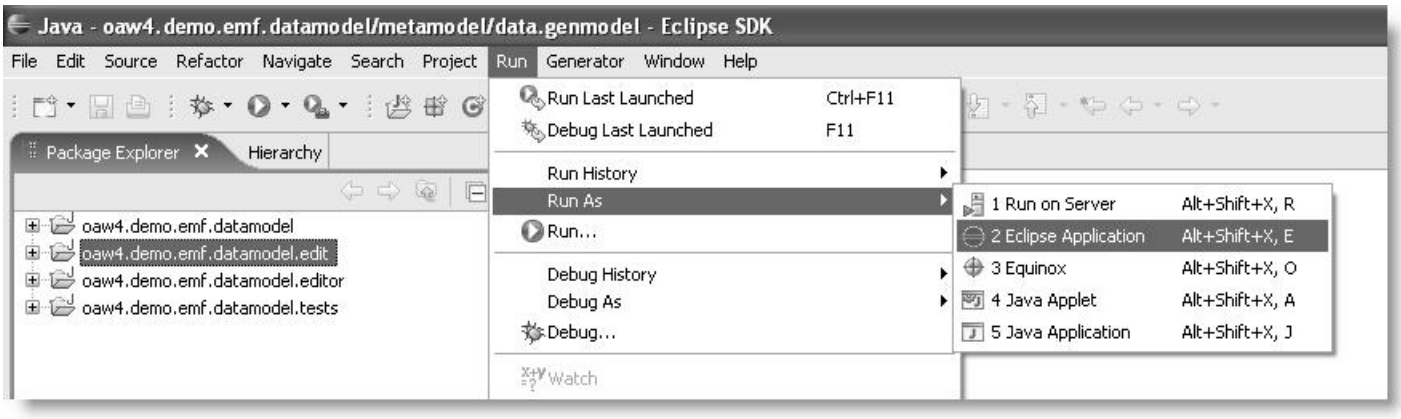


Figure 1.10. Launch runtime platform

If you are using a Mac or *nix you should now open the workspace preference page and change the default encoding to ISO-8859-1.[4]Import the `oaw4.demo.emf.datamodel` project from your original workspace.[5] Note that importing the project does not physically move the files,[6] so you can have the project be part of both workspaces at the same time.

Create a new openArchitectureWare Project[7] called `oaw4.demo.emf.datamodel.generator`. Do not choose the option "Generate a simple example".

[4]Window -> Preferences -> General -> Workspace -> Text file encoding. This is necessary to have the *guillemet* brackets available.

[5]File -> Import -> General -> Existing Project into Workspace

[6]Unless you checked the option "Copy projects into workspace"

[7]File -> New -> Project -> openArchitectureWare -> openArchitectureWare Project

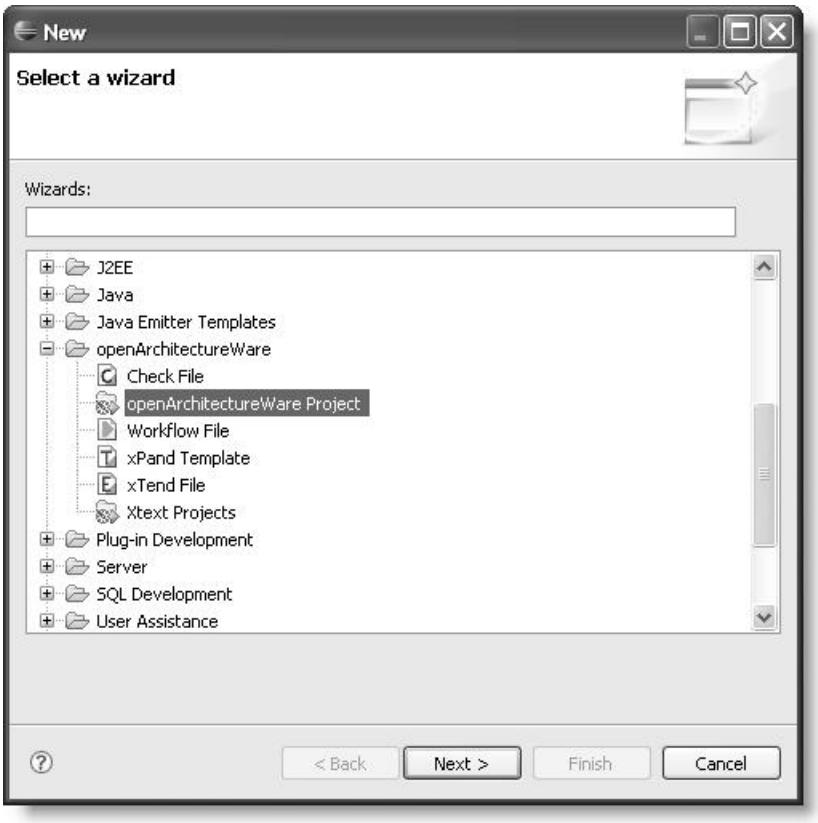


Figure 1.11. Create new oAW project

Your openArchitectureWare project will already be configured for use of EMF models. You can check this in the project properties dialog:

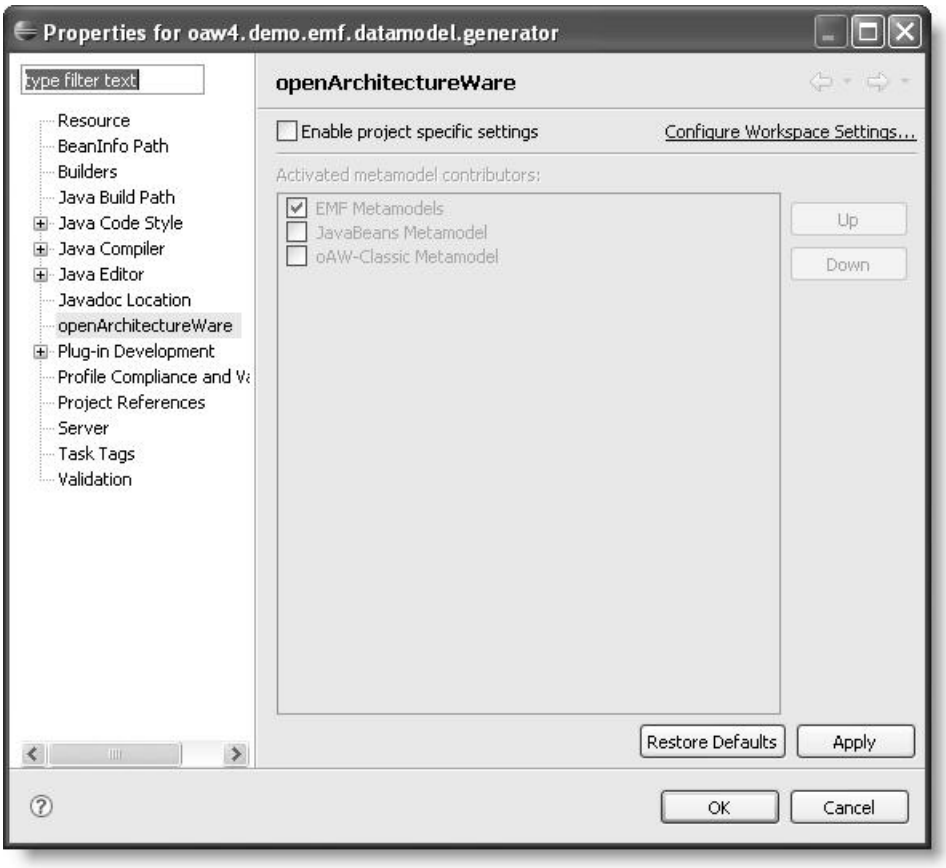


Figure 1.12. Project properties

1.6. Defining an Example Data Model

Select the `src` folder and then choose New -> Other -> Example EMF Model Creation Wizards -> Data Model. Create a new data model, call it `example.data`. On the last page of the wizard, select *Model* as model object.

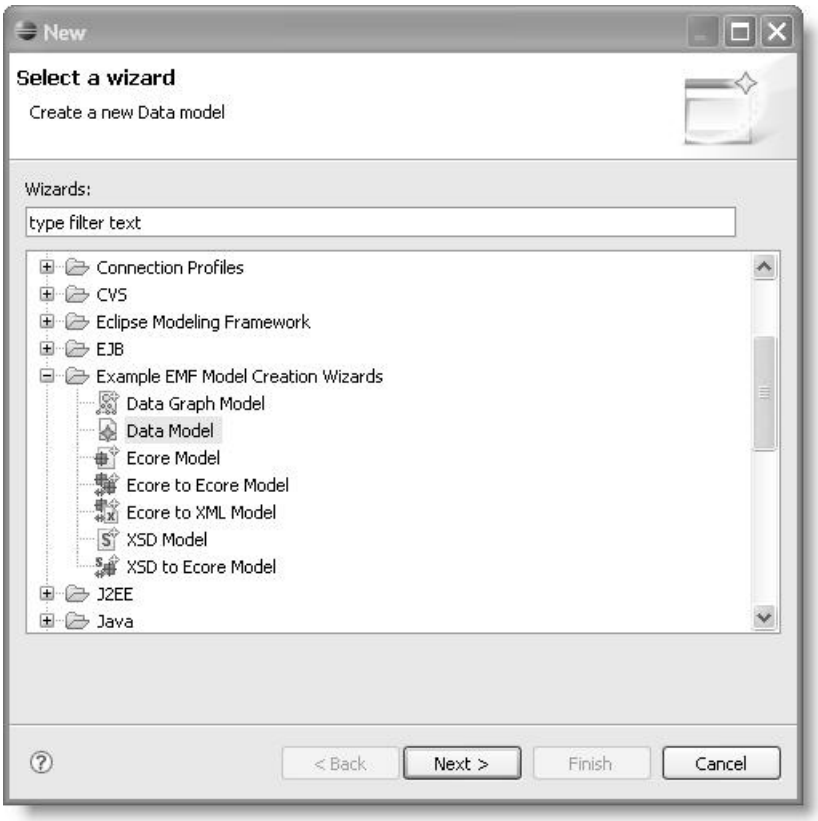


Figure 1.13. Create a sample data model

Next, populate this very model as following. Please note that in the case of attributes you have to define a type as well (i.e. String), not just a name.

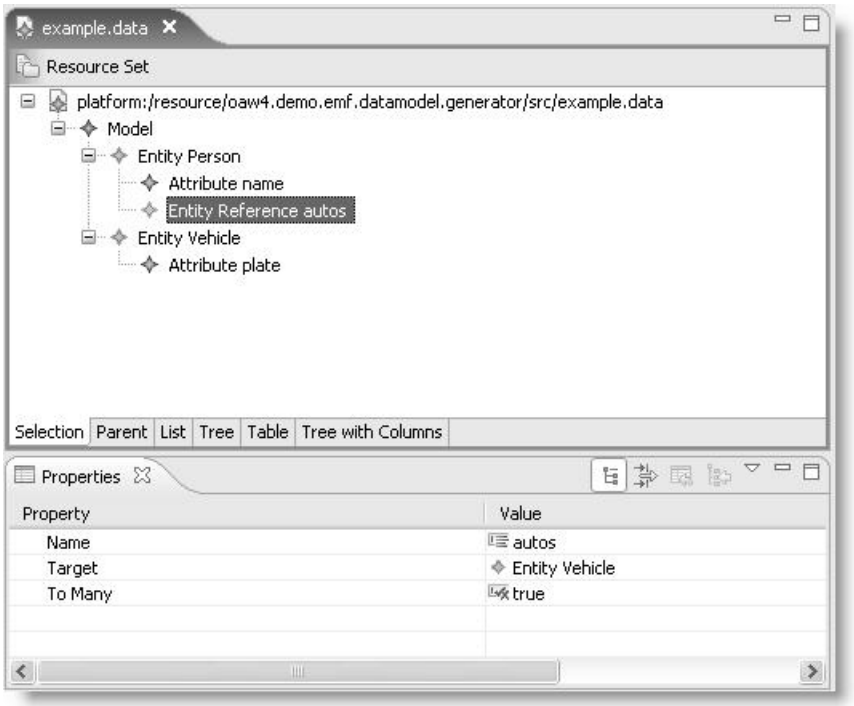


Figure 1.14. Sample data model

Again, to avoid any 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 name="autos" toMany="true" target="//@entity.1"/>
 </entity>
 <entity name="Vehicle">
  <attribute name="plate" type="String"/>
 </entity>
</data:DataModel>
```

1.7. 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 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 is no reason to generate code. However, if you want to programmatically work with the model, the generated metaclasses (not the editors!) are really helpful. Please also keep in mind: in subsequent parts of the tutorial, you will specify the *metaModelPackage* in various component configurations in the workflow file, like this:

```
<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 metamodel 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>
```

1.8. Generating code from the example model

1.8.1. 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` in the `src` folder. The contents of these files is shown below:

```
<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>
```

`workflow.properties`:

```
modelFile=example.data
srcGenPath=src-gen
fileEncoding=ISO-8859-1
```

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 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).

1.8.2. Running the workflow

Before you actually run the workflow, make sure you have a log4j configuration in the classpath; 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
```

Also, please make sure your metamodel can be found on the classpath. In our case, this can be achieved by adding the `oaw4.demo.emf.datamodel` project to the plug-in dependencies of `oaw4.demo.emf.datamodel.generator`. To do this, double click the file `oaw4.demo.emf.datamodel.generator/META-INF/MANIFEST.MF`. The manifest editor will appear. Go to the Dependencies tab and click on Add... to add a new dependency. In the dialog appearing, choose `oaw3.demo.emf.datamodel`:

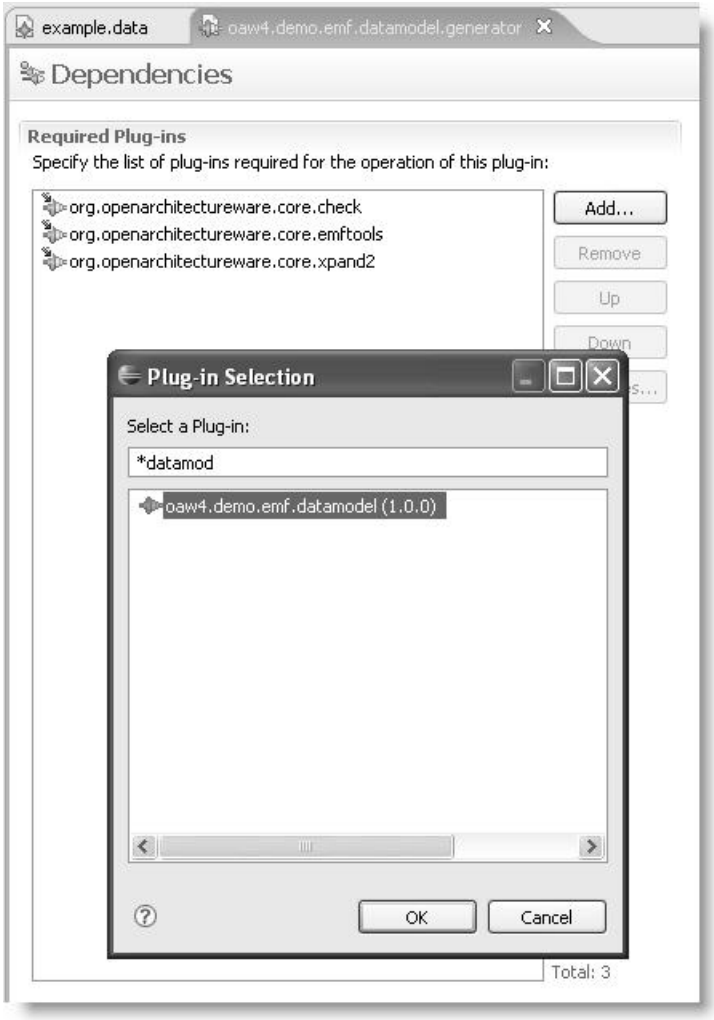


Figure 1.15. Add metamodel dependency

Do not forget to save the manifest file!

Now, you can run the workflow from within Eclipse:

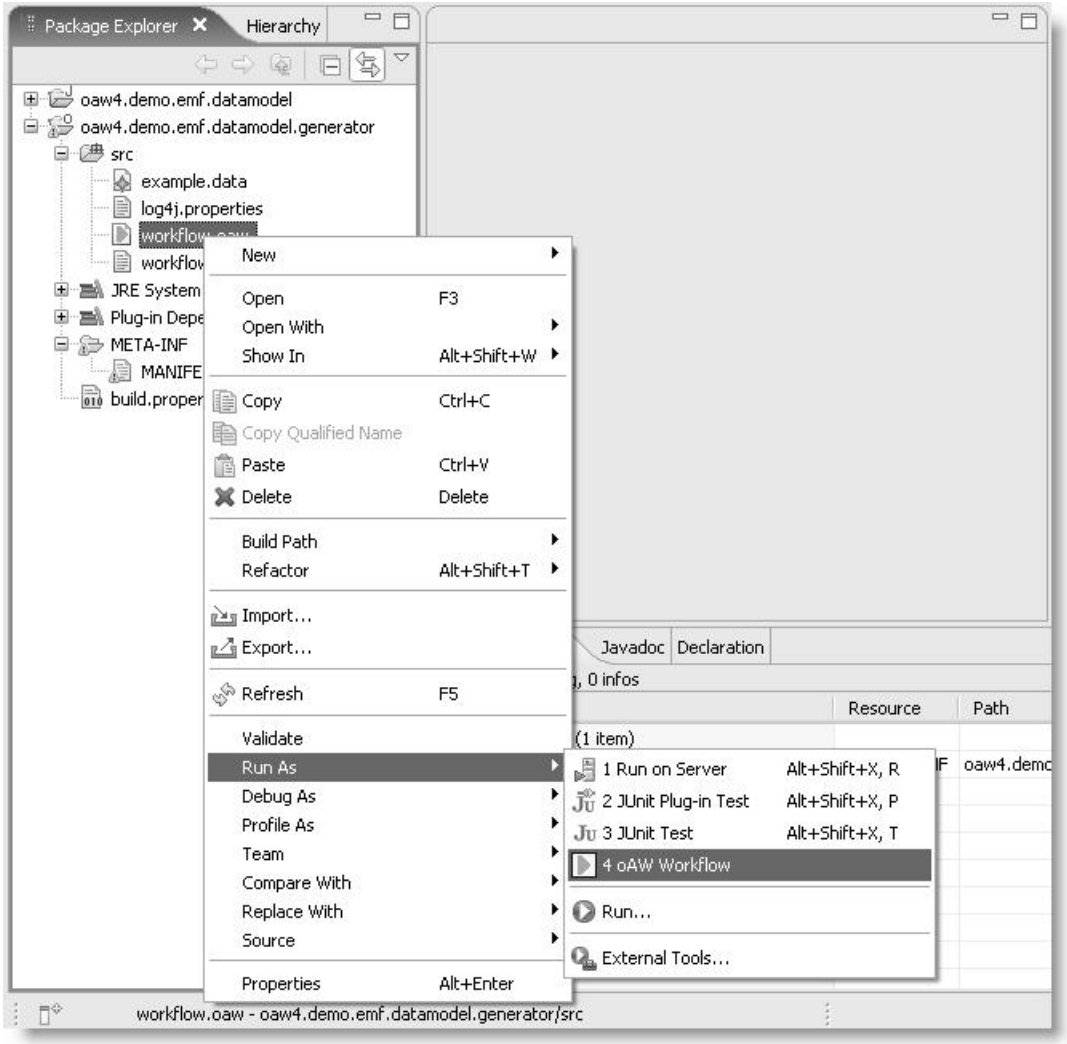


Figure 1.16. Sample data model

The following should be the output:

```
0    INFO  - ----------------------------------------------------------------------------
171  INFO  - openArchitectureWare 4.1.2, Build v20070314
171  INFO  - (c) 2005-2007 openarchitectureware.org and contributors
171  INFO  - ----------------------------------------------------------------------------
171  INFO  - running workflow: D:/oAW-emftutorial/oaw4.demo.emf.datamodel.generator/src/workflow.oaw
171  INFO  -
750  INFO  - xmiParser: file 'example.data' => slot 'model'
875  INFO  - workflow completed in 125ms!
```

1.8.3. Templates

No code is generated yet. This is not surprising, since we did not yet add any templates. Let us change this. Create a package templates in the `src`folder and within the package a file called `Root.xpt`.

The `Root.xpt` looks as follows. By the way, if you need to type the *guillemets* (« 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»
```

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.openarchitectureware.workflow.common.DirectoryCleaner" >
  <directories value="${srcGenPath}"/>
 </component>
```

Then, we start the generator component. Its configuration is slightly involved.

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

First of all, you have to define the metamodel. 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 has to be on 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 *Xpand*. 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 is 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 and that this generated code should be processed by a code beautifier:

```
  <outlet path="${srcGenPath}/">
   <postprocessor
    class="org.openarchitectureware.xpand2.output.JavaBeautifier"/>
  </outlet>
```

Now, we are almost done.

```
 </component>
</workflow>
```

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

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

1.8.4. 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;
}
```

1.9. Checking Constraints with the *Check* 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.

1.9.1. Defining the constraint

We start by defining the constraint itself. We create a new file called `checks.chk` in the `src` folder of our project. It is important that this file resides in the classpath! The file has the following content:

```
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 ) );
```

1.9.2. Integration into the workflow file

The following piece of XML is the workflow file we have 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 reading 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 metamodel 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 is why it has to be located in the classpath.

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

Finally, we have to tell the engine on which model or part of the model the checks should work. In general, you can use the `<expressionvalue="..."/>` 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 the content element of a specific slot, 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).

1.10. 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 *Xtend Language Documentation*, we provide an simple example here.

1.10.1. Expression Extensions

Assume we wanted to change the *Attribute*s 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 would 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 are called as their first parameter, a kind of "explicitly 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»
```

1.10.2. 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 is 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 its implementation:

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

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

Sometimes, it is 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 will 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()»
  // generated 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.

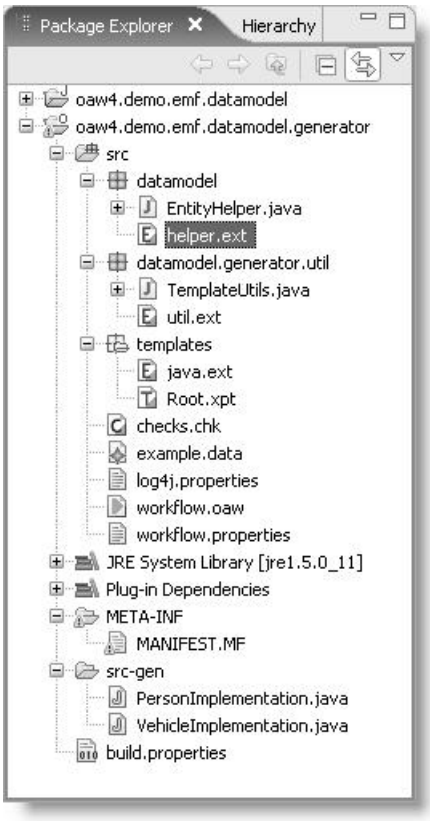


Figure 1.17. What has happened so far

1.11. Integrating Recipes

Let us assume, we wanted to allow developers to add their own business logic to the entities, maybe adding a couple of derived properties. In that case, we have to integrate the generated code with manually written fragments. Let us further assume that you – just like me – do not 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 us first change the generator accordingly.

1.11.1. Adjusting project settings

We will now use the Recipe framework of openArchitectureWare to achieve our task. Since this framework is an add-on we need to set up an additional dependency to it for our generator project. Otherwise the required classes will not be found.

Open the projects Manifest file `META-INF/MANIFEST.MF`, go to the *Dependencies* page and add the org.openarchitectureware.recipe.* plugins.

1.11.2. Adapting the existing generator

First, let us 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()»
   // generated 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:

```
// generated 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 is 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 verify that these "programming guidelines" have been met by developers.

1.11.3. 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 us look at what you need to do.

In order to simplify life, your recipe creation component should use 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 = EcoreUtil2.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 existence 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;
 }
}
```

1.11.4. Workflow Integration

Here is 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>
```

1.11.5. 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 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.

1.12. Transforming Models

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

1. An actual model *transformation* generates a completely new model – usually based on a different metamodel – from an input model. The transformation has no side effects with respect to the input model.

2. A model *modification* completes/extends/finishes/modifies a model. No additional model is created.

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

1.12.1. 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 directly implementing 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 execute this component, 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 superclass (`SimpleJavaTransformerComponent`) provides the slot setter and passes the object from that slot to the `doTransform` operation.

Chapter 2. Xtext Tutorial

2.1. Introduction

The purpose of this tutorial is to illustrate the definition of external DSLs using tools form the Eclipse Modeling Project (EMP). The main focus is on the *Xtext* framework. We will start by defining our own DSL in an *Xtext* grammar. Then we will use the *Xtext* framework to generate a parser, an *Ecore-based* metamodel and a textual editor for Eclipse. Afterwards we will see how to refine the DSL and its editor by means of *Xtend* extensions. Finally, we will learn how one can generate code out of textual models using the template language *Xpand*.

The actual content of this example is rather trivial – our DSL will describe entities with properties and references between them from which we generate Java classes according to the JavaBean conventions – a rather typical data model. In a real setting, we might also generate persistence mappings, etc. from the same models. We skipped this to keep the tutorial simple.

2.2. Setting up the Environment

To follow this tutorial, you will need to install the following components

- A Java 5 or 6 SDK. Download it at [java_download] or use another SDK that suits your environment.
- Eclipse SDK 3.3 aka Europa. You can download it from [eclipse_europa]. Install by simply unpacking the archive.
- openArchitectureWare 4.3. Download the ZIP file from [oaw_download] or point your eclipse update manager to [oaw_update_site].

2.3. Defining the DSL

2.3.1. Creating an Xtext Project

Xtext projects are based on the well-known Eclipse plug-in architecture. In fact, to create a new textual DSL with Xtext, you'll need up to three projects that depend on each other. But fear not - Xtext comes with a handy wizards to get you up and running in no time.

To create a new Xtext project,

- Start up Eclipse 3.3 with oAW 4.3 installed (see Section 2.2, “Setting up the Environment”) in a fresh workspace and close the welcome screen
- Select **File > New... > Project... Xtext Project**
- Specify the project settings in the wizard dialog. Since you started in a fresh workspace, the wizard should provide sensible defaults. See the Xtext reference documentation for a detailed overview of what all those settings mean.
- Click **Finish**

The wizard creates three projects, `my.dsl`, `my.dsl.editor`, and `my.dsl.generator`:

- **my.dsl** is the language project, in which we will define the grammar for our DSL. After running the Xtext generator, this model also contains a parser for the DSL and a metamodel backing the language.
- **my.dsl.editor** will contain the DSL editor. Since we have not yet defined a grammar, this project is still empty. It will be filled by the Xtext generator later.
- **my.dsl.generator** contains an openArchitectureWare code generator skeleton. Later in this tutorial, you will write a couple of templates that process models created with your DSL editor.

2.3.2. Defining the Grammar

Now that you have created a new Xtext project, you can define the grammar for your DSL. The grammar specifies the metamodel *and* the concrete syntax for your domain specific language. This allows for fast roundtrips and an incremental development of your language, as you will see later.

To specify the grammar, you will be using the Xtext grammar language. The Xtext documentation contains an extensive reference of all grammar elements. However, to make it easier for you to follow along this tutorial, we have included the relevant grammar rules here.

In this tutorial, we will develop a DSL for entities (since entities are something most developers know quite well).

- Open the Xtext grammar definition `my.dsl/src/mydsl.xtxt`
- And type in the following grammar definition:

```
Model:
  (types+=Type)*;            ❶

Type:
  DataType | Entity;         ❷

DataType:
  "datatype" name=ID;        ❸

Entity:
  "entity" name=ID "{"
    (features+=Feature)*     ❹
  "}";

Feature:
  type=[Type|ID] name=ID;    ❺
```

❶ The **Model** rule specifies that a model contains zero or more **Types**.

❷ The **Type** rule is an abstract rule. It specifies that a **Type** may either be a **DataType** or an **Entity**

❸ The **DataType** rule specifies that a **DataType** starts with the literal **datatype**, followed by a name. The name must comply with the (built-in) rule for identifiers (that is, only characters followed by zero or more characters mixed with any number of numbers are valid).

❹ The **Entity** rule specifies that an **Entity** starts with the literal **entity**, followed by the name of the entity (which, in turn, must be an identifier). An entity definition has a body which is surrounded by curly braces. The body may then contain any number (zero or more) of **Feature**s.

❺ A **Feature** has a reference to a **Type** and a name. The reference to **Type** is particularly interesting, because by appending the |*ID* modifier, we point out that the reference to **Type** will be determined by an ID.

Your grammar should now look like in Figure 2.1, “DSL grammar”.

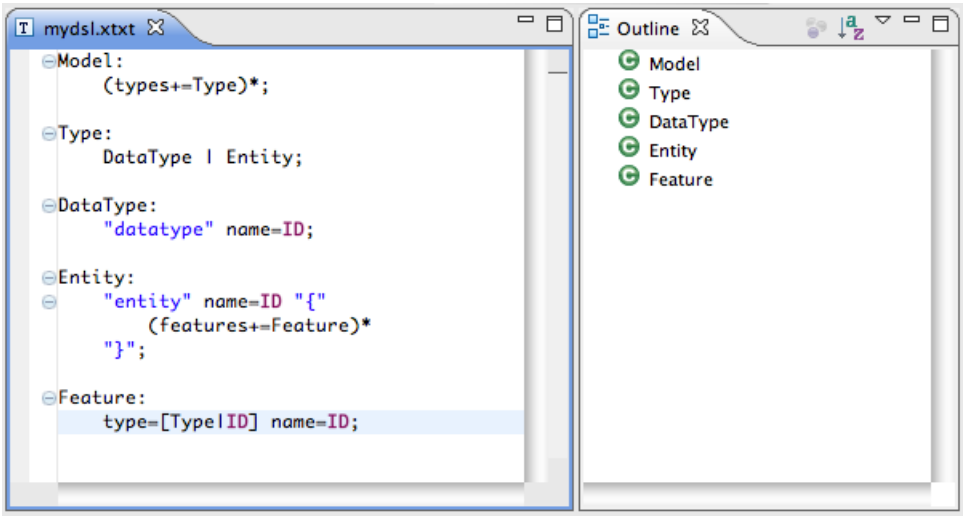


Figure 2.1. DSL grammar

2.3.3. Generating the DSL Editor

Having specified the grammar, we can now generate the DSL editor.

- Right-click inside the Xtext grammar editor to open the context menu.
- Select **Generate Xtext Artifacts** to generate the DSL parser, the corresonding metamodel and, last but not least, the DSL editor.

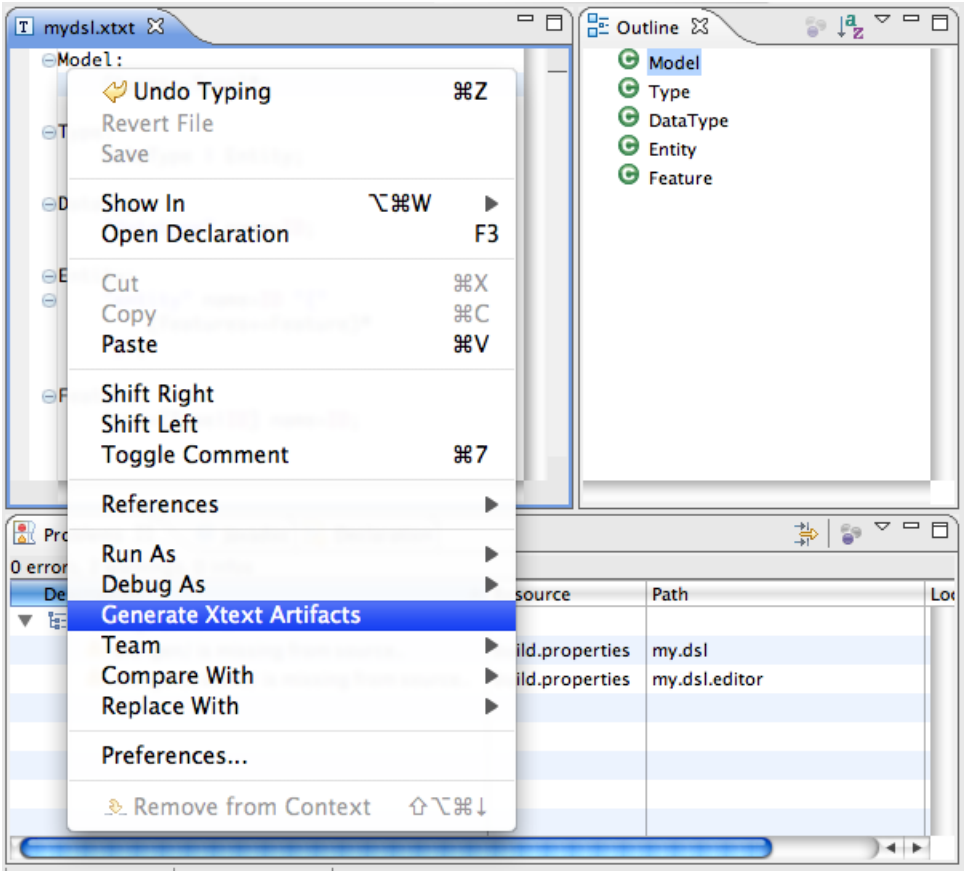


Figure 2.2. Generate Xtext artifacts

2.3.4. Deploying and Running the Editor

To run the generated editor, you have to deploy the DSL plug-ins to an Eclipse installation. For simplicity, we take the one we are already running.

- Choose **Export... > Deployable plug-ins and fragments...**
- The *Export* dialog appears. Select the three DSL plug-ins.
- Enter the path to your Eclipse installation. Make sure the selected directory contains the Eclipse executable and a folder named *plugins*. Usually, the directory is called *eclipse*.
- Choose **Finish**.
- Restart Eclipse.

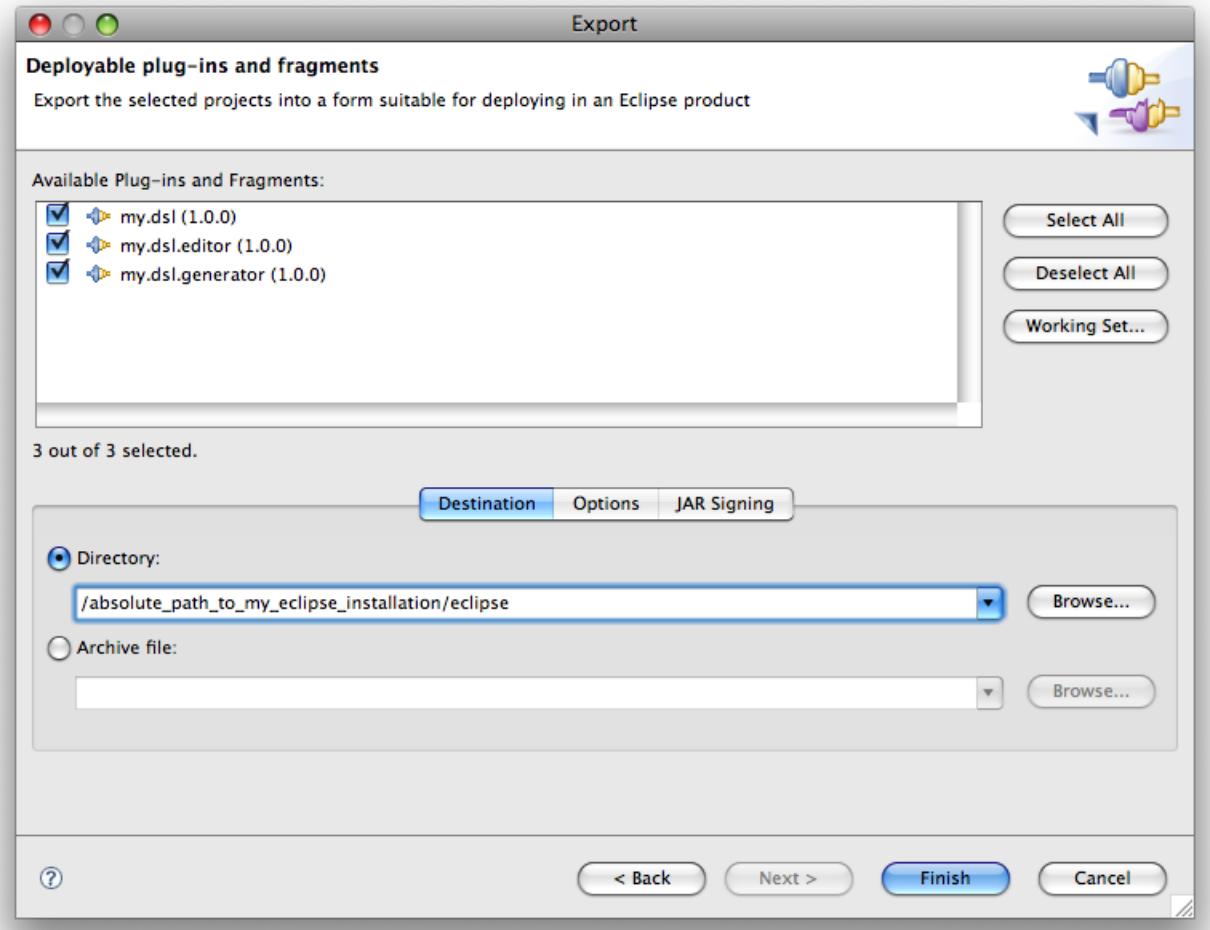


Figure 2.3. Deployment of the DSL plug-ins

2.3.5. Taking it for a spin

You should now see the same workspace as before. To check out the DSL editor, create a new *mydsl* project and model in the runtime instance:

- Select **File > New... > Other... > mydsl Project** to create a new *mydsl* project.
- Click **Next** to proceed to the next wizard page.
- Leave the project name as is (it should read *mydslproject*) and click **Finish** to create the project.

The wizard will now create a new project for you, including an empty sample model.

Key in the following model to see your editor in action. Note how the outline reflects the contents of your model. While typing, try using the content assist feature by pressing **CTRL-Space**.

```
datatype String
datatype String

entity Person {
 String name
 String lastName
 Address home
 Address business
}

entity Address {
 String street
 String zip
 String city
}
```

Xtext-based editors support a number of features right out of the box:

- Syntax coloring
- Code completion (press **CTRL-Space** to invoke)
- Navigation (either by holding the **CTRL** key and left-clicking an identifier or by pressing the **F3** key when the cursor is on an identifier)
- Find References (place the cursor on an identifier and press **CTRL-SHIFT-G**)
- Folding
- Outline
- Quick Outline (press **CTRL-O**)
- Syntax checking / Error markers

It is important to note that all those features have been derived from the grammar you defined earlier. If you make changes to the grammar, the generated tooling will reflect these changes as well, as you will see in a minute.

2.4. Refining the DSL

While Xtext-based DSL editors have a collection of great feature that come for free, they can be easily customized to your needs. In the following section, we will add some extra features that improve your editor's usability. As you will see, implementing those features will not cost us much effort.

2.4.1. Adjusting code completion

First, let's enhance code completion. Let's assume you want to assist the user of your editor in choosing the right data types. In most projects, there's probably only about five or six different data types in use, so why not provide them in the suggestion list for the `datatype` grammar rule?

To do so, open `my.dsl.editor/src/org.example.dsl/ContentAssist.ext` and insert the following lines at the end of the file:

```
/* proposals for Feature DataType::name */
List[Proposal] completeDataType_name❶(emf::EObject ctx, String prefix) :
 {
  newProposal("String"),
  newProposal("Date"),
  newProposal("Boolean"),
  newProposal("Long"),
  newProposal("int")
 };
```

❶ The name of he extension function must match the following rule: **complete<name of the metatype>_<name of the attribute to be completed>**. In this sample, the extension function will be invoked as soon as the user requests content assist for the name of a **DataType**.

After saving the extension file, the DSL editor display the new proposals:

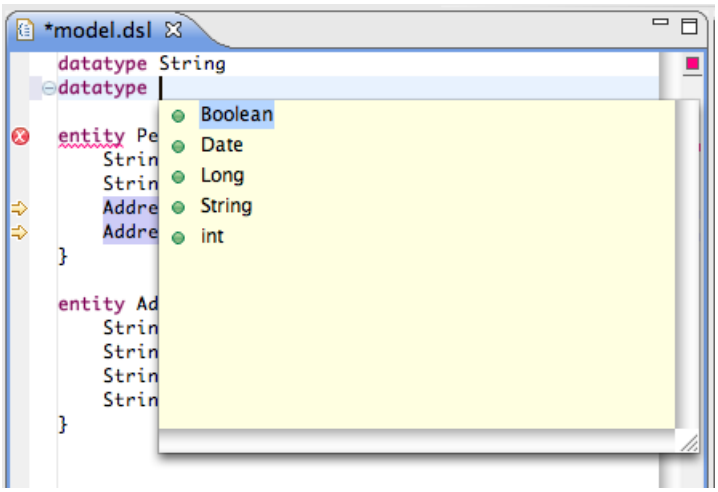


Figure 2.4. Enhanced content assist in action

2.4.2. Defining constraints for your model

You may have noticed that although the generated DSL editor detects syntax violations in your models, it is still possible to define illegal models, e.g. by defining several datatype definitions with the same name.

The *Check* language from the openArchitectuerWare stack can be used to define constraints that ensure the validity of your models.

Let's define a constraint that ensures that a model does not contain more than one data type with the same name. To do so, open `my.dsl/src/org.example.dsl/Checks.chk` and add the following contraint to the end of the file:

```
context Type ERROR "Duplicate type detected: " + this.name :
 allElements()❶.typeSelect(Type)❷.select(e|e.name == this.name).size ==1;
```

This constraint basically means the following:

- From the collection of *all model elements*,

- select all elements that are of type **Type** (i.e, all **DataTypes** and all **Entities**).
- Of the resulting collection, select all elements whose name equals the name of the current **Type**.
- Finally, check whether the size of the resulting collection is exactly one (1).

In other words: each model may only have exactly one **Type** with the same name.

After saving the check file, the DSL editor now issues an error if you enter two types with the same name:

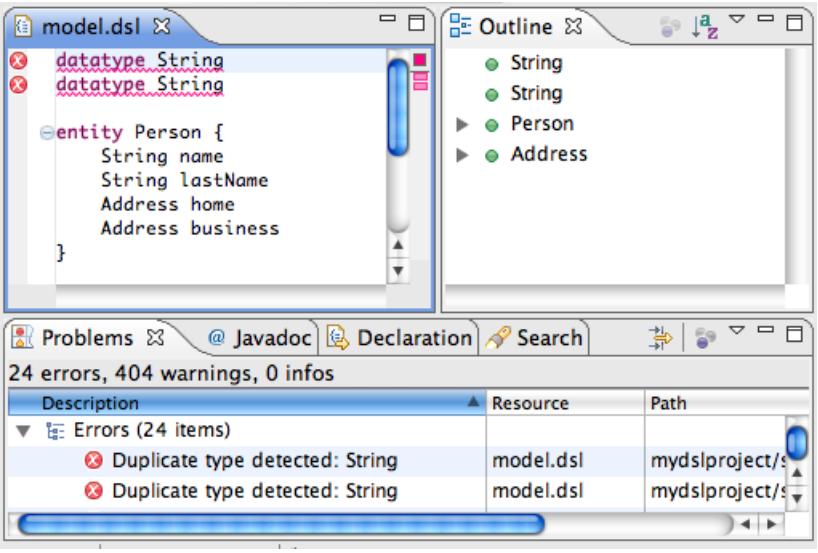


Figure 2.5. Constraint fails on duplicate data types

2.5. Generating code

Now, that we have a DSL, we may want to do something useful with it. DSLs are essentially small programming languages. A programming language has to be understandable by a computer. There are basically two ways to make a language "understandable" by a computer. The first one is to write a compiler which transforms expressions made in one language into another language, which is already understandable by a computer. For example, a Java compiler transforms Java programs to bytecode programs. Bytecode is understandable, because there are VMs which translate expressions in Java bytecode into more native instructions. This is usually done at runtime. Translating a language at runtime is called interpretation (ignoring special cases like just-in-time compilation here).

With Xtext, models one can either create a compiler (also called generator) or an interpreter. Although there are good reasons for both approaches, we will just discuss how one creates a generator in this tutorial.

2.5.1. Code generation with Xpand

The Xtext wizard already created a generator project for us. We are going to write an Xpand template, which generates simple JavaBeans from our entities. It is assumed, that there is a Java data type corresponding to the data types used in the models (e.g. **String**). So, we do not need to care about mapping data types.

So just open the Xpand template (**Main.xpt**) and modify it like this:

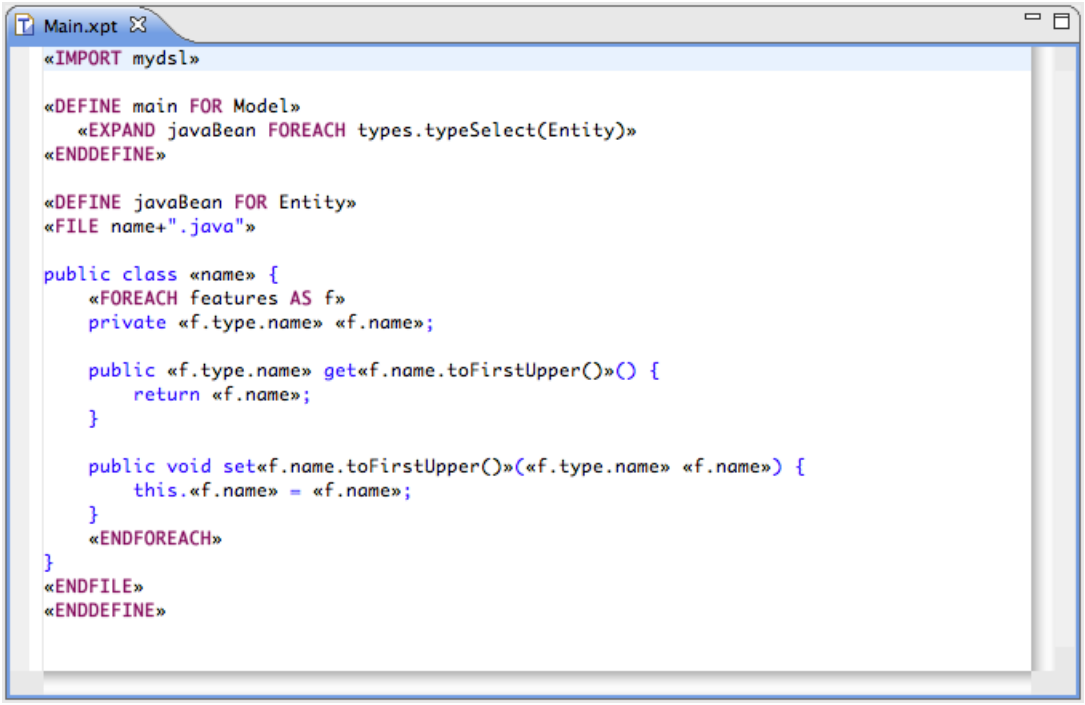


Figure 2.6. Xpand template

The definition **main** is invoked from the workflow file. It is declared for elements of type **mydsl::Model**, which corresponds to the root node of our DSL models. Within this definition, another definition (**javaBean**) is called (**«EXPAND javaBean...»**) for each model element (**...FOREACH...**) contained in the reference '**types**' of **Model** which is of type **Entity**. (**typeSelect(Entity)**).

The definition **javaBean** is declared for elements of type **Entity**. In this definition, we open a file («FILE ...»). The path name of the file is defined by an expression. In this case, it corresponds to the name of the entity suffixed with '`.java`'. It is going to be generated into the `src-gen` directory directly.

All text contained between **«FILE ...»** and **«ENDFILE»** will go to the new file. *Xpand* provides control statements (**FOR**, **IF**, **ELSEIF**, ...), as well as evaluation of expression, in order to create the desired code. See the openArchitectureWare reference documentation for details.

To see our template in action, we have to run the code generator:

- Locate the oAW workflow file *mydslproject.oaw* in your *mydslproject* plug-in.
- Right-click on it and choose **Run as > oAW Workflow** from the context menu.
- You can see the generator running and logging into the *Console* view.
- The result will be stored in a new source folder *src-gen* in the *mydslproject* project.

Resources

[java_download] *Sun's Java SDK*. http://java.sun.com/javase/downloads/index.jsp .

[eclipse_europa] *Eclipse 3.3*. http://www.eclipse.org/europa/ .

[oaw_download] *openArchitectureWare download page*. http://www.eclipse.org/gmt/oaw/download/ .

[oaw_update_site] *openArchitectureWare update site*. http://www.openarchitectureware.org/updatesite/milestone/site.xml .

[oaw_reference_documentation] *openArchitectureWare reference documentation*. http://www.eclipse.org/gmt/doc/ .

Chapter 3. oAW 4 Eclipse Integration

3.1. Introduction

This document describes the various functionalities that the openArchitectureWare plugins contribute to the Eclipse installation. It is intended as user instruction for the work with Eclipse. You need to read other documentation to understand the openArchitectureWare framework itself.

3.2. Installation

It is assumed that you already have installed the oAW core and the oAW plugin feature from the update site as described in the *Installation documentation*.

There are some more Eclipse plugins available that belong to special subprojects, such as the *Recipe* framework. They are not subject of this document and will be described in its specific documentation.

3.3. Overview

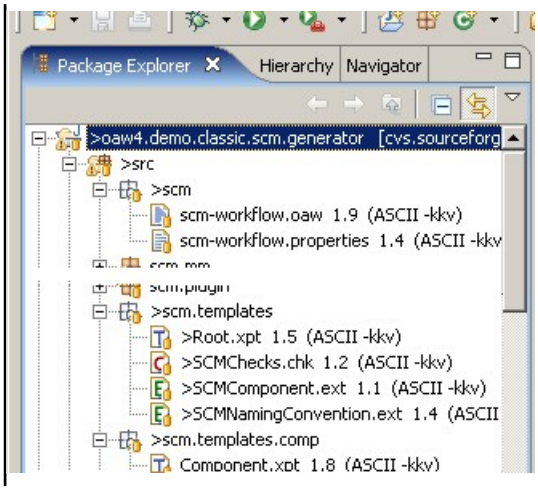
The openArchitectureWare core plugins provide editors for the different languages and a launch shortcut to start workflow files. Let us first have a look at the different oAW specific files.

3.4. File decorations

When you open Eclipse and import a project into the workspace you can see several file decorating images.

There are specific images for

- Workflow files (`.oaw` extension)
- *Xpand2* templates (`.xpt`extension)
- Extension files (`.ext` extension)
- *Check* constraints (`.chk` extension)

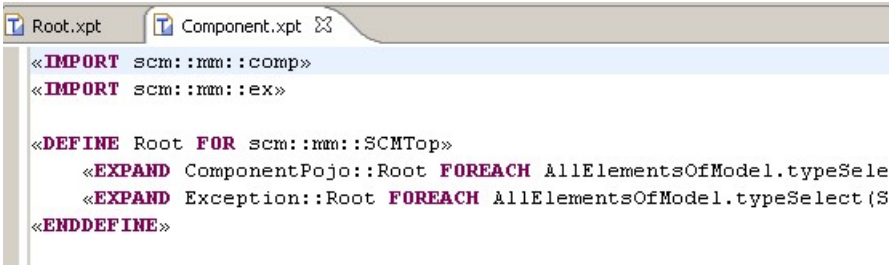


3.5. Editors

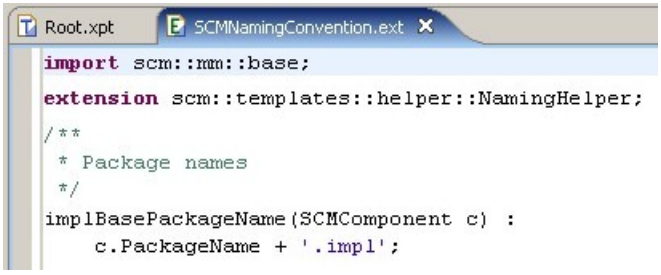
When you double-click on one of the above mentioned file types, special editors will open that provide appropriate syntax coloring.

3.5.1. Syntax coloring

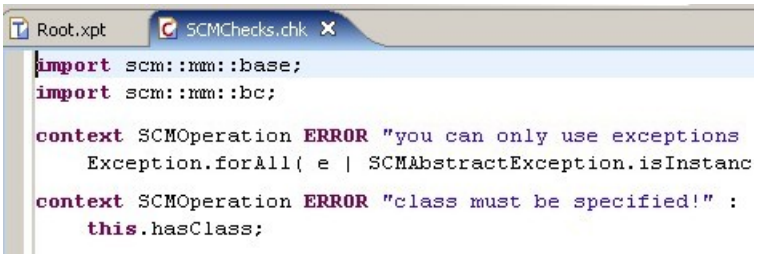
Here are examples for the *Xpand* editor:



for the Extensions editor:

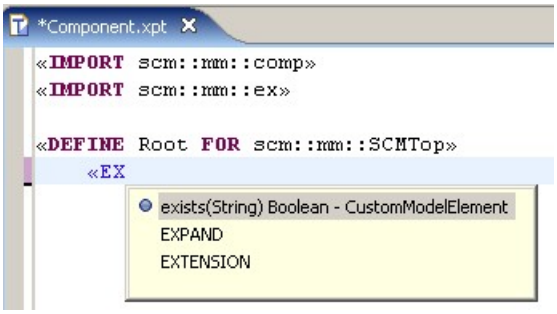


and for *Check* editor:



3.5.2. Code completion

The Editors provide extensive code completion support by pressing **Ctrl + Space** similar to what is known from the Java editor. Available types, properties, and operation, as well as extensions from `.ext` files will be found. The *Xpand* editor provides additionally support for the *Xpand* language statements.



3.5.3. *Xpand* tag delimiter creation support

In the *Xpand* editor there is an additional keystroke available to create the opening and closing tag brackets, the *guillemets* ("«" and "»").

Ctrl + < creates "«"

and

Ctrl + > creates "»"

3.6. Preference pages

3.6.1. Metamodel contributors

The openArchitectureWare framework version 4 supports several types of meta-metamodels.

From older versions, the classic UML metamodels may be known. Currently also JavaBeans metamodels and EMF based metamodels are supported out of the box.

Additional metamodel contributors can be registered through an extension point.

The editors need to know with what kind of metamodels you are working. Therefore, one can configure the metamodel contributors on workspace and on project level.

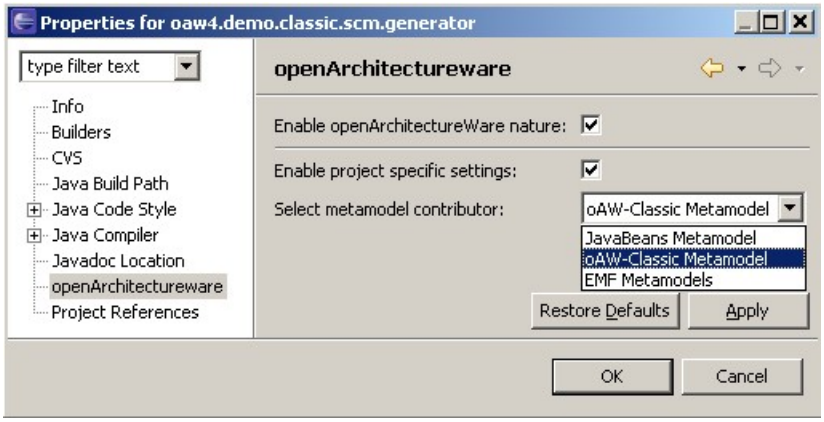
3.6.2. Global preferences

If you work always with the same type of metamodels, you can specify the metamodel contributors in the global preference page. It is available under Windows --> Preferences in the *openArchitectureWare* section.

3.6.3. Preferences per project

In the project property page there is also an *openArchitectureWare* section available.

Therein you can enable the openArchitectureWare nature (see below) and set project specific metamodel contributor settings.

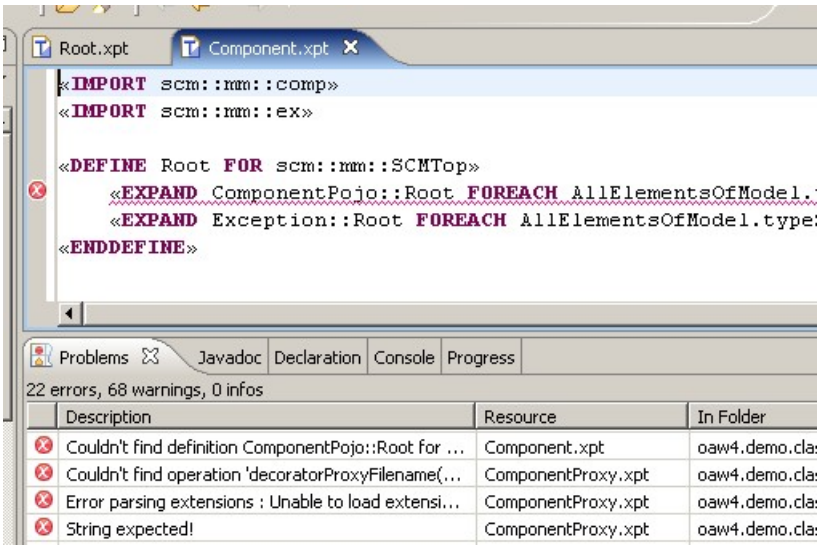


3.7. oAW Nature and oAW Builder

You have seen in the last screenshot that you can switch the *openArchitectureWare* nature on. If you do so, you enable analyzer support for all oAW specific file types in that project.

3.7.1. Problem markers

During the build process, all found problems are marked in the editors as well as listed in the *Problems* view.

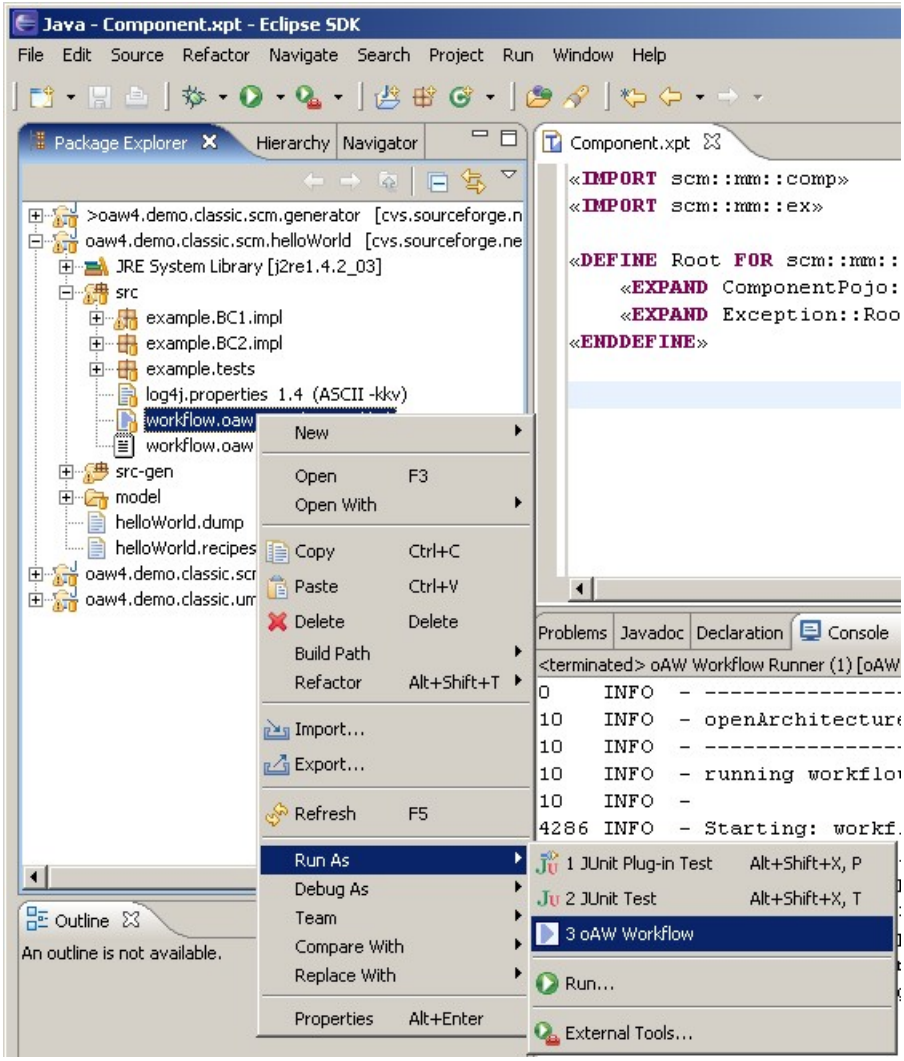


In the current implementation analyzes take place when *Eclipse* runs an incremental or full build. That means, problem markers are actualized when a file is saved, for instance. If you are in doubt about the actuality of problem markers, you should clean your project and let it rebuild again.

Note that if you change signatures of e.g. extensions the referencing artifacts (Xpand templates, etc.) are not analyzed automatically.

3.8. Running a workflow

You can start a workflow by right clicking on a workflow file (`*.oaw`) and selecting Run As -> oAW workflow.



Because it is a normal launch configuration, you could run or even debug it using the normal Eclipse functionality.

Part II. Reference

Chapter 4. Workflow Reference

4.1. Introduction

The oAW 4 workflow engine is a declarative configurable generator engine. It provides a simple, XML-based configuration language with which all kinds of generator workflows can be described. A generator workflow consists of a number of so-called workflow components that are executed sequentially in a single JVM.

4.2. Workflow components

At the heart of the workflow engine lies the `WorkflowComponent`. A workflow component represents a part of a generator process. Such parts are typically model parsers, model validators, model transformers and code generators. oAW 4 ships with different workflow components which should be used where suitable, but you can also implement your own. The only thing you have to do is to implement the `org.openarchitectureware.workflow.WorkflowComponent` interface:

```
public interface WorkflowComponent {

 /**
  * @param ctx
  *   current workflow context
  * @param monitor
  *   implementors should provide some feedback about the progress
  *   using this monitor
  * @param issues
  */
 public void invoke(WorkflowContext ctx, ProgressMonitor monitor, Issues issues);

 /**
  * Is called by the container after configuration so the
  * component can validate the configuration before invocation.
  *
  * @param issues -
  * implementors should report configuration issues to this.
  */
  public void checkConfiguration(Issues issues);

}
```

The `invoke()` operation performs the actual work of the component. `checkConfiguration` is used to check whether the component is configured correctly before the workflow starts. More on these two operations later.

A workflow description consists of a list of configured WorkflowComponents. Here is an example:

```
<workflow>
  <component class="my.first.WorkflowComponent">
   <aProp value="test"/>
  </component>
  <component class="my.second.WorkflowComponent">
   <anotherProp value="test2"/>
  </component>
  <component class="my.third.WorkflowComponent">
   <prop value="test"/>
  </component>
</workflow>
```

The workflow shown above consists of three different workflow components. The order of the declaration is important! The workflow engine will execute the components in the specified order. To allow the workflow engine to instantiate the workflow component classes, WorkflowComponent implementations must have a default constructor.

4.2.1. Workflow

A workflow is just a composite implementation of the `WorkflowComponent` interface. The `invoke` and `checkConfiguration` methods delegate to the contained workflow components.

The Workflow class declares an `addComponent()` method:

```
public void addComponent(WorkflowComponent comp);
```

which is used by the workflow factory in order to wire up a workflow (see next section *Workflow Configuration*).

4.2.2. Workflow Components with IDs

If you want your workflow components to have an ID (so that you can recognize its output in the log) you have to implement the interface `WorkflowComponentWithID` and the `setID()` and `getID()` operations. Alternatively, you can also extend the base class `AbstractWorkflowComponent`, which handles the ID setter/getter for you.

4.2.3. More convenience

There is another base class for workflow components called `AbstractWorkflowComponent2`. Its main feature is, that it has a property called skipOnErrors. If set to `true`, it will not execute if the workflow issues collection contains errors. This is convenient, if you want to be able to skip code generation when the preceding model verification finds errors. Note that instead of implementing `invoke(...)` and `checkConfiguration(...)`, subclasses of `AbstractWorkflowComponent2` have to implement `invokeInternal(...)` and `checkConfigurationInternal(...)`. This is necessary to allow the framework to intercept the invocation and stop it when there are errors in the workflow.

4.3. Workflow Configuration

A workflow is wired up using an XML configuration language based on the dependency injection pattern (DI). Here is an example (not working, just an example!):

```
<workflow>
 <property name='genPath' value='/home/user/target'/>
 <property name='model' value='/home/user/model.xmi'/>
 <component class='oaw.emf.XmiReader'>
  <model value='${model}'/>
 </component>
 <component class='oaw.xpand2.Generator'>
  <outlet>
   <path value='${genPath}'/>
  </outlet>
 </component>
</workflow>
```

The root element is named *workflow*, then there are some property declarations followed by the declaration of two components.

Here is a tree representation of the resulting Java object graph:

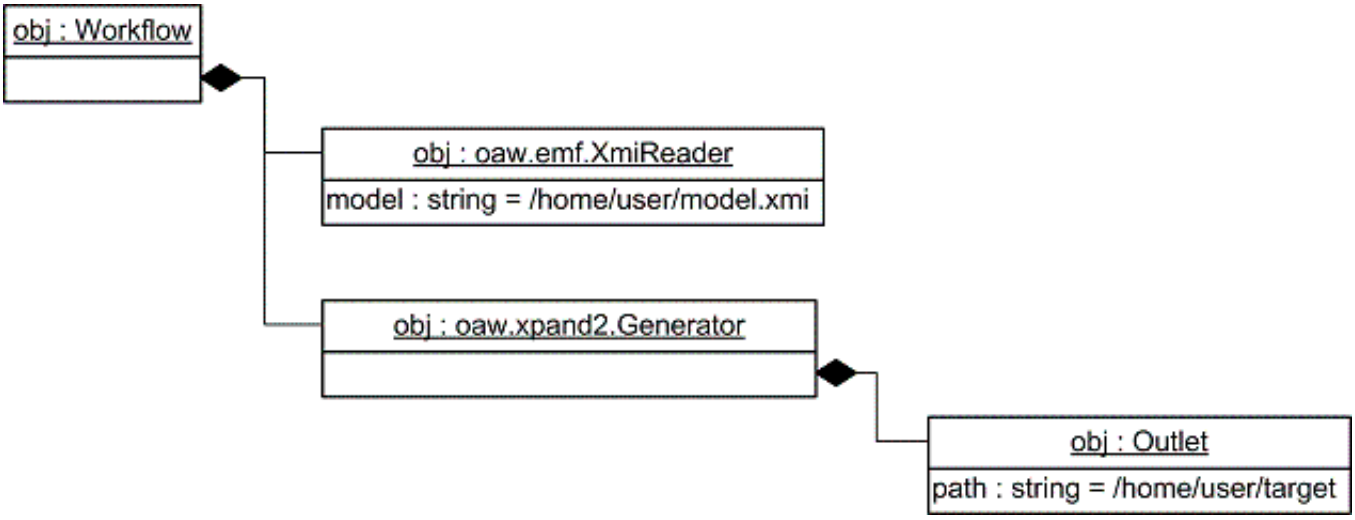


Figure 4.1. Java Object Graph

The configuration language expresses four different concepts:

4.3.1. Properties

Borrowing from Apache Ant, we use the concept of properties. Properties can be declared anywhere in a workflow file. They will be available after declaration.

We have two different kinds of properties

1. simple properties

2. property files

Here is an example:

```
<workflow>
 <property name='baseDir' value='./'/>
 <property file='${baseDir}/my.properties'/>
 <component
  class='my.Comp'
  srcDir='${baseDir}'
  modelName='${model}'
  pathToModel='${pathToModel}'/>
</workflow>
```

First, there is a simple property baseDir with the value "." defined. This property can be used in any attributes in the workflow file. The second property statement imports a property file. Property files use the well-known Java properties file syntax. There is one feature we added: You can use previously declared properties inside the properties file.

Example:

```
model = myModel
pathToModel = ${baseDir}/${model}.xmi
```

4.3.1.1. Components

The wired up object graph consists of so called components (A workflow component is a special kind of a component). A component is declared by an XML element. The name represents the property of the parent component holding this component.

Example:

```
<component class='MyBean'>
 <bean class='MyBean'/>
</component>
```

The Java class MyBean needs to have a corresponding property accessor. E.g.:

```
public class MyBean {
 ...
 public void setBean(MyBean b) {
  bean = b;
 }
 ...
}
```

There are currently the following possibilities for declaring the property accessors:

4.3.1.1.1. Accessor methods

As we have seen, one possibility for declaring a dependency is to declare a corresponding setter Method.

```
public void set<propertyname>(<PropertyType>; e)
```

If you want to set multiple multiple values for the same property, you should define an adder method.

```
public void add<propertyname>(<PropertyType> e)
```

In some cases you may want to have key value pairs specified. This is done by providing the following method:

```
public void put(Object k,Object v)
```

4.3.1.2. Component creation

The corresponding Java class (specified using the class attribute) needs to have a default constructor declared. If the class attribute is omitted, the Java class determined from the accessor method will be used. For the preceding example we could write

```
<component class='MyBean'>
 <bean/>
</component>
```

because the setter method uses the MyBean type as its parameter type. This is especially useful for more complex configurations of workflow components.

Note that we will probably add factory support in the future.

4.3.1.3. References

A component can have an attribute `id`. If this is the case, we can refer to this component throughout the following workflow configuration.

Example:

```
<workflow>
 <component class='my.Checker'>
  <metaModel id='mm' class='my.MetaModel'
   metaModelPackage='org.oaw.metamodel'/>
 </component>
 <component class='my.Generator'>
  <metaModel idRef='mm'/>
 </component>
 ...
</workflow>
```

In this example, an object with the id *mm* (an instance of *my.MetaModel*), is first declared and then referenced using the attribute idRef. Note that this object will only be instantiated once and then reused. It is not allowed to specify any other attributes besides idRef for object references.

4.3.1.4. Simple Parameters

Elements with only one attribute value are simple parameters. Simple parameters may not have any child elements.

Example:

```
<workflow>
 <component class='my.Checker' myParam='foo'>
  <anotherParam value='bar'/>
 </component>
```

As you can see, there are two ways to specify a simple paramter.

1. using an XML attribute

2. using a nested XML element with an attribute value

Both methods are equivalent, although declaring an attribute way saves a few keystrokes. However, the attributes `class`, `id`, and `file` are reserved so they cannot be used.

Parameters are injected using the same accessor methods as described for components. The only difference is, that they are not instantiated using a default constructor, but instead, they are using a so-called converter.

4.3.1.4.1. Converters

There are currently converter implementations registered for the following Java types:

1. `Object`

2. `String`

3. `String[]` (uses `s.split(',')`)

4. `Boolean` (both primitive and wrapper)

5. `Integer` (both primitive and wrapper)

4.3.1.5. Including other workflow files (also known as *cartridges*)

If an element has a property file, it is handled as an inclusion. Using an inclusion, one can inject a graph described in another workflow file. Here is an example:

file 1: mybean.oaw

```
<anyname class='MyClass'/>
```

file 2: workflow.oaw

```
<comp class='MyBean'>
 <bean file='mybean.oaw'/>
</comp>
```

One can pass properties and components into the included file in the usual way.

file 1: mybean.oaw

```
<anyname class='MyClass' aProp='${myParam}'>
 <bean idRef='myComponent'/>
</anyname>
```

file 2: workflow.oaw

```
<comp class='MyBean'>
 <bean file='mybean.oaw'>
  <myParam value='foo'/>
  <myComponent class='MyBean'/>
 </bean>
</comp>
```

As you can see, simple parameters are mapped to properties in the included workflow file, and components can be accessed using the idRef attribute.

Properties defined in the included workflow description will be overwritten by the passed properties.

The root element of a workflow description can have any name, because there is no parent defining an accessor method. Additionally, you have to specify the attribute class for a root element. There is only one exception: If the root element is named workflow the engine knows that it has to instantiate the type `org.openarchitectureware.Workflow`. Of course you can specify your own subtype of `org.openarchitectureware.Workflow` using the class attribute (if you need to for any reason).

4.3.1.6. InheritAll Feature

If you do not want to explicitely pass the parameters to an included workflow description, you can use the inheritAll attribute. This will make all the properties and beans that are visible to the actual workflow file also visible to the included workflow file.

```
<component file="my/included/workflow.oaw" inheritAll="true"/>
```

4.3.2. Component Implementation and Workflow Execution

This section describes how to implement workflow components, how they can communicate with each other and how the workflow execution can be controlled.

4.3.2.1. The Workflow Context

Workflow components have to communicate among each other. For example, if an XMIReader component reads a model that a constraint checker component wants to check, the model must be passed from the reader to the checker. The way this happens is as follows: In the `invoke` operation, a workflow component has access to the so-called *workflow context*. This context contains any number of named slots. In order to communicate, two components agree on a slot name, the first component puts an object into that slot and the second component takes it from there. Basically, slots are named variables global to the workflow. The slot names are configured from the workflow file. Here is an example:

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

 <component id="xmiParser"
  class="org.openarchitectureware.emf.XmiReader">
  <outputSlot value="model"/>
 </component>

 <component id="checker" class="datamodel.generator.Checker">
  <modelSlot value="model"/>
 </component>
</workflow>
```

As you can see, both these workflow components use the slot named *model*. Below is the (abbreviated) implementation of the `XmiReader`. It stores the model data structure into the workflow context in the slot whose name was configured in the workflow file.

```
public class XmiReader implements WorkflowComponent {

 private String outputSlot = null;

 public void setOutputSlot(String outputSlot) {
  this.outputSlot = outputSlot;
 }

 public void invoke(WorkflowContext ctx, ProgressMonitor monitor,
  Issues issues) {
  Object theModel = readModel();
  ctx.put( outputSlot, theModel );
 }

}
```

The checker component reads the model from that slot:

```
public class Checker implements WorkflowComponent {

 private String modelSlot;

 public final void setModelSlot( String ms ) {
  this.modelSlot = ms;
 }

 public final void invoke(WorkflowContext ctx,
  ProgressMonitor monitor, Issues issues) {

  Object model = ctx.get(modelSlot);
  check(model);
 }
}
```

4.3.2.2. Issues

Issues provide a way to report errors and warnings. There are two places, where issues are used in component implementations:

1. Inside the checkConfiguration operation, you can report errors or warnings. This operation is called before the workflow starts running.

2. Inside the invoke operation, you can report errors or warnings that occur during the execution of the workflow. Typical examples are constraint violations.

The Issues API is pretty straightforward: you can call addError and addWarning. The operations have three parameters: the reporting component, a message as well as the model element that caused the problem, if there is one. The operations are also available in a two-parameter version, omitting the first (reporting component) parameter.

4.3.2.3. Controlling the Workflow

There is an implicit way of controlling the workflow: if there are errors reported from any of the checkConfiguration operations of any workflow component, the workflow will not start running.

There is also an explicit way of terminating the execution of the workflow: if any invoke operation throws a `WorkflowInterruptedException` (a runtime exception) the workflow will terminate immediately.

4.3.2.3.1. Using Aspect Orientation wih Workflows

It is sometimes necessary to enhance existing workflow component declarations with additional properties. This is exemplified in the Template AOP example. To implement such an advice component, you have to extend the `AbstractWorkflowAdvice` class. You have to implement all the necessary getters and setters for the properties you want to be able to specify for that advice; also you have to implement the `weave()` operation. In this operation, which takes the advised component as a parameter, you have to set the advised parameters:

```
public class GeneratorAdvice extends AbstractWorkflowAdvice {

 private String advices;

 public String getAdvices() {
  return advices;
 }

 public void setAdvices(String advices) {
  this.advices = advices;
 }

 @Override
 public void weave(WorkflowComponent c) {
  Generator gen = (Generator)c;
  gen.setAdvices(advices);
 }

}
```

In the workflow file, things are straight forward: You have to specify the component class of the advice, and use the special property `adviceTarget` to identify the target component:

```
<workflow>

 <cartridge file="workflow.oaw"/>
  <component adviceTarget="generator"
   class="oaw.xpand2.GeneratorAdvice">
   <advices value="templates::Advices"/>
  </component>
 </workflow>
```

4.3.3. Invoking a workflow

If you have described your generator process in a workflow file, you might want to run it. There are different possibilities for doing so.

4.3.3.1. Starting the WorkflowRunner

The class `org.openarchitectureware.workflow.WorkflowRunner` is the main entry point if you want to run the workflow from the command line. Take a look at the following example:

```
java org.openarchitectureware.workflow.WorkflowRunner path/workflow.oaw
```

You can override properties using the -p option:

```
java org.openarchitectureware.workflow.WorkflowRunner -pbasedir=/base/ path/workflow.oaw
```

4.3.3.2. Starting with Ant

We also have an Ant task. Here is an example:

```
 <target name='generate'>
  <taskdef name="workflow" classname="org.openarchitectureware.workflow.ant.WorkflowAntTask"/>
   <workflow file='path/workflow.oaw'>
    <param name='baseDir' value='/base/'/>
   </workflow>
   ...
  </target>
```

The Workflow ant task extends the Java ant task. Therefore, you have all the properties known from that task (classpath, etc.).

4.3.3.3. Starting from you own code

You can also run the generator from your own application code. Two things to note:

1. the contents of the properties map override the properties defined in the workflow.

2. The slotContents map allows you to fill stuff into the workflow from your application. This is a typical use case: you run oAW from within your app because you already have a model in memory.

```
String wfFile = "somePath\\workflow.oaw";
Map properties = new HashMap();
Map slotContents = new HashMap();
new WorkflowRunner().run(wfFile ,
 new NullProgressMonitor(), properties, slotContents)
```

4.3.3.4. Starting from Eclipse

You can also run a workflow file from within Eclipse if you have installed the oAW plugins. Just right-click on the workflow file (`whatever.oaw`) and select Run As -> oAW Workflow. See the section *Running a workflow* in the documentation of the Eclipse integration of oAW for details.

Chapter 5. *Check* / *Xtend* / *Xpand* Reference

5.1. Introduction

The oAW4 generator framework provides textual languages, that are useful in different contexts in the MDSD process (e.g. checks, extensions, code generation, model transformation). Each oAW language (*Check*, *Xtend*, and *Xpand*) is built up on a common expression language and type system. Therefore, they can operate on the same models, metamodels and meta-metamodels and you do not need to learn the syntax again and again, because it is always the same.

The expressions framework provides a uniform abstraction layer over different meta-meta-models (e.g. EMF Ecore, Eclipse UML, JavaBeans, XML Schema etc.). Additionally, it offers a powerful, statically typed expressions language, which is used in the various textual languages.

5.2. Type System

The abstraction layer on API basis is called a type system. It provides access to built-in types and different registered metamodel implementations. These registered metamodel implementations offer access to the types they provide. The first part of this documentation describes the type system. The expression sub-language is described afterwards in the second part of this documentation. This differentiation is necessary because the type system and the expression language are two different things. The type system is a kind of reflection layer, that can be extended with metamodel implementations. The expression language defines a concrete syntax for executable expressions, using the type system.

The Java API described here is located in the org.openarchitectureware.type package and is a part of the subproject core-expressions.

5.2.1. Types

Every object (e.g. model elements, values, etc.) has a type. A type contains properties and operations. In addition it might inherit from other types (multiple inheritance).

5.2.1.1. Type Names

Types have a simple Name (e.g. `String`) and an optional namespace used to distingish between two types with the same name (e.g. `my::metamodel`). The delimiter for name space fragments is a double colon "`::`". A fully qualified name looks like this:

```
my::fully::qualified::MetaType
```

The namespace and name used by a specific type is defined by the corresponding `MetaModel` implementation. The `EmfMetaModel`, for instance, maps `EPackages` to namespace and `EClassifiers` to names. Therefore, the name of the Ecore element `EClassifier` is called:

```
ecore::EClassifier
```

If you do not want to use namespaces (for whatever reason), you can always implement your own metamodel and map the names accordingly.

5.2.1.2. Collection Type Names

The built-in type system also contains the following collection types: `Collection`, `List` and `Set`. Because the expressions language is statically type checked and we do not like casts and `ClassCastExceptions`, we introduced the concept of *parameterized types*. The type system does not support full featured generics, because we do not need them.

The syntax is:

```
Collection[my::Type]
List[my::Type]
Set[my::Type]
```

5.2.1.3. Features

Each type offers features. The type (resp. the metamodel) is responsible for mapping the features. There are three different kinds of features:

- Properties
- Operations
- Static properties

Properties are straight forward: They have a name and a type. They can be invoked on instances of the corresponding type. The same is true for *Operations*. But in contrast to properties, they can have parameters. *Static properties* are the equivalent to enums or constants. They must be invoked statically and they do not have parameters.

5.2.2. Built-In Types

As mentioned before, the expressions framework has several built-in types that define operations and properties. In the following, we will give a rough overview of the types and their features. We will not document all of the operations here, because the built-.in types will evolve over time and we want to derive the documentation from the implementation (model-driven, of course). For a complete reference, consult the generated API documentation (http://www.openarchitectureware.org/api/built-ins/).

5.2.2.1. `Object`

`Object` defines a couple of basic operations, like `equals()`. Every type has to extend `Object`.

5.2.2.2. `Void`

The `Void` type can be specified as the return type for operations, although it is not recommended, because whenever possible expressions should be free of side effects whenever possible.

5.2.2.3. Simple types (Data types)

The type system doesn't have a concept data type. Data types are just types. As in OCL, we support the following types: `String`, `Boolean`, `Integer`, `Real`.

- `String` : A rich and convenient `String` library is especially important for code generation. The type system supports the '+' operator for concatenation, the usual `java.lang.String` operations (`length()` , etc.) and some special operations (like `toFirstUpper()` , `toFirstLower()` , regular expressions, etc. often needed in code generation templates).

- `Boolean` : `Boolean` offers the usual operators (Java syntax): &&, ||, !, etc.
- `Integer` and `Real` : `Integer` and `Real` offer the usual compare operators (<,>,<=,>=) and simple arithmetics (+,-,*,/). Note that *`Integer` extends `Real`* !

5.2.2.4. Collection types

The type system has three different Collection types. `Collection` is the base type, it provides several operations known from `java.util.Collection`. The other two types (`List`, `Set`) correspond to their java.util equivalents, too.

5.2.2.5. Type system types

The type system describes itself, hence, there are types for the different concepts. These types are needed for reflective programming. To avoid confusion with metatypes with the same name (it is not unusual to have a metatype called `Operation`, for instance) we have prefixed all of the types with the namespace `oaw`. We have:

- `oaw::Type`
- `oaw::Feature`
- `oaw::Property`
- `oaw::StaticProperty`
- `oaw::Operation`

5.2.3. Metamodel Implementations (also known as Meta-Metamodels)

By default, the type system only knows the built-in types. In order to register your own metatypes (e.g. `Entity` or `State`), you need to register a respective metamodel implementation with the type system. Within a metamodel implementation the oAW type system elements (Type, Property, Operation) are mapped to an arbitrary other type system (Java reflections, Ecore or XML Schema).

5.2.3.1. Example JavaMetaModel

For instance, if you want to have the following JavaBean act as a metatype (i.e. your model contains instances of the type):

```
public class Attribute {
  private String name;
  private String type;
  public String getName() {
   return name;
  }
  public void setName(String name) {
   this.name = name;
  }
  public String getType() {
   return type;
  }
  public void setType(String type) {
   this.type = type;
  }
}
```

You need to use the `JavaMetaModel` implementation which uses the ordinary Java reflection layer in order to map access to the model.

So, if you have the following expression in e.g. *Xpand*:

```
myattr.name.toFirstUpper()
```

and `myattr` is the name of a local variable pointing to an instance of `Attribute`. The oAW type system asks the metamodel implementations, if they 'know' a type for the instance of Attribute. If you have the `JavaMetaModel` registered it will return an `oaw::Type` which maps to the underlying Java class. When the type is asked if it knows a property '`name`', it will inspect the Java class using the Java reflection API.

The JavaMetaModel implementation shipped with oAW can be configured with a strategy [GOF95-Pattern] in order to control or change the mapping. For instance, the JavaBeansStrategy maps getter and setter methods to simple properties, so we would use this strategy for the example above.

5.2.3.2. Eclipse IDE MetaModelContributors

You should know that for each `Metamodel` implementation you use at runtime, you need to have a so called `MetamodelContributor` extension for the plugins to work with. If you just use one of the standard metamodel implementations (EMF, UML2 or Java) you don't have to worry about it, since oAW is shipped with respective MetamodelContributors (see the corresponding docs for details). If you need to implement your own `MetamodelContributor` you should have a look at the Eclipse plug-in reference doc.

5.2.3.3. Configuring Metamodel implementations with the workflow

You need to configure your oAW language components with the respective metamodel implementations.

A possible configuration of the `Xpand2` generator component looks like this:

```
<component class="oaw.xpand2.Generator">
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelPackage value="my.generated.MetaModel1Package"/>
  </metaModel>
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelFile value="my/java/package/metamodel2.ecore"/>
  </metaModel>
  ...
</component>
```

In this example the `EmfMetaModel` implementation is configured two times. This means that we want to use two metamodels at the same time, both based on EMF. The metaModelPackage property is a property that is specific to the `EmfMetaModel` (located in the `core.emftools` project). It points to the generated `EPackages` interface. The second meta model is configured using the Ecore file. You do no need to have a generated Ecore model for oAW in order to work. The `EmfMetaModel` works with dynamic EMF models just as it works with generated EMF models.

5.2.4. Using different Metamodel implementations (also known as Meta-Metamodels)

With oAW you can work on different kinds of Model representations at the same time in a transparent manner. One can work with EMF models, XML DOM models, and simple JavaBeans in the same Xpand-Template. You just need to configure the respective MetaModel implementations.

If you want to do so you need to know how the type lookup works. Let us assume that we have an EMF metamodel and a model based on some Java classes. Then the following would be a possible configuration:

```
<component class="oaw.xpand2.Generator">
  <metaModel class="oaw.type.impl.java.JavaMetaModel"/>
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelFile value="my/java/package/metamodel.ecore"/>
  </metaModel>

  ...
</component>
```

When the oAW runtime needs to access a property of a given object, it asks the metamodels in the configured order. Let us assume that our model element is an instance of the Java type `org.eclipse.emf.ecore.EObject` and it is a dynamic instance of an EMF EClass `MyType`.

We have *three* Metamodels:

1. Built-Ins (always the first one)
2. JavaMetaModel
3. EMFMetaModel – metamodel.ecore

The first one will return the type `Object` (not `java.lang.Object` but `Object` of oAW). At this point the type `Object` best fits the request, so it will act as the desired type.

The second metamodel returns an oAW type called `oaw::eclipse::emf::ecore::EObject` The type system will check if the returned type is a specialization of the current 'best-fit' type (`Object`). It is, because it extends `Object` (Every metatype has to extend `Object`). At this time the type system assumes `oaw::eclipse::emf::ecore::EObject` to be the desired type.

The third metamodel will return `metamodel::MyType` which is the desired type. But unfortunately it doesn't extend `org::eclipse::emf::ecore::EObject` as it has nothing to do with those Java types. Instead it extends `emf::EObject` which extends `Object`.

We need to swap the configuration of the two metamodels to get the desired type.

```
<component class="oaw.xpand2.Generator">
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelFile value="my/java/package/metamodel.ecore"/>
  </metaModel>
  <metaModel class="oaw.type.impl.java.JavaMetaModel"/>

  ...
</component>
```

5.3. Expressions

The oAW expression sub-language is a syntactical mixture of Java and OCL. This documentation provides a detailed description of each available expression. Let us start with some simple examples.

Accessing a property:

```
myModelElement.name
```

Accessing an operation:

```
myModelElement.doStuff()
```

simple arithmetic:

```
1 + 1 * 2
```

boolean expressions (just an example:-)):

```
!('text'.startsWith('t') && ! false)
```

5.3.1. Literals and special operators for built-in types

There are several literals for built-in types:

5.3.1.1. `Object`

There are naturally no literals for object, but we have two operators:

equals:

```
obj1 == obj2
```

not equals:

```
obj1 != obj2
```

5.3.1.2. `Void`

The only possible instance of `Void` is the `null` reference. Therefore, we have one literal:

```
null
```

5.3.1.3. Type literals

The literal for types is just the name of the type (no '.class' suffix, etc.). Example:

```
String // the type string
my::special::Type // evaluates to the type 'my::special::Type'
```

5.3.1.4. StaticProperty literals

The literal for static properties (aka enum literals) is correlative to type literals:

```
my::Color::RED
```

5.3.1.5. `String`

There are two different literal syntaxes (with the same semantics):

```
'a String literal'
"a String literal" // both are okay
```

For Strings the expression sub-language supports the plus operator that is overloaded with concatenation:

```
'my element '+ ele.name +' is really cool!'
```

Note, that multi-line Strings are supported.

5.3.1.6. `Boolean`

The boolean literals are:

```
true
false
```

Operators are:

```
true && false // AND
true || false // OR
! true        // NOT
```

5.3.1.7. `Integer` and `Real`

The syntax for integer literals is as expected:

```
// integer literals
3
57278
// real literals
3.0
0.75
```

Additionally, we have the common arithmetic operators:

```
3 + 4  // addition
4 - 5  // subtraction
2 * 6  // multiplication
3 / 64 // divide
// Unary minus operator
- 42
- 47.11
```

Furthermore, the well known compare operators are defined:

```
4 > 5 // greater than
4 < 5 // smaller than
4 >= 23 // greater equals than
4 <= 12 // smaller equals than
```

5.3.1.8. Collections

There is a literal for lists:

```
{1,2,3,4} // a list with four integers
```

There is no other special concrete syntax for collections. If you need a set, you have to call the `toSet()` operation on the list literal:

```
{1,2,4,4}.toSet() // a set with 3(!) integers
```

5.3.2. Special Collection operations

Like OCL, the oAW expression sub-language defines several special operations on collections. However, those operations are not members of the type system, therefore you cannot use them in a reflective manner.

5.3.2.1. `select`

Sometimes, an expression yields a large collection, but one is only interested in a special subset of the collection. The expression sub-language has special constructs to specify a selection out of a specific collection. These are the `select` and `reject` operations. The select specifies a subset of a collection. A `select` is an operation on a collection and is specified as follows:

```
collection.select(v | boolean-expression-with-v)
```

`select` returns a sublist of the specified collection. The list contains all elements for which the evaluation of boolean-expression-with-v results is `true`. Example:

```
{1,2,3,4}.select(i | i >= 3) // returns {3,4}
```

5.3.2.2. `typeSelect`

A special version of a select expression is `typeSelect`. Rather than providing a boolean expression a class name is here provided.

```
collection.typeSelect(classname)
```

`typeSelect` returns that sublist of the specified collection, that contains only objects which are an instance of the specified class (also inherited).

5.3.2.3. `reject`

The `reject` operation is similar to the `select` operation, but with `reject` we get the subset of all the elements of the collection for which the expression evaluates to `false`. The `reject` syntax is identical to the `select` syntax:

```
collection.reject(v | boolean-expression-with-v)
```

Example:

```
{1,2,3,4}.reject(i | i >= 3) // returns {1,2}
```

5.3.2.4. `collect`

As shown in the previous section, the `select` and `reject` operations always result in a sub-collection of the original collection. Sometimes one wants to specify a collection which is derived from another collection, but which contains objects that are not in the original collection (it is not a sub-collection). In such cases, we can use a `collect` operation. The `collect` operation uses the same syntax as the `select` and `reject` and is written like this:

```
collection.collect(v | expression-with-v)
```

`collect` again iterates over the target collection and evaluates the given expression on each element. In contrast to `select`, the evaluation result is collected in a list. When an iteration is finished the list with all results is returned. Example:

```
namedElements.collect(ne | ne.name) // returns a list of strings
```

5.3.2.5. Shorthand for `collect` (and more than that)

As navigation through many objects is very common, there is a shorthand notation for collect that makes the expressions more readable. Instead of

```
self.employee.collect(e | e.birthdate)
```

one can also write:

```
self.employee.birthdate
```

In general, when a property is applied to a collection of Objects, it will automatically be interpreted as a `collect` over the members of the collection with the specified property.

The syntax is a shorthand for `collect`, if the feature does not return a collection itself. But sometimes we have the following:

```
self.buildings.rooms.windows // returns a list of windows
```

This syntax works, but one cannot express it using the `collect` operation in an easy way.

5.3.2.6. `forAll`

Often a boolean expression has to be evaluated for all elements in a collection. The `forAll` operation allows specifying a Boolean expression, which must be `true` for all objects in a collection in order for the `forAll` operation to return `true`:

```
collection.forAll(v | boolean-expression-with-v)
```

The result of `forAll` is `true` if `boolean-expression-with-v` is `true` for all the elements contained in a collection. If `boolean-expression-with-v` is `false` for one or more of the elements in the collection, then the `forAll` expression evaluates to `false`.

Example:

```
{3,4,500}.forAll(i | i < 10) // evaluates to false (500 < 10 is false)
```

5.3.2.7. `exists`

Often you will need to know whether there is at least one element in a collection for which a boolean is `true`. The exists operation allows you to specify a Boolean expression which must be `true` for at least one object in a collection:

```
collection.exists(v | boolean-expression-with-v)
```

The result of the exists operation is `true` if `boolean-expression-with-v` is `true` for at least one element of collection. If the `boolean-expression-with-v` is `false` for all elements in collection, then the complete expression evaluates to `false`.

Example:

```
{3,4,500}.exists(i | i < 10) // evaluates to true (e.g. 3 < 10 is true)
```

5.3.2.8. `sortBy`[1]

If you want to sort a list of elements, you can use the higher order function `sortBy`. The list you invoke the `sortBy` operation on, is sorted by the results of the given expression.

Example:

```
myListOfEntity.sortBy(entity | entity.name)
```

In the example the list of entities is sorted by the name of the entities. Note that there is no such `Comparable` type in oaw. If the values returned from the expression are instances of `java.util.Comparable` the `compareTo` method is used, otherwise `toString()` is invoked and the the result is used.

More Examples – all the following expressions return `true`:

```
{'C','B','A'}.sortBy(e | e) == {'A','B','C'}
{'AAA','BB','C'}.sortBy(e | e.length) == {'C','BB','AAA'}
{5,3,1,2}.sortBy(e | e) == {1,2,3,5}
{5,3,1,2}.sortBy(e | e - 2 * e) == {5,3,2,1}
...
```

5.3.3. `if` expression

There are two different forms of conditional expressions. The first one is the so-called *if expression*. Syntax:

```
condition ? thenExpression : elseExpression
```

Example:

```
name != null ? name : 'unknown'
```

5.3.4. `switch` expression

The other one is called *switch expression*. Syntax:

```
switch (expression) {
  (case expression : thenExpression)*
  default : catchAllExpression
}
```

The default part is mandatory, because `switch` is an expression, therefore it needs to evaluate to something in any case. Example:

```
switch (person.name) {
  case 'Hansen' : 'Du kanns platt schnacken'
  default : 'Du kanns mi nech verstohn!'
}
```

There is an abbreviation for *Boolean* expressions:

[1] since 4.1.2

```
switch {
  case booleanExpression : thenExpression
  default : catchAllExpression
}
```

5.3.5. Chain expression

Expressions and functional languages should be free of side effects as far as possible. But sometimes there you need invocations that do have side effects. In some cases expressions even don not have a return type (i.e. the return type is `Void`). If you need to call such operations, you can use the chain expression. Syntax:

```
anExpr ->
anotherExpr ->
lastExpr
```

Each expression is evaluated in sequence, but only the result of the last expression is returned. Example:

```
pers.setName('test') ->
pers
```

This chain expression will set the `name` of the person first, before it returns the person object itself.

5.3.6. `create` expression

The `create` expression is used to instantiate new objects of a given type:

```
new TypeName
```

5.3.7. `let` expression

The `let` expression lets you define local variables. Syntax is as follows:

```
let v = expression : expression-with-v
```

This is especially useful together with a chain- and a create expressions. Example:

```
let p = new Person :
  p.name('John Doe') ->
  p.age(42) ->
  p.city('New York') ->
  p
```

5.3.8. 'GLOBALVAR' expression

Sometimes you don't want to pass everything down the call stack by parameter. Therefore, we have the `GLOBALVAR` expression. There are two things you need to do, to use global variables within one of the openArchitectureWare languages (*Check*, *Xtend* or *Xpand*).

5.3.8.1. Using GLOBALVARS to configure workflows

Each workflow component using the expression framework (*Xpand*, *Check* and *Xtend*) can be configured with global variables. Here is an example:

```
<workflow>
  .... stuff
  <component class="oaw.xpand2.Generator">
   ... usual stuff (see ref doc)
   <globalVarDef name="MyPSM" value="slotNameOfPSM"/>
   <globalVarDef name="ImplClassSuffix" value="'Impl'"/>
  </component>
</workflow>
```

If you have injected global variables into the respective component, you can call them using the following syntax:

```
GLOBALVAR ImplClassSuffix
```

Note, we don't have any static type information. Therefore `Object` is assumed. So, you have to down cast the global variable to the intended type:

```
((String) GLOBALVAR ImplClassSuffix)
```

It is good practice to type it once, using an Extension and then always refer to that extension:

```
String implClassSuffix() : GLOBALVAR ImplClassSuffix;
// usage of the typed global var extension
ImplName(Class c) :
  name+implClassSuffix();
```

5.3.9. Multi methods (multiple dispatch)

The expressions language supports multiple dispatching. This means that when there is a bunch of overloaded operations, the decision which operation has to be resolved is based on the dynamic type of all parameters (the implicit '`this`' included).

In Java only the dynamic type of the '`this`' element is considered, for parameters the static type is used. (this is called single dispatch)

Here is a Java example:

```
class MyClass {
  boolean equals(Object o) {
   if (o instanceof MyClass) {
     return equals((MyClass)o);
   }
   return super.equals(o);
  }
  boolean equals(MyType mt) {
   //implementation...
  }
}
```

The method `equals(Object o)` would not have to be overwritten, if Java would support multiple dispatch.

5.3.10. Casting

The expression language is statically type checked. Although there are many concepts that help the programmer to have really good static type information, sometimes. one knows more about the real type than the system. To explicitly give the system such an information casts are available. *Casts are 100% static, so you do not need them, if you never statically typecheck your expressions!*

The syntax for casts is very Java-like:

```
((String)unTypedList.get(0)).toUpperCase()
```

5.4. *Check*

openArchitectureWare also provides a language to specify constraints that the model has to fulfill in order to be correct. This language is very easy to understand and use. Basically, it is built around the expression syntax that has been discussed in detail in the previous section. Constraints specified in the *Check* language have to be stored in files with the file extension `.chk`. Furthermore, these files have to be on the Java classpath, of course, in order to be found. Let us look at an example, in order to understand, what these constraints look like and what they do:

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

Now, let us look at the example line by line:

1. First, the metamodel has to be imported.

2. Then, the context is specified for which the constraint applies. In other words, after the `context` keyword, we put the name of the metaclass that is going to be checked by the constraint. Then, there follows either `ERROR` or `WARNING`, These keywords specify what kind of action will be taken in case the constraint fails:

   Table 5.1. Types of action for *Check* constraints

`WARNING`	If the constraint fails, the specified message is printed, but the workflow execution is not stopped.
`ERROR`	If the constraint fails, the specified message is printed and all further processing is stopped.

3. Now, the message that is put in case that the constraint fails is specified as a string. It is possible to include the value of attributes or the return value of functions into the message in order to make the message more clear. For example, it would be possible to improve the above example by rewriting it like this:

   ```
   import data;
   context Attribute ERROR
    "Name of '" + name + "too short. Names have to be more than one character long." :
    name.length > 1;
   ```

4. Finally, there is the condition itself, which is specified by an expression, which has been discussed in detail in the previous section. If this expression is `true`, the constraint is fulfilled.

Important

Please always keep in mind that the message that is associated with the constraint is printed, if the condition of the constraint is `false`! Thus, if the specified constraint condition is `true`, nothing will be printed out and the constraint will be fulfilled.

5.5. *Xtend*

Like the expressions sublanguage that summarizes the syntax of expressions for all the other textual languages delivered with the openArchitectureWare framework, there is another commonly used language called *Xtend*.

This language provides the possibility to define rich libraries of independent operations and no-invasive metamodel extensions based on either Java methods or oAW expressions. Those libraries can be referenced from all other textual languages, that are based on the expressions framework.

5.5.1. Extend files

An extend file must reside in the Java class path of the used execution context. Additionally it is file extension must be `*.ext`. Let us have a look at an extend file.

```
import my::metamodel;extension other::ExtensionFile;

/**
 * Documentation
 */
anExpressionExtension(String stringParam) :
 doingStuff(with(stringParam))
;

/**
 * java extensions are just mappings
 */
String aJavaExtension(String param) : JAVA
 my.JavaClass.staticMethod(java.lang.String)
;
```

The example shows the following statements:

1. import statements
2. extension import statements
3. expression or java extensions

5.5.2. Comments

We have single- and multi-line comments. The syntax for single line comments is:

```
// my comment
```

Multi line comments are written like this:

```
/* My multi line comment */
```

5.5.3. Import Statements

Using the import statement one can import name spaces of different types.(see expressions framework reference documentation).

Syntax is:

```
import my::imported::namespace;
```

Extend does not support static imports or any similar concept. Therefore, the following is incorrect syntax:

```
import my::imported::namespace::*; // WRONG! import my::Type; // WRONG!
```

5.5.4. Extension Import Statement

You can import another extend file using the extension statement. The syntax is:

```
extension fully::qualified::ExtensionFileName;
```

Note, that no file extension (`*.ext`) is specified.

5.5.4.1. Reexporting Extensions

If you want to export extensions from another extension file together with your local extensions, you can add the keyword 'reexport' to the end of the respective extension import statement.

```
extension fully::qualified::ExtensionFileName reexport;
```

5.5.5. Extensions

The syntax of a simple expression extension is as follows:

```
ReturnType extensionName(ParamType1 paramName1, ParamType2...): expression-using-params;
```

Example:

```
String getterName(NamedElement ele) : 'get'+ele.name.firstUpper();
```

5.5.5.1. Extension Invocation

There are two different ways of how to invoke an extension. It can be invoked like a function:

```
getterName(myNamedElement)
```

The other way to invoke an extension is through the "member syntax":

```
myNamedElement.getterName()
```

For any invocation in member syntax, the target expression (the member) is mapped to the first parameter. Therefore, both syntactical forms do the same thing.

It is important to understand that extensions are not members of the type system, hence, they are not accessible through reflection and you cannot specialize or overwrite operations using them.

The expression evaluation engine first looks for an appropriate operation before looking for an extension, in other words operations have higher precedence.

5.5.5.2. Type Inference

For most extensions, you do not need to specify the return type, because it can be derived from the specified expression. The special thing is, that the static return type of such an extension depends on the context of use.

For instance, if you have the following extension

```
asList(Object o): {o};
```

the invocation of

```
asList('text')
```

has the static type List[String]. This means you can call

```
asList('text').get(0).toUpperCase()
```

The expression is statically type safe, because its return type is derived automatically.

There is always a return value, whether you specify it or not, even if you specify explicitly '`Void`'.

See the following example.

```
modelTarget.ownedElements.addAllNotNull(modelSource.contents.duplicate())
```

In this example duplicate() dispatches polymorphically. Two of the extensions might look like:

```
Void duplicate(Realization realization):
  realization.Specifier().duplicate()->
  realization.Realizer().duplicate()
;

create target::Class duplicate(source::Class):
  ...
;
```

If a '`Realization`' is contained in the '`contents`' list of '`modelSource`', the '`Realizer`' of the '`Realization`' will be added to the '`ownedElements`' list of the '`modelTarget`'. If you do not want to add in the case that the contained element is a 'Realization' you might change the extension to:

```
Void duplicate(Realization realization):
  realization.Specifier().duplicate()->
  realization.Realizer().duplicate() ->
  {}
;
```

5.5.5.3. Recursion

There is only one exception: For recursive extensions the return type cannot be inferred, therefore you need to specify it explicitly:

```
String fullyQualifiedName(NamedElement n) : n.parent == null ? n.name :
  fullyQualifiedName(n.parent)+'::'+n.name
;
```

Recursive extensions are non-deterministic in a static context, therefore, it is necessary to specify a return type.

5.5.5.4. Cached Extensions

If you call an extension without side effects very often, you would like to cache the result for each set of parameters, in order improve the performance. You can just add the keyword '`cached`' to the extension in order to achieve this:

```
cached String getterName(NamedElement ele) :
  'get'+ele.name.firstUpper()
;
```

The `getterName` will be computed only once for each `NamedElement`.

5.5.5.5. Private Extensions

By default all extensions are public, i.e. they are visible from outside the extension file. If you want to hide extensions you can add the keyword 'private' in front of them:

```
private internalHelper(NamedElement ele) :
  // implementation....
;
```

5.5.6. Java Extensions

In some rare cases one does want to call a Java method from inside an expression. This can be done by providing a Java extension:

```
Void myJavaExtension(String param) :
  JAVA my.Type.staticMethod(java.lang.String)
;
```

The signature is the same as for any other extension. The implementation is redirected to a public static method in a Java class.

Its syntax is:

```
JAVA fully.qualified.Type.staticMethod(my.ParamType1,
                    my.ParamType2,
                    ...)
;
```

Note that you cannot use any imported namespaces. You have to specify the type, its method and the parameter types in a fully qualified way.

Example:

If you have defined the following Java extension:

```
Void dump(String s) :
  JAVA my.Helper.dump(java.lang.String)
;
```

and you have the following Java class:

```
package my;

public class Helper {
  public final static void dump(String aString) {
   System.out.println(aString);
  }
}
```

the expressions

```
dump('Hello world!')
'Hello World'.dump()
```

both result are invoking the Java method void dump(String aString)

5.5.7. Create Extensions (Model Transformation)

Since Version 4.1 the *Xtend* language supports additional support for model transformation. The new concept is called *create extension* and it is explained a bit more comprehensive as usual.

Elements contained in a model are usually referenced multiple times. Consider the following model structure:

```
   P
  / \
 C1 C2
  \ /
  R
```

A package P contains two classes C1 and C2. C1 contains a reference R of type C2 (P also references C2).

We could write the following extensions in order to transform an Ecore (EMF) model to our metamodel (Package, Class, Reference).

```
toPackage(EPackage x) :
  let p = new Package :
   p.ownedMember.addAll(x.eClassifiers.toClass()) ->
   p;

toClass(EClass x) :
  let c = new Class :
   c.attributes.addAll(x.eReferences.toReference()) ->
   c;

toReference(EReference x) :
  let r = new Reference :
   r.setType(x.eType.toClass()) ->
   r;
```

For an Ecore model with the above structure, the result would be:

```
   P
  / \
 C1 C2
 |
 R - C2
```

What happened? The C2 class has been created 2 times (one time for the package containment and another time for the reference R that also refers to C2). We can solve the problem by adding the 'cached' keyword to the second extension:

```
cached toClass(EClass x) :
  let c = new Class :
   c.attributes.addAll(c.eAttributes.toAttribute()) ->
   c;
```

The process goes like this:

1. start create P
    a. start create C1 (contained in P)

i. start create R (contained in C1)

A. start create C2 (referenced from R)

B. end (result C2 is cached)

ii. end R

b. end C1

c. start get cached C2 (contained in P)

2. end P

So this works very well. We will get the intended structure. But what about circular dependencies? For instance, C2 could contain a Reference R2 of type C1 (bidirectional references):

The transformation would occur like this:

1. start create P

a. start create C1 (contained in P)

i. start create R (contained in C1)

A. start create C2 (referenced from R)

I. start create R2 (contained in C2)

1. start create C1 (referenced from R1)... OOPS!

C1 is already in creation and will not complete until the stack is reduced. Deadlock! The problem is that the cache caches the return value, but C1 was not returned so far, because it is still in construction. The solution: create extensions

The syntax is as follows:

```
create Package toPackage(EPackage x) :
  this.classifiers.addAll(x.eClassifiers.toClass());

create Class toClass(EClass x) :
  this.attributes.addAll(x.eReferences.toReference());

create Reference toReference(EReference x) :
  this.setType(x.eType.toClass());
```

This is not only a shorter syntax, but it also has the needed semantics: The created model element will be added to the cache before evaluating the body. The return value is always the reference to the created and maybe not completely initialized element.

5.5.8. Calling Extensions From Java

The previous section showed how to implement Extensions in Java. This section shows how to call Extensions from Java.

```
// setup
XtendFacade f = XtendFacade.create("my::path::MyExtensionFile");

// use
f.call("sayHello",new Object[]{"World"});
```

The called extension file looks like this:

```
sayHello(String s) :
  "Hello " + s;
```

This example uses only features of the BuiltinMetaModel, in this case the "+" feature from the StringTypeImpl.

Here is another example, that uses the JavaBeansMetaModel strategy. This strategy provides as additional feature: the access to properties using the getter and setter methods.

For more information about type systems, see the *Expressions* reference documentation.

We have one JavaBean-like metamodel class:

```
package mypackage;
public class MyBeanMetaClass {
  private String myProp;
  public String getMyProp() { return myProp; }
  public void setMyProp(String s) { myProp = s;}
}
```

in addition to the built-in metamodel type system, we register the JavaMetaModel with the JavaBeansStrategy for our facade. Now, we can use also this strategy in our extension:

```
// setup facade

XtendFacade f = XtendFacade.create("myext::JavaBeanExtension");

// setup additional type system
JavaMetaModel jmm =
  new JavaMetaModel("JavaMM", new JavaBeansStrategy());

f.registerMetaModel(jmm);

// use the facade
MyBeanMetaClass jb = MyBeanMetaClass();
jb.setMyProp("test");
f.call("readMyProp", new Object[]{jb}));
```

The called extension file looks like this:

```
import mypackage;

readMyProp(MyBeanMetaClass jb) :
  jb.myProp
;
```

5.5.9. WorkflowComponent

With the additional support for model transformation, it makes sense to invoke *Xtend* within a workflow. A typical workflow configuration of the *Xtend* component looks like this:

```
<component class="oaw.xtend.XtendComponent">
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelFile value="metamodel1.ecore"/>
  </metamodel>
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelFile value="metamodel2.ecore"/>
  </metaModel>
  <invoke value="my::example::Trafo::transform(inputSlot)"/>
  <outputSlot value="transformedModel"/>
</component>
```

Note that you can mix and use any kinds of metamodels (not only EMF metamodels).

5.5.10. Aspect-Oriented Programming in *Xtend* (since 4.2)

Using the workflow engine, it is now possible to package (e.g. zip) a written generator and deliver it as a kind of black box. If you want to use such a generator but need to change some things without modifying any code, you can make use of around advices that are supported by *Xtend*.

The following advice is weaved around every invocation of an extension whose name starts with 'my::generator::':

```
around my::generator::*(*) :
 log('Invoking ' + ctx.name) -> ctx.proceed()
;
```

Around advices let you change behaviour in an non-invasive way (you do not need to touch the packaged extensions).

5.5.10.1. Join Point and Point Cut Syntax

Aspect orientaton is basically about weaving code into different points inside the call graph of a software module. Such points are called *join points*. In *Xtend* the join points are the extension invocations (Note that *Xpand* offers a similar feature, see the *Xpand* documentation).

One specifies on which join points the contributed code should be executed by specifying something like a 'query' on all available join points. Such a query is called a point cut.

```
around [pointcut] :
  expression;
```

A point cut consists of a fully qualified name and a list of parameter declarations.

5.5.10.1.1. Extensions Name

The extension name part of a point cut must match the fully qualified name of the definition of the join point. Such expressions are case sensitive. The asterisk character is used to specify wildcards. Some examples:

```
my::Extension::definition // extensions with the specified name
org::oaw::* //extensions prefixed with 'org::oaw::'
*Operation* // extensions containing the word 'Operation' in it.
* // all extensions
```

> ### Warning
>
> Be careful when using wildcards, because you will get an endless recursion, in case you weave an extension, which is called inside the advice.

5.5.10.1.2. Parameter Types

The parameters of the extensions that we want to add our advice to, can also be specified in the point cut. The rule is, that the type of the specified parameter must be the same or a supertype of the corresponding parameter type (the dynamic type at runtime) of the definition to be called.

Additionally, one can set the wildcard at the end of the parameter list, to specify that there might be none or more parameters of any kind.

Some examples:

```
my::Templ::extension() // extension without parameters
my::Templ::extension(String s) // extension with exactly one parameter of type String
my::Templ::extension(String s,*) // templ def with one or more parameters,
                 // where the first parameter is of type String
my::Templ::extension(*) // templ def with any number of parameters
```

5.5.10.1.3. Proceeding

Inside an advice, you might want to call the underlying definition. This can be done using the implicit variable `ctx`, which is of the type xtend::AdviceContext and provides an operation `proceed()` which invokes the underlying definition with the original parameters (Note that you might have changed any mutable object in the advice before).

If you want to control what parameters are to be passed to the definition, you can use the operation `proceed(List[Object] params)`. You should be aware, that in advices, no type checking is done.

Additionally, there are some inspection properties (like `name`, `paramTypes`, etc.) available.

5.5.10.2. Workflow configuration

To weave the defined advices into the different join points, you need to configure the `XtendComponent` with the qualified names of the Extension files containing the advices.

Example:

```
<component class="oaw.xtend.XtendComponent">
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelFile value="metamodel1.ecore"/>
  </metamodel>
  <metaModel class="oaw.type.emf.EmfMetaModel">
   <metaModelFile value="metamodel2.ecore"/>
  </metaModel>

  <invoke value="my::example::Trafo::transform(inputSlot)"/>
   <outputSlot value="transformedModel"/>
  <value="my::Advices,my::Advices2"/>
</component>
```

5.5.10.3. Model-to-Model transformation with *Xtend*

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.

You can download the example from http://www.eclipse.org/gmt/oaw/download.

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.

5.5.10.4. 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

5.5.10.5. The Transformation

The transformation, as mentioned above, can be found in the `Trafo.ext` file in the `test` package in the `src` folder. Let us 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 side effect 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 do not 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`. This 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 do not 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.

5.6. *Xpand2*

The openArchitectureWare framework contains a special language called *Xpand* that is used in templates to control the output generation. This documentation describes the general syntax and semantics of the *Xpand* language.

Typing the *guillemets* (« and ») used in the templates is supported by the Eclipse editor: which provides keyboard shortcuts with **Ctrl**+< and **Ctrl**+>.

5.6.1. Template files and encoding

Templates are stored in files with the extension `.xpt`. Template files must reside on the Java classpath of the generator process.

Almost all characters used in the standard syntax are part of *ASCII* and should therefore be available in any encoding. The only limitation are the tag brackets (*guillemets*), for which the characters "«" (Unicode `00AB`) and "»" (Unicode `00BB`) are used. So for reading templates, an encoding should be used that supports these characters (e.g. `ISO-8859-1` or `UTF-8`).

Names of properties, templates, namespaces etc. must only contain letters, numbers and underscores.

5.6.2. General structure of template files

Here is a first example of a template:

```
«IMPORT meta::model»
«EXTENSION my::ExtensionFile»

«DEFINE javaClass FOR Entity»
  «FILE fileName()»
   package «javaPackage()»;

   public class «name» {
     // implementation
   }
  «ENDFILE»
«ENDDEFINE»
```

A template file consists of any number of IMPORT statements, followed by any number of EXTENSION statements, followed by one or more DEFINE blocks (called definitions).

5.6.3. Statements of the *Xpand* language

5.6.3.1. IMPORT

If you are tired of always typing the fully qualified names of your types and definitions, you can import a namespace using the IMPORT statement.

```
«IMPORT meta::model»
```

This one imports the namespace `meta::model`. If your template contains such a statement, you can use the unqualified names of all types and template files contained in that namespace. This is similar to a Java import statement `import meta.model.*`.

5.6.3.2. EXTENSION

Metamodels are typically described in a structural way (graphical, or hierarchical, etc.) . A shortcoming of this is that it is difficult to specify additional behaviour (query operations, derived properties, etc.). Also, it is a good idea not to pollute the metamodel with target platform specific information (e.g. Java type names, packages, getter and setter names, etc.).

Extensions provide a flexible and convenient way of defining additional features of metaclasses. You do this by using the *Extend* language. (See the corresponding reference documentation for details)

An EXTENSION import points to the *Xtend* file containing the required extensions:

```
«EXTENSION my::ExtensionFile»
```

Note that extension files have to reside on the Java classpath, too. Therefore, they use the same namespace mechanism (and syntax) as types and template files.

5.6.3.3. DEFINE

The central concept of *Xpand* is the `DEFINE` block, also called a template. This is the smallest identifiable unit in a template file. The tag consists of a name, an optional comma-separated parameter list, as well as the name of the metamodel class for which the template is defined.

```
«DEFINE templateName(formalParameterList) FOR MetaClass»
  a sequence of statements
«ENDDEFINE»
```

To some extent, templates can be seen as special methods of the metaclass – there is always an implicit *this* parameter which can be used to address the "underlying" model element; in our example above, this model element is "MetaClass".

As in Java, a formal parameter list entry consists of the type followed by the name of that parameter.

The body of a template can contain a sequence of other statements including any text.

A full parametric polymorphism is available for templates. If there are two templates with the same name that are defined for two metaclasses which inherit from the same superclass, *Xpand* will use the corresponding subclass template, in case the template is called for the superclass. Vice versa, the template of the superclass would be used in case a subclass template is not available. Note that this not only works for the target type, but for all parameters. Technically, the target type is handled as the first parameter.

So, let us assume you have the following metamodel:

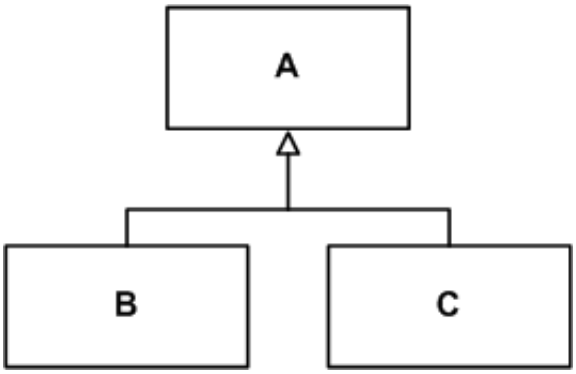


Figure 5.1. Sample metamodel

Assume further, you would have a model which contains a collection of `A`, `B` and `C` instances in the property `listOfAs`. Then, you can write the following template:

```
«DEFINE someOtherDefine FOR SomeMetaClass»
  «EXPAND implClass FOREACH listOfAs»
«ENDDEFINE»

«DEFINE implClass FOR A»
  // this is the code generated for the superclass A
«ENDDEFINE»

«DEFINE implClass FOR B»
  // this is the code generated for the subclass B
«ENDDEFINE»

«DEFINE implClass FOR C»
  // this is the code generated for the subclass C
«ENDDEFINE»
```

So for each `B` in the list, the template defined for `B` is executed, for each `C` in the collection the template defined for `C` is invoked, and for all others (which are then instances of `A`) the default template is executed.

5.6.3.4. FILE

The `FILE` statement redirects the output generated from its body statements to the specified target.

```
«FILE expression [outletName]»
  a sequence of statements
«ENDFILE»
```

The target is a file in the file system whose name is specified by the expression (relative to the specified target directory for that generator run). The expression for the target specification can be a concatenation (using the + operator). Additionally, you can specify an identifier (a legal Java identifier) for the name of the outlet. (See the configuration section for a description of outlets).

The body of a `FILE` statement can contain any other statements. Example:

```
«FILE InterfaceName + ".java"»
  package «InterfacePackageName»;

  /* generated class! Do not modify! */
  public interface «InterfaceName» {
   «EXPAND Operation::InterfaceImplementation FOREACH  Operation»
  }
«ENDFILE»


«FILE ImplName + ".java" MY_OUTLET»
  package «ImplPackageName»;

  public class «ImplName» extends «ImplBaseName»
              implements «InterfaceName» {
  //TODO: implement it
  }
«ENDFILE»
```

5.6.3.5. EXPAND

The `EXPAND` statement "expands" another `DEFINE` block (in a separate variable context), inserts its output at the current location and continues with the next statement. This is similar in concept to a subroutine call.

```
«EXPAND definitionName [(parameterList)]
  [FOR expression | FOREACH expression [SEPARATOR expression] ]»
```

The various alternative syntaxes are explained below.

5.6.3.5.1. Names

If the *definitionName* is a simple unqualified name, the corresponding `DEFINE` block must be in the same template file.

If the called definition is not contained in the same template file, the name of the template file must be specified. As usual, the double colon is used to delimit namespaces.

```
«EXPAND TemplateFile::definitionName FOR myModelElement»
```

Note that you would need to import the namespace of the template file (if there is one). For instance, if the template file resides in the java package `my.templates`, there are two alternatives. You could either write

```
«IMPORT my::templates»
...
«EXPAND TemplateFile::definitionName FOR myModelElement»
```

or

```
«EXPAND my::templates::TemplateFile::definitionName
    FOR myModelElement»
```

5.6.3.6. FOR vs. FOREACH

If `FOR` or `FOREACH` is omitted the other template is called `FOR this`.

```
«EXPAND TemplateFile::definitionName»
```

equals

```
«EXPAND TemplateFile::definitionName FOR this»
```

If `FOR` is specified, the definition is executed for the result of the target expression.

```
«EXPAND myDef FOR entity»
```

If `FOREACH` is specified, the target expression must evaluate to a collection type. In this case, the specified definition is executed for each element of that collection.

```
«EXPAND myDef FOREACH entity.allAttributes»
```

5.6.3.6.1. Specifying a Separator

If a definition is to be expanded `FOREACH` element of the target expression it is possible to specify a `SEPARATOR` expression:

```
«EXPAND paramTypeAndName FOREACH params SEPARATOR ','»
```

The result of the separator expression will be written to the output between each evaluation of the target definition (not *after* each one, but rather only in *between* two elements. This comes in handy for things such as comma-separated parameter lists).

An `EvaluationException` will be thrown if the specified target expression cannot be evaluated to an existing element of the instantiated model or no suitable `DEFINE` block can be found.

5.6.3.7. FOREACH

This statement expands the body of the `FOREACH` block for each element of the target collection that results from the expression. The current element is bound to a variable with the specified name in the current context.

```
«FOREACH expression AS variableName [ITERATOR iterName] [SEPARATOR expression]»
  a sequence of statements using variableName to access the
  current element of the iteration
«ENDFOREACH»
```

The body of a `FOREACH` block can contain any other statements; specifically `FOREACH` statements may be nested. If `ITERATOR` name is specified, an object of the type *xpand2::Iterator* (see API doc for details) is accessible using the specified name. The `SEPARATOR` expression works in the same way as the one for `EXPAND`.

Example:

```
«FOREACH {'A','B','C'} AS c ITERATOR iter SEPARATOR ','»
  «iter.counter1» : «c»
«ENDFOREACH»
```

The evaluation of the above statement results in the following text:

```
1 : A,
2 : B,
3 : C
```

5.6.3.8. IF

The `IF` statement supports conditional expansion. Any number of `ELSEIF` statements are allowed. The `ELSE` block is optional. Every `IF` statement must be closed with an `ENDIF`. The body of an `IF` block can contain any other statement, specifically, `IF` statements may be nested.

```
«IF expression»
  a sequence of statements
[ «ELSEIF expression» ]
  a sequence of statements ]
[ «ELSE»
  a sequence of statements ]
«ENDIF»
```

5.6.3.9. PROTECT

Protected Regions are used to mark sections in the generated code that shall not be overridden again by the subsequent generator run. These sections typically contain manually written code.

```
«PROTECT CSTART expression CEND expression ID expression (DISABLE)?»
  a sequence of statements
«ENDPROTECT»
```

The values of CSTART and CEND expressions are used to enclose the protected regions marker in the output. They should build valid comment beginning and end strings corresponding to the generated target language (e.g. *"/*"* and *"*/"* for Java). The following is an example for Java:

```
«PROTECT CSTART "/*" CEND "*/" ID ElementsUniqueID»
  here goes some content
«ENDPROTECT»
```

The ID is set by the `ID` expression and must be globally unique (at least for one complete pass of the generator).

Generated target code looks like this:

```
public class Person {
/*PROTECTED REGION ID(Person) ENABLED START*/
  This protected region is enabled, therefore the contents will
  always be preserved. If you want to get the default contents
  from the template you must remove the ENABLED keyword (or even
  remove the whole file :-))
/*PROTECTED REGION END*/
}
```

Protected regions are generated in enabled state by default. Unless you manually disable them, by removing the `ENABLED` keyword, they will always be preserved.

If you want the generator to generate disabled protected regions, you need to add the `DISABLE` keyword inside the declaration:

```
«PROTECT CSTART '/*' CEND '*/' ID this.name DISABLE»
```

5.6.3.10. LET

`LET` lets you specify local variables:

```
«LET expression AS variableName»
  a sequence of statements
«ENDLET»
```

During the expansion of the body of the `LET` block, the value of the expression is bound to the specified variable. Note that the expression will only be evaluated once, independent from the number of usages of the variable within the `LET` block. Example:

```
«LET packageName + "." + className AS fqn»
  the fully qualified name is: «fqn»;
«ENDLET»
```

5.6.3.11. ERROR

The `ERROR` statement aborts the evaluation of the templates by throwing an `XpandException` with the specified message.

```
«ERROR expression»
```

Note that you should use this facility very sparingly, since it is better practice to check for invalid models using constraints on the metamodel, and not in the templates.

5.6.3.12. Comments

Comments are only allowed outside of tags.

```
«REM»
  text comment
«ENDREM»
```

Comments may not contain a REM tag, this implies that comments are not nestable. A comment may not have a white space between the REM keyword and its brackets. Example:

```
«REM»«LET expression AS variableName»«ENDREM»
  a sequence of statements
«REM»  «variableName.stuff»
«ENDLET»«ENDREM»
```

5.6.3.13. Expression Statement

Expressions support processing of the information provided by the instantiated metamodel. Xpand provides powerful expressions for selection, aggregation, and navigation. Xpand uses the expressions sublanguage in almost any statement that we have seen so far. The expression statement just evaluates the contained expression and writes the result to the output (using the `toString()` method of `java.lang.Object`). Example:

```
public class «this.name» {
```

All expressions defined by the oArchitectureWare expressions sublanguage are also available in Xpand. You can invoke imported extensions. (See the *Expressions* and *Xtend language reference* for more details).

5.6.3.14. Controlling generation of white space

If you want to omit the output of superfluous whitespace you can add a minus sign just before any closing bracket. Example:

```
«FILE InterfaceName + ".java"-»
«IF hasPackage-»
package «InterfacePackageName»;
«ENDIF-»
...
«ENDFILE»
```

The generated file would start with two new lines (one after the `FILE` and one after the `IF` statement) if the minus characters had not been set.

In general, this mechanism works as follows: If a statement (or comment) ends with such a minus all preceding whitespace up to the newline character (excluded!) is removed. Additionally all following whitespace including the first newline character (\r\n is handled as one character) is also removed.

5.6.4. Aspect-Oriented Programming in *Xpand*

Using the workflow engine it is now possible to package (*e.g.* zip) a written generator and deliver it as a kind of black box. If you want to use such a generator but need to change some small generation stuff, you can make use of the `AROUND` aspects.

```
«AROUND qualifiedDefinitionName(parameterList)? FOR type»
  a sequence of statements
«ENDAROUND»
```

`AROUND` lets you add templates in an non-invasive way (you do not need to touch the generator templates). Because aspects are invasive, a template file containing `AROUND` aspects must be wrapped by configuration (see next section).

5.6.4.1. Join Point and Point Cut Syntax

AOP is basically about weaving code into different points inside the call graph of a software module. Such points are called *Join Points*. In *Xpand*, there is only one join point so far: a call to a definition.

You specify on which join points the contributed code should be executed by specifying something like a 'query' on all available join points. Such a query is called a *point cut*.

```
«AROUND [pointcut]»
  do stuff
«ENDAROUND»
```

A pointcut consists of a fully qualified name, parameter types and the target type.

5.6.4.1.1. Definition Name

The definition name part of a point cut must match the fully qualified name of the join point definition. Such expressions are case sensitive. The asterisk character is used to specify wildcards.

Some examples:

```
my::Template::definition // definitions with the specified name
org::oaw::* // definitions prefixed with 'org::oaw::'
*Operation* // definitions containing the word 'Operation' in it.
*           // all definitions
```

5.6.4.1.2. Parameter Types

The parameters of the definitions we want to add our advice to, can also be specified in the point cut. The rule is that the type of the specified parameter must be the same or a supertype of the corresponding parameter type (the dynamic type at runtime!) of the definition to be called.

Additionally, one can set a wildcard at the end of the parameter list, to specify that there might be an arbitrary number of parameters of any kind.

Some examples:

```
my::Templ::def() // templ def without parameters
my::Templ::def(String s) // templ def with exactly one parameter
             // of type String
my::Templ::def(String s,*) // templ def with one or more parameters,
              // where the first parameter is of type String
my::Templ::def(*) // templ def with any number of parameters
```

5.6.4.1.3. Target Type

Finally, we have to specify the target type. This is straightforward:

```
my::Templ::def() FOR Object// templ def for any target type
my::Templ::def() FOR Entity// templ def objects of type Entity
```

5.6.4.2. Proceeding

Inside an advice, you might want to call the underlying definition. This can be done using the implicit variable `targetDef`, which is of the type xpand2::Definition and which provides an operation `proceed()`that invokes the underlying definition with the original parameters (Note that you might have changed any mutable object in the advice before).

If you want to control, what parameters are to be passed to the definition, you can use the operation `proceed(Object target, List params)`. Please keep in mind that no type checking is done in this context.

Additionally, there are some inspection properties (like `name`, `paramTypes`, etc.) available.

5.6.5. Generator Workflow Component

This section describes the workflow component that is provided to perform the code generation, i.e. run the templates. You should have a basic idea of how the workflow engine works. (see *Workflow reference*). A simple generator component configuration could look as follows:

```
<component class="oaw.xpand2.Generator">
  <fileEncoding value="ISO-8859-1"/>
  <metaModel class="oaw.type.emf.EmfMetaModel">
    <metaModelPackage value="org.eclipse.emf.ecore.EcorePackage"/>
  </metaModel>
  <expand value="example::Java::all FOR myModel"/>

  <!-- aop configuration -->
  <advices value='example::Advices1, example::Advices2'/>

  <!--  output configuration -->
  <outlet path='main/src-gen'/>
  <outlet name='TO_SRC' path='main/src' overwrite='false'/>
  <beautifier class="oaw.xpand2.output.JavaBeautifier"/>
  <beautifier class="oaw.xpand2.output.XmlBeautifier"/>

  <!-- protected regions configuration -->
  <prSrcPaths value="main/src"/>
  <prDefaultExcludes value="false"/>
  <prExcludes value="*.xml"/>
</component>
```

Now, let us go through the different properties one by one.

5.6.5.1. Main configuration

The first thing to note is that the qualified Java name of the component is `org.openarchitectureware.xpand2.Generator2`. One can use the shortcut `oaw` instead of a preceding `org.openarchitectureware`. The workflow engine will resolve it.

5.6.5.2. Encoding

For *Xpand*, it is important to have the file encoding in mind because of the *guillemet* characters used to delimit keywords and property access. The `fileEncoding` property specifies the file encoding to use for reading the templates, reading the protected regions and writing the generated files. This property defaults to the default file encoding of your JVM.

5.6.5.3. Metamodel

The property `metaModel` is used to tell the generator engine on which metamodels the *Xpand* templates should be evaluated. One can specify more than one metamodel here. Metamodel implementations are required by the expression framework (see *Expressions*) used by *Xpand2*. In the example above we configured the Ecore metamodel using the *EMFMetaModel* implementation shipped with the core part of the openArchitectureWare 4 release.

A mandatory configuration is the `expand` property. It expects a syntax similar to that of the `EXPAND` statement (described above). The only difference is that we omit the `EXPAND` keyword. Instead, we specify the name of the property. Examples:

```
<expand value="Template::define FOR mySlot"/>
```

or:

```
<expand value="Template::define('foo') FOREACH {mySlot1,mySlot2}"/>
```

The expressions are evaluated using the workflow context. Each slot is mapped to a variable. For the examples above the workflow context needs to contain elements in the slots `'mySlot'`, `'mySlot1'` and `'mySlot2'`. It is

also possible to specify some complex expressions here. If, for instance, the slot `myModel` contains a collection of model elements one could write:

```
<expand value="Template::define FOREACH myModel.typeSelect(Entity)"/>
```

This selects all elements of type *Entity* contained in the collection stored in the `myModel` slot.

5.6.5.4. Output configuration

The second mandatory configuration is the specification of so called outlets (a concept borrowed from AndroMDA). Outlets are responsible for writing the generated files to disk . Example:

```
<component class="oaw.xpand2.Generator2">
  ...
  <outlet path='main/src-gen'/>
  <outlet name='TO_SRC' path='main/src' overwrite='false'/>
  ...
</component>
```

In the example there are two outlets configured. The first one has no name and is therefore handled as the default outlet. Default outlets are triggered by omitting an outlet name:

```
«FILE 'test/note.txt'»
# this goes to the default outlet
«ENDFILE»
```

The configured base path is '`main/src-gen`', so the file from above would go to '`main/src-gen/test/note.txt`'.

The second outlet has a `name` ('TO_SRC') specified. Additionally the flag `overwrite` is set to `false` (defaults to `true`). The following *Xpand* fragment

```
«FILE 'test/note.txt' TO_SRC»
# this goes to the TO_SRC outlet
«ENDFILE»
```

would cause the generator to write the contents to '`main/src/test/note.txt`' if the file does not already exist (the `overwrite` flag).

Another option called `append` (defaults to `false`) causes the generator to append the generated text to an existing file. If `overwrite` is set to `false` this flag has no effect.

5.6.5.5. Beautifier

Beautifying the generated code is a good idea. It is very important that generated code looks good, because developers should be able to understand it. On the other hand template files should look good, too. It is thus best practice to write nice looking template files and not to care how the generated code looks – and then you run a beautifier over the generated code to fix that problem. Of course, if a beautifier is not available, or if white space has syntactical meaning (as in Python), you would have to write your templates with that in mind (using the minus character before closing brackets as described in a preceding section).

The *Xpand* workflow component can be configured with multiple beautifiers:

```
<beautifier
  class="org.openarchitectureware.xpand2.output.JavaBeautifier"/>
<beautifier
  class="org.openarchitectureware.xpand2.output.XmlBeautifier"/>
```

These are the two beautifiers delivered with *Xpand*. If you want to use your own beautifier, you would just need to implement the following Java interface:

```
package org.openarchitectureware.xpand2.output;

public interface PostProcessor {
  public void beforeWriteAndClose(FileHandle handle);
  public void afterClose(FileHandle handle);
}
```

The `beforeWriteAndClose` method is called for each `ENDFILE` statement.

5.6.5.5.1. JavaBeautifier

The JavaBeautifier is based on the Eclipse Java formatter provides base beautifying for Java files.

5.6.5.5.2. XmlBeautifier

The XmlBeautifier is based on *dom4j* and provides a single option `fileExtensions` (defaults to "`.xml`, `.xsl`, `.wsdd`, `.wsdl`") used to specify which files should be pretty-printed.

5.6.5.6. Protected Region Configuration

Finally, you need to configure the protected region resolver, if you want to use protected regions.

```
<prSrcPaths value="main/src"/>
<prDefaultExcludes value="false"/>
<prExcludes value="*.xml"/>
```

The *prSrcPaths* property points to a comma-separated list of directories. The protected region resolver will scan these directories for files containing activated protected regions.

There are several file names which are excluded by default:

```
RCS, SCCS, CVS, CVS.adm, RCSLOG, cvslog.*, tags, TAGS, .make.state, .nse_depinfo, *~, #*,
.#*, ',*', _$*,*$, *.old, *.bak, *.BAK, *.orig, *.rej, .del-*, *.a, *.olb, *.o, *.obj,
 *.so, *.exe, *.Z,* .elc, *.ln, core, .svn
```

If you do not want to exclude any of these, you must set `prDefaultExcludes` to false.

```
<prDefaultExcludes value="false"/>
```

If you want to add additional excludes, you should use the prExcludes property.

```
<prExcludes value="*.xml,*.hbm"/>
```

> **Note**
>
> It is bad practice to mix generated and non-generated code in one artifact. Instead of using protected regions, you should try to leverage the extension features of the used target language (inheritance, inclusion, references, etc.) wherever possible. It is very rare that the use of protected regions is an appropriate solution.

5.6.6. Example for using Aspect-Oriented Programming in *Xpand*

This example shows how to use aspect-oriented programming techniques in *Xpand* templates. It is applicable to EMF based and *Classic* systems. However, we explain the idea based on the *emfExample* – hence you should read that before.

5.6.7. The Problem

There are many circumstances when template-AOP is useful. Here are two examples:

Scenario 1: Assume you have a nice generator that generates certain artifacts. The generator (or cartridge) might be a third party product, delivered in a single JAR file. Still you might want to adapt certain aspects of the generation process – *without modifying the original generator*.

Scenario 2: You are building a family of generators that can generate variations of the generate code, e.g. Implementations for different embedded platforms. In such a scenario, you need to be able to express those differences (variabilities) sensibly without creating a non-understandable chaos of *if* statements in the templates.

5.6.8. Example

To illustrate the idea of extending a generator without "touching" it, let us create a new project called `oaw4.demo.emf.datamodel.generator-aop`. The idea is that it will "extend" the original `oaw4.demo.emf.datamodel.generator` project introduced in the *emfExample*. So this new projects needs to have a project dependency to the former one.

5.6.8.1. Templates

An AOP system always needs to define a join point model; this is, you have to define, at which locations of a (template) program you can add additional (template) code. In Xpand, the join points are simply templates (i.e. *DEFINE .. ENDDEFINE*) blocks. An "aspect template" can be declared *AROUND* previously existing templates. If you take a look at the `oaw4.demo.emf.datamodel.generator` source folder of the project, you can find the `Root.xpt` template file. Inside, you can find a template called `Impl` that generates the implementation of the JavaBean.

```
«DEFINE Entity FOR data::Entity»
  «FILE baseClassFileName() »
   // generated at «timestamp()»
   public abstract class «baseClassName()» {
     «EXPAND Impl»
   }
  «ENDFILE»
«ENDDEFINE»

«DEFINE Impl FOR data::Entity»
  «EXPAND GettersAndSetters»
«ENDDEFINE»

«DEFINE Impl FOR data::PersistentEntity»
  «EXPAND GettersAndSetters»
  public void save() {

  }
«ENDDEFINE»
```

What we now want to do is as follows: Whenever the *Impl* template is executed, we want to run an additional template that generates additional code (for example, some kind of meta information for frameworks – the specific code is not important for the example here).

So, in our new project, we define the following template file:

```
«AROUND Impl FOR data::Entity»
  «FOREACH attribute AS a»
   public static final AttrInfo «a.name»Info = new AttrInfo(
     "«a.name»", «a.type».class );
  «ENDFOREACH»
  «targetDef.proceed()»
«ENDAROUND»
```

So, this new template wraps around the existing template called `Impl` It first generates additional code and then forwards the execution to the original template using `targetDef.proceed()`. So, in effect, this is a `BEFORE` advice. Moving the `proceed` statement to the beginning makes it an `AFTER` advice, omitting it, makes it an override.

5.6.8.2. Workflow File

Let us take a look at the workflow file to run this generator:

```
<workflow>
  <cartridge file="workflow.oaw"/>
  <component adviceTarget="generator"
       id="reflectionAdvice"
       class="oaw.xpand2.GeneratorAdvice">
   <advices value="templates::Advices"/>
  </component>
</workflow>
```

Mainly, what we do here, is to call the original workflow file. It has to be available from the classpath. After this cartridge call, we define an additional workflow component, a so called *advice component*. It specifies *generator* as its *adviceTarget*. That means, that all the properties we define inside this advice component will be added to the component referenced by name in the *adviceTarget* instead. In our case, this is the generator. So, in effect, we add the <advices value="templates::Advices" /> to the original generator component (without invasively modifying its own definition). This contributes the advice templates to the generator.

5.6.8.3. Running the new generator

Running the generator produces the following code:

```
public abstract class PersonImplBase {
  public static final AttrInfo
   nameInfo = new AttrInfo("name", String.class);
  public static final AttrInfo
   name2Info = new AttrInfo("name2", String.class);
  private String name;
  private String name2;

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

  public String getName() {
   return this.name;
  }

  public void setName2(String value) {
   this.name2 = value;
  }

  public String getName2() {
   return this.name2;
  }
}
```

5.6.9. More Aspect Orientation

In general, the syntax for the *AROUND* construct is as follows:

```
<<AROUND fullyQualifiedDefinitionNameWithWildcards
   (Paramlist (*)?) FOR TypeName>>
  do Stuff
<<ENDAROUND>>
```

Here are some examples:

```
<<AROUND *(*) FOR Object>>
```

matches all templates

```
<<AROUND *define(*) FOR Object>>
```

matches all templates with *define* at the end of its name and any number of parameters

```
<<AROUND org::oaw::* FOR Entity>>
```

matches all templates with namespace *org::oaw::* that do not have any parameters and whose type is Entity or a subclass

```
<<AROUND *(String s) FOR Object>>
```

matches all templates that have exactly one `String` parameter

```
<<AROUND *(String s,*) FOR Object>>
```

matches all templates that have at least one `String` parameter

```
<<AROUND my::Template::definition(String s) FOR Entity>>
```

matches exactly this single definition

Inside an `AROUND`, there is the variable `targetDef`, which has the type `xpand2::Definition`. On this variable, you can call `proceed`, and also query a number of other things:

```
<<AROUND my::Template::definition(String s) FOR String>>
  log('invoking '+<<targetDef.name>>+' with '+this)
  <<targetDef.proceed()>>
<<ENDAROUND>>
```

5.7. Built-in types API documentation

5.7.1. `Object`

Super type: none

Table 5.2. Properties

Type	Name	Description
`oaw::Type`	`metaType`	Returns the meta type of the project.

Table 5.3. Operations

Return type	Name	Description
`Boolean`	`<` (`Object`)	
`Boolean`	`!=` (`Object`)	
`Boolean`	`>=` (`Object`)	
`Boolean`	`<=` (`Object`)	
`Boolean`	`>` (`Object`)	
`Integer`	`compareTo` (`Object`)	Compares this object with the specified object for order. It returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
`Boolean`	`==` (`Object`)	
`String`	`toString` ()	Returns the String representation of this object (calling toString() method of Java).

5.7.2. `String`

Super type: `Object`

Table 5.4. Properties

Type	Name	Description
`Integer`	`length`	The length of this string.

Table 5.5. Operations

Return type	Name	Description
`String`	`toUpperCase` ()	Converts all of the characters in this string to upper case using the rules of the default locale (from Java)
`String`	`toLowerCase` ()	Converts all of the characters in this string to lower case using the rules of the default locale (from Java)
`List`	`split` (`String`)	Splits this string around matches of the given regular expression (from Java 1.4)
`String`	`trim` ()	Returns a copy of the string, with leading and trailing whitespace omitted. (from Java 1.4).
`String`	`+` (`Object`)	Concatenates two strings.
`String`	`replaceAll` (`String`, `String`)	Replaces each substring of this string that matches the given regular expression with the given replacement.
`String`	`subString` (`Integer`, `Integer`)	Tests if the string ends with the specified prefix.
`Boolean`	`endsWith` (`String`)	Tests if the string ends with the specified prefix.
`Integer`	`asInteger` ()	Returns an Integer object holding the value of the specified string (from Java 1.5)
`Boolean`	`contains` (`String`)	Tests if the string contains the specified substring.
`String`	`toFirstUpper` ()	Converts the first character in this string to upper case using the rules of the default locale (from Java)
`String`	`toFirstLower` ()	Converts the first character in this string to lower case using the rules of the default locale (from Java)
`String`	`replaceFirst` (`String`, `String`)	Replaces the first substring of this string that matches the given regular expression with the given replacement.
`Boolean`	`startsWith` (`String`)	Tests if this string starts with the specified prefix.
`List`	`toCharList` ()	Splits this string into a List[String] containing strings of length 1
`Boolean`	`matches` (`String`)	Tells whether or not this string matches the given regular expression. (from Java 1.4)

5.7.3. `Integer`

Super type: `Real`

Table 5.6. Operations

Return type	Name	Description
`Boolean`	`<= (Object)`	
`Integer`	`+ (Integer)`	
`Integer`	`* (Integer)`	
`Boolean`	`> (Object)`	
`List`	`upTo (Integer, Integer)`	Returns a list of integers starting with the value of the target expression, up to the value of the first parameter, incremented by the second parameter. E.g. '1.upTo(10, 2)' evaluates to {1,3,5,7,9}
`Boolean`	`!= (Object)`	
`Integer`	`/ (Integer)`	
`Integer`	`- ()`	
`List`	`upTo (Integer)`	Returns a list of integers starting with the value of the target expression, up tothe value of the specified Integer, incremented by one. e.g. '1.upTo(5)' evaluates to {1,2,3,4,5}
`Boolean`	`< (Object)`	
`Integer`	`- (Integer)`	
`Boolean`	`== (Object)`	
`Boolean`	`>= (Object)`	

5.7.4. `Boolean`

Super type: `Object`

Table 5.7. Operations

Return type	Name	Description
`Boolean`	`! ()`	

5.7.5. `Real`

Super type: `Object`

Table 5.8. Operations

Return type	Name	Description
`Boolean`	`==(Object)`	
`Boolean`	`!=(Object)`	
`Boolean`	`<=(Object)`	
`Real`	`/(Real)`	
`Boolean`	`<(Object)`	
`Real`	`*(Real)`	
`Boolean`	`>(Object)`	
`Real`	`-()`	
`Real`	`-(Real)`	
`Boolean`	`>=(Object)`	
`Real`	`+(Real)`	

5.7.6. `Collection`

Super type: `Object`

Table 5.9. Properties

Type	Name	Description
`Integer`	`size`	Returns the size of this Collection.
`Boolean`	`isEmpty`	Returns true if this Collection is empty.

Table 5.10. Operations

Return type	Name	Description
`Collection`	`remove` (`Object`)	Removes the specified element from this collection if contained (modifies it!) and returns this Collection.
`Set`	`toSet` ()	Converts this collection to a set.
`List`	`toList` ()	Converts this collection to a list.
`Boolean`	`containsAll` (`Collection`)	Returns true if this collection contains each element contained in the specified collection. otherwise false.
`Collection`	`removeAll` (`Object`)	Removes all elements contained in the specified collection if contained (modifies it!) and returns this collection.
`Collection`	`addAll` (`Collection`)	Adds all elements to the collection (modifies it!) and returns this collection.
`String`	`toString` (`String`)	Concatenates each contained element (using `toString()`), separated by the specified string.
`Boolean`	`contains` (`Object`)	Returns true if this collection contains the specified object. otherwise false.
`Set`	`intersect` (`Collection`)	Returns a new Set, containing only the elements contained in this and the specified collection.
`Set`	`without` (`Collection`)	Returns a new Set, containing all elements from this collection without the elements from specified collection.
`List`	`flatten` ()	Returns a flattened list.
`Set`	`union` (`Collection`)	Returns a new set, containing all elements from this and the specified collection.
`Collection`	`add` (`Object`)	Adds an element to the collection (modifies it!) and returns this collection.

5.7.7. `List`

Super type: `Collection`

Table 5.11. Operations

Return type	Name	Description
`Object`	`last` ()	
`Object`	`first` ()	
`Integer`	`indexOf` (`Object`)	
`Object`	`get` (`Integer`)	
`List`	`withoutLast` ()	
`List`	`withoutFirst` ()	

5.7.8. `Set`

Super type: `Collection`

5.7.9. `oaw::Type`

Super type: `Object`

Table 5.12. Properties

Type	Name	Description
`String`	`name`	
`Set`	`superTypes`	
`Set`	`allProperties`	
`Set`	`allFeatures`	
`Set`	`allOperations`	
`Set`	`allStaticProperties`	
`String`	`documentation`	

Table 5.13. Operations

Return type	Name	Description
`oaw::Property`	`getProperty` (`String`)	
`oaw::Operation`	`getOperation` (`String`, `List`)	
`oaw::StaticProperty`	`getStaticProperty` (`String`)	
`oaw::Feature`	`getFeature` (`String`, `List`)	
`Boolean`	`isAssignableFrom` (`oaw::Type`)	
`Boolean`	`isInstance` (`Object`)	
`Object`	`newInstance` ()	

5.7.10. `oaw::Feature`

Super type: `Object`

Table 5.14. Properties

Type	Name	Description
`oaw::Type`	`owner`	
`oaw::Type`	`returnType`	
`String`	`documentation`	
`String`	`name`	

5.7.11. `oaw::Property`

Super type: `oaw::Feature`

Table 5.15. Operations

Return type	Name	Description
`Void`	`set` (`Object`, `Object`)	
`Object`	`get` (`Object`)	

5.7.12. `oaw::Operation`

Super type: `oaw::Feature`

Table 5.16. Operations

Return type	Name	Description
`Object`	`evaluate` (`Object`, `List`)	
`List`	`getParameterTypes` ()	

5.7.13. `oaw::StaticProperty`

Super type: `oaw::Feature`

Table 5.17. Operations

Return type	Name	Description
`Object`	`get` ()	Returns the static value.

5.7.14. `Void`

Super type: `Object`

5.7.15. `xtend::AdviceContext`

Super type: `Object`

Table 5.18. Properties

Type	Name	Description
`List`	`paramNames`	
`String`	`name`	
`List`	`paramTypes`	
`List`	`paramValues`	

Table 5.19. Operations

Return type	Name	Description
`Object`	`proceed` ()	
`Object`	`proceed` (`List`)	

5.7.16. `xpand2::Definition`

Super type: `Object`

Table 5.20. Properties

Type	Name	Description
`List`	`paramNames`	
`List`	`paramTypes`	
`oaw::Type`	`targetType`	
`String`	`name`	

Table 5.21. Operations

Return type	Name	Description
`String`	`toString` ()	
`Void`	`proceed` (`Object`, `List`)	
`Void`	`proceed` ()	

5.7.17. `xpand2::Iterator`

Super type: `Object`

Table 5.22. Properties

Type	Name	Description
`Boolean`	`firstIteration`	
`Boolean`	`lastIteration`	
`Integer`	`counter1`	
`Integer`	`elements`	
`Integer`	`counter0`	

Chapter 6. *Xtext* Reference (from oAW 4.2)

6.1. Introduction

Does this sound familiar to you?

Oh, I need to rename this misspelled property within our domain model. Ok, so let's startup this big UML tool... and by the way let's get a new cup of coffee. Cool, it has been started up already. Grabbing the mouse, clicking through the different diagrams and graph visualizations... Ahhh, there is the name of the property right down there in the properties view. Let's change it, export it to XMI (...drinking coffee), starting the oAW generator (in a jiffy ;-)). Oh, it's not allowed for the property to be named that way, a constraint says that properties names should start with a lower case letter. Ok, let's change that and again reexport...

Some moments later, everything seems to works (tests are green). Ok let's check it in!

Oh someone else has also modified the model! Aaarrrgggh....

Think of this:

Want to change a properties name? Ok, open the respective text file, rename the properties name and save it. The editor complains about a violated constraint. Ok fix the issue, save again and generate. Check the changes into SVN (CVS). Oh, there is a conflict, ok, let's simply merge it using Diff.

And now? Let's have a cup of coffee :-)

Xtext is a textual DSL development framework. Providing the ability to describe your DSL using a simple EBNF notation. Xtext will create a parser, a metamodel and a specific Eclipse text editor for you!

6.2. Installation

Xtext is part of the openArchitectureWare SDK. An easy way to install is to download and unzip the "*Eclipse 3.3 for RCP/Plug-in Developers*" from the download site of Eclipse (http://www.eclipse.org/downloads/).

Afterwards, download the `org.openarchitectureware.all_in_one_feature-4.3.0.*.zip` release from our website and extract it to the directory where you have unzipped the Eclipse release (i.e. the Eclipse installation directory).

Make sure that you start Eclipse with a Java VM Version greater than 5.0.

6.3. Migrating from Xtext 4.2 to Xtext 4.3

If you have been using Xtext prior to version 4.3, you need to re-generate your DSL editors in order to use the updated version 4.3. To do this, please perform the followng steps:

- In your DSL project, open `generate.properties` and enable overwriting of plug-in resource by setting `overwrite.pluginresources` to `true`
- Open your grammar definition. From the context menu, select **Generate Xtext Artifacts** in order to re-generate the DSL and the DSL editor.

- Finally, open the context menu of your DSL project and select **PDE Tools > Update classpath...**. Tick all your DSL projects and click **Finish**.

6.4. Getting started

If you didn't already do so, you should have a look at the ???. This will give you a good overview of how *Xtext* basically works. Come back to this document to find out about additional details.

6.5. The Grammar Language

At the heart of *Xtext* lies its grammar language. It is a lot like an extended Backus-Naur-Form, but it not only describes the concrete syntax, but can also be used to describe the abstract syntax (metamodel).

A grammar file consists of a list of so called *Rules*.

6.5.1. Example

This is an example of a rule, describing something called an *entity*:

```
Entity :
 "entity" name=ID "{"
  (features+=Feature)+
 "}";
```

Entity is both the name of the rule and the name of the metatype corresponding to this rule. After the colon, the description of the rule follows. A description is made up of *tokens*. The first token is a *keyword* token which says that a description of an entity starts with the keyword `entity`. A so-called *Assignment* follows (`name=ID`).

The left-hand side refers to a property of the metatype (in this case it is the property `name` of type Entity). The right-hand side is a call to the built-in token `ID`. Which means Identifier and allows character sequences of the form (`'a-zA-Z_' ('a-zA-Z_0-9)*`). The parser will assign ('=') the Identifier to the specified property (name).

Then (enclosed in curly brackets, both are essentially keyword tokens) one or more features can be declared `(features+=Feature)+`. This one, again, is an assignment. This time, the token points to another rule (called Feature) and each feature is added (note `+=` operator) to the reference of the entity called `features`.

The Feature rule itself could be described like this:

```
Feature :
 type=ID name=ID ";";
```

so, that the following description of an entity would be valid according to the grammar:

```
entity Customer {
 String name;
 String street;
 Integer age;
 Boolean isPremiumCustomer;
}
```

Note that the types (String, Integer, Boolean) used in this description of a customer, are simple identifiers, they do not have been mapped to e.g. Java types or something else. So, according to the grammar, this would also be valid, so far:

```
entity X {
 X X;
 X X;
 X X;
 cjbdlfjerifuerfijerf dkjdhferifheirhf;
}
```

6.5.2. How the parsers work in general

As stated before, the grammar is not only used as input for the parser generator, but it is also used to compute a metamodel for your DSL. We will first talk about how an *Xtext* parser works in general, before we look at how a metamodel is being constructed.

The analysis of text is divided in two separate tasks: the lexing and the parsing.

The lexer is responsible of creating a sequence of tokens from a character stream. Such tokens are identifiers, keywords, whitespace, comments, operators, etc. *Xtext* comes with a set of built-in lexer rules which can be extended or overwritten if necessary. You have already seen some of them (e.g. `ID`).

The parser gets the stream of tokens and creates a parse tree out of them. The type rules from the example are essentially parser rules.

Now, let us have a look at how the metamodel is constructed.

6.5.3. Type Rules

We have already seen, how type rules work in general. The name of the rule is used as name for the metatype generated by *Xtext*.

6.5.3.1. Assignment tokens / Properties

Each assignment token within an *Xtext* grammar is not only used to create a corresponding assignment action in the parser but also to compute the properties of the current metatype.

Properties can refer to the simple types such as String, Boolean or Integer as well as to other complex metatypes (i.e. other rules). It depends on the assignment operator and the type of the token on the right, what the type actually is.

There are three different assignment operators:

- Standard assignment '=' : The type will be computed from the token on the right.
- Boolean assignment '?=' : The type will be Boolean.
- Add assignment '+=' : The type will be List. The inner type of the list depends on the type returned by the token on the right.

Example:

```
Entity :
 (isAbstract?="abstract")? "entity" name=ID "{"
   (features+=Feature)*
 "}";
```

The metatype Entity will have three properties:

1. `Boolean isAbstract`
2. `String name`
3. `List[Feature] features`

6.5.3.2. Cross References

Parsers construct parse trees not graphs. This means that the outcome of a parser has no crossreferences only so called containment references (composition).

In order to implement crosslinks in the model, one usually has to add third task: the linking. However, *Xtext* supports specifying the linking information in the grammar, so that the metamodel contains cross references and the generated linker links the model elements automatically (for most cases). Linking semantic can be arbitrary complex. *Xtext* generates a default semantic (find by id) which can be selectively overwritten. We will see, how this can be done, later in this document.

Let us now see in detail, what the grammar language supports:

```
Entity :
 "entity" name=ID ("extends" superType=[Entity])?
 "{"
   (features+=Feature)*
 "}";
```

Have a look at the optional `extends` clause. The rule name Entity on the right is surrounded by squared parenthesis. That's it.

By default, the parser expects an `ID` to point to the referred element. If you want another kind of token to act as a reference, you can optionally specify the type of token you want to use, separated by a vertical bar:

```
 ... ("extends" superType=[Entity|MyComplexTokenRule])? ...
```

Where MyComplexTokenRule must be either a NativeLexerRule or a StringRule (explanation follows).

6.5.3.3. Metatype Inheritance

We have seen how to define simple concrete metatypes and its features. One can also define type hierarchies using the grammar language of *Xtext*. Sometimes you want to abstract rules, in order to let a feature contain elements of different types.

We have seen the Feature rule in the example. If you would like to have two different kinds of Feature (e.g. Attribute and Reference), you could create an abstract type rule like this:

```
Feature :
 Attribute | Reference;

Attribute :
 type=ID name=ID ";";

Reference :
 "ref" (containment?="+")? type=ID name=ID ("<->" oppositeName=ID)? ";";
```

The transformation creating the metamodel automatically normalizes the type hierarchy. This means that properties defined in all subtypes will automatically be moved to the common supertype. In this case, the abstract type Feature would be created containing the two features (`name` and `type`). Attribute and Reference would be subtypes of Feature, inheriting those properties.

It is also possible to define concrete supertypes like this:

```
Feature :
 type=ID name=ID ";" | Reference;

Reference :
 "ref" (containment?="+")? type=ID name=ID ("<->" oppositeName=ID)? ";";
```

In this case Feature would not be abstract, but it would be the supertype of Reference.

If you need multiple inheritance you can simply add an abstract rule. Such a rule must not be called from another rule.

Example:

```
Model : TypeA TypeA TypeC;

TypeA : "A" | TypeB;
TypeB : "B";
TypeC : "C";

CommonSuper : TypeB | TypeC; // just for the type hierarchy
```

The resulting type hierarchy will look like this:

- Model

- TypeA

- TypeB extends TypeA, CommonSuper

- TypeC extends CommonSuper

- CommonSuper

6.5.4. Enum Rule

The enum rule is used to define enumerations. For example, if you would like to hardwire the possible data types for attributes into the language, you could just write:

```
Attribute :
 type=DataType name=ID ";";

Enum DataType :
 String="string" | Integer="int" | Boolean="bool";
```

In this case, this would be valid:

```
entity Customer {
 string name;
 string street;
 int age;
 bool isPremiumCustomer;
}
```

However, this would not be valid:

```
entity Customer {
 X name; // type X is not known
 String street; // type String is not known (case sensitivity!)
}
```

6.5.5. String Rule

Xtext provides built-in tokens. We have already seen the IdentifierToken and the KeywordToken.

Sometimes, this is not sufficient, so we might want to create our own tokens. Therefore, we have the so-called `String` rule , which is implemented as a parser rule (it is not a lexer rule).

Example

```
String JavaIdentifier :
 ID ("." ID)*;
```

The contents of the `String` rule is simply concatenated and returned as a string. One can refer to a `String` rule in the same manner we refer to any other rule.

So, just in case you want to declare data types using your DSL and therein specify how it is mapped to Java (not platform independent, of course, but expressive and pragmatic), you could do so using the following rules:

```
Attribute :
 type=DataType name=ID ";";

DataType :
 "datatype" name=ID "mappedto" javaType=JavaIdentifier;

String JavaIdentifier :
 ID ("." ID)*;
```

A respective model could look like this:

```
entity Customer {
 string name;
 string street;
 int age;
 bool isPremiumCustomer;
}

datatype string mappedto java.util.String
datatype int mappedto int
datatype bool mappedto boolean
```

You could of course point to a custom type mapping implementation, if you need to support multiple platforms (like e.g. SQL, WSDL, Java,...). Additionally, you should consider to define the data types in a separate file, so the users of your DSL can import and use them.

6.6. Lexer Rules

As mentioned before we *Xtext* provides some common built-in lexer rules. Let us start with the two simplest ones.

6.6.1. Keyword Tokens

All static characters or words, known as *keywords*, can be specified directly in the grammar using the usual string literal syntax. We never need the value of keyword because we already know it since it is static. But sometimes, there are optional keywords like e.g. the modifiers in Java. The existence of a keyword can be assigned using the boolean assignment operator "`?=`". However, if you want to assign the value of the keyword to a property just use the assignment operator '`=`'.

Example:

```
Entity :
 (abstract?="abstract")? "entity" name=ID ("<" extends=ID)?
 "{"
  (features+=Feature)*
 "}";
```

With this the type Entity will have the boolean property `abstract`, which is set to `true` if the respective keyword has been specified for an entity. (The `extends` part is added, because an abstract entity would not make sense without inheritance).

Note that operators such as '`<`' in the example are keywords, too.

6.6.2. The `ID` Token

We also have seen the identifier token (`ID`). This is the token rule expressed in AntLR grammar syntax:

```
('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
```

So, an identifier is a word starting with a character or an underscore, optionally followed by further characters, underscores, or digits. The return value of the `ID` token is a `String`. So, if you use the usual assignment operator "`=`", the feature the value is assigned to will be of type `String`. You could also use the boolean operator (`?=`) and the type will be `Boolean`.

If an identifier conflicts with a keyword or another lexer rule, it can be escaped with the " `^` " character.

6.6.3. The `String` Token

There is also a built-in String Token. Here is an example:

```
Attribute :
  type=DataType name=ID (description=STRING)? ";";
```

With this token, one can optionally specify a description for an entity like this:

```
entity Customer {
 string name ;
 string street "should include the street number, too.";
 int age;
 bool isPremiumCustomer;
}
```

By default, the two string literal syntaxes "my text" and 'my text' are supported. Note that, unlike in Java, also multi-line strings are supported:

```
entity Customer {
 string name ;
 string street "should include the street number, too.
        And if you do not want to specify it, you
        should consider specifying it somewhere else.";
 int age;
 bool isPremiumCustomer;
}
```

6.6.4. The `INT` Token

Sometimes, you want to assign Integers. Xtext supports it with the built-in lexer rule INT.

```
Index:
 "#" index=INT;
```

The default pattern is:

```
('-')?('0'..'9')+
```

It can be overwritten (see next section), but you have to take care, that the coercion (*Integer.valueOf(String)* is used) works.

6.6.5. The `URI` Token

If you want to allow inclusion of and references to other models, use the `URI` token, e.g.

```
Import:
     'import' model=URI ';';
```

From the parser's point of view, an `URI` token is just a String literal enclosed in quotes. The Runtime interprets this string as the Uniform Resource Identifier (URI) of another model file whose elements can be referenced from the importing model. With the above grammar rule example, you can allow references to elements of the model file `refModel.model` by adding a line

```
import "platform:/resource/myproject/model/refModel.model";
```

in your model. See Section 6.9.1, “Cross-References to Models of the Same DSL/Metamodel” and Section 6.10.2, “Cross-references to Models of a Different DSL/Metamodel” for examples on model import.

6.6.6. Comments

There are two different kinds of comments automatically available in any *Xtext* language.

```
 // single-line comments and

 /*
  mutli-line comments
 */
```

Note that those comments are ignored by the language parser by default (i.e. they are not contained in the AST returned from the parser).

If you do not want ignored comments, or you want to have a different syntax, you need to overwrite the default implementation (The name of the corresponding comment rule is `SL_COMMENT` for single-line comments, i.e. `ML_COMMENT` for multi-line comments).

6.6.7. Whitespace

Every textual model contains whitespace. As most languages simply ignore whitespace, *Xtext* does so by default, too. If you want to have semantic whitespace in your language (like e.g. Python), you have to overwrite the built-in whitespace rule (its name is `WS`).

6.6.8. Native rules / Overwriting built-in lexer rules

If you want to overwrite one or more of the built-in lexer rules or add an additional one, the so-called *native rule* is your friend.

Example:

```
// overwriting SL_COMMENTS we do not want Java syntax (//) but bash syntax (#)
Native SL_COMMENT :
 "'#' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}";

// fully qualified names as a lexer rule
Native FQN :
 "ID ('.' ID)*";
```

The syntax is :

```
"Native" ID ":"
  STRING // The string contains a valid ANTLR 3 lexer rule expression
  ";"    // (see http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation)
```

6.7. The Generator

It is assumed that you have used the *Xtext* project wizard, and that you have successfully written an *Xtext* grammar file describing your little language. Next up you need to start the generator of Xtext in order to get a parser, a metamodel and an editor. To do so. just right-click the workflow file (`*.oaw`) located next to the grammar file and choose Run As+oAW workflow (in Eclipse, of course). The generator will read the grammar file in and create a bunch of files. Some of them located in the `src-gen` directories others located in the `src` directory.

> **Important**
>
> IMPORTANT : You should now (after first generation) open the *.properties file and set the "overwritePluginRes=true" option to false!

6.7.1. Configuring the Generator

The generator can be configured with the following properties defined in `generate.properties`:

property name (default)	description
grammar	The grammar file to generate the code from.

property name (default)	description
debug.grammar (false)	Specifies whether a debug grammar should be generated. A debug grammar is an AntLR grammar without any action code, so it can be interpreted within AntLRWorks.
language.name	The name of the DSL. Is used throughout the generated code
language.nsURI ("http://www.oaw.org/xtext/dsl/${language.name}")	A unique URI used within the derived ecore package.
language.fileextension ("${language.name}")	The file extension the generated editor is configured for.
overwrite.pluginresources ("false")	If this is set to true the plugin resources (META-INF/MANIFEST.MF, plugin.xml) will be overwritten by the generator!!!
wipeout.src-gen ("true")	Specifies whether the src-gen folders should be deleted before generation.
generator.project.name ("")	If this property is set to something, a project wizard will be generated referencing the generator project.
workspace.dir	The root of the workspace.
core.project.name	name of the main project
core.project.src ("src/")	src folder of the main project
core.project.src.gen ("src-gen/")	src-gen folder of the main project
editor.generate ("true")	should an editor be generated at all
editor.project.name ("${core.project.name}.editor")	name of the editor project
editor.project.src ("${core.project.src}")	src folder of the editor project
editor.project.src.gen ("${core.project.src.gen}")	src-gen folder of the editor project

6.7.2. Generated and manual code

Any textual artifacts located in the `src` directory (of any project) will always stay untouched. The generator just creates them the first time when they do not exist.

Files generated to the `src-gen` directory should never be touched! The whole directory will be wiped out the next time one starts the generator.

6.7.3. The different projects and artifacts

Xtext generates artifact into two different projects.

6.7.3.1. The main project

The name of the main project can be specified in the wizard. This project contains the main language artifacts and is 100% eclipse independent. The default locations of the most important resources are:

Table 6.1. Resources of the main project

Location	Description
src/[dslname].xtxt	The grammar file, containing the grammar rules describing your DSL
src/generate.oaw	The workflow file for the Xtext generator.
src/generator.properties	Properties passed to the Xtext generator
src/[base.package.name]/Checks.chk	The Check-file used by the parser and within the editor. Used to add semantic constraints to your language.
src-gen/[base.package.name]/ GenChecks.chk	The generated Check-file contains checks automatically derived from the grammar.
src/[base.package.name]/ Extensions.ext	The Extension-file is used (imported) by all another extensions and check files. It reexports the extensions from `GenExtensions.ext` (contained in src-gen/). You can specify (use more concrete parameter types) or overwrite (use the same signature) the extensions from `GenExtensions.ext` here.
src-gen/[base.package.name]/ GenExtensions.ext	generated extensions (reexported by `Extensions.ext`).
src/[base.package.name]/Linking.ext	Used by `Linker.ext` from *`src-gen/[base.package.name]/parser/Linker.ext`* Default linking semantic is generated in `GenLinking.ext` (src.gen/) and can be overwritten here.

src-gen/[base.package.name]/ GenLinking.ext	Contains the default linking semantic for each cross reference.
	
	Example:
	
	`Void link_featureName(my::MetaType this) :`
	`  (let ents = this.allVisibleElements().typeSelect(my::ReferredType) :`
	`  this.setFeatureName(ents.select(e|e.id()`
	`  == this.parsed_featureName).first()) );`
	
	This code means the following:
	
	Get all instances of the referred type using the allVisibleElements() *extension. Select the first one where the* id() *equals the parsed value (by default an identifier).*
	
	Both extensions, *id()* and *allVisibleElements()*) can be overwritten or specialized in the `Extensions.ext` file. The *link_[featurename]()* extension can be overwritten in `Linking.ext`

src-gen/[base.package.name]/ [dslname].ecore	Metamodel derived from the grammar
src-gen/[base.package.name]/parser/ *	Generated AntLR parser artifacts

6.7.3.2. The editor project

The name of the editor project is derived from the name of the main project by appending the suffix `.editor` to it. The editor project contains the informations specific to the Eclipse text editor. Note that it uses a generic xtext.editor plugin, which does most of the job. These are the most important resources:

Table 6.2. Resources of the editor

Location	Description
src/[base.package.name]/ [dslname]EditorExtensions.ext	The Xtend-file is used by the outline view. If you want to customize the labels of the outline view, you can do that here.
src-gen/[base.package.name]/ [dslname]Utilities.java	Contains all the important DSL-specific information. You should subclass it in order to overwrite the default behaviours.
src/[base.package.name]/ [dslname]EditorPlugin.java	If you have subclassed the *Utilities class, make sure to change the respective instantiation here.

6.7.3.3. The generator project

The name of the generator project is derived from the name of the main project by appending the suffix `.generator` to it. The generator project is intended to contain all needed generator resources such as *Xpand* templates, platform-specific *Xtend* files etc..

These are the most important resources:

Table 6.3. Resources of the generator

Location	Description
src/[base.package.name]/ generator.oaw	The generators workflow preconfigured with the generated DSL parser and the Xpand component. As this is just a proposal, feel free to change/add the workflow as you see fit.
src-gen/[base.package.name]/ Main.xpt	The proposed *Xpand* template file.

6.8. Pimping the editor

The generated editor supports a number of features known from other eclipse editors. Although most of them have a decent default implementation, we will see how to tweak and enhance each of them.

6.8.1. Code Completion

Code Completion is controlled using oAW extensions. The default implementation provides keyword proposals as well as proposals for cross references.

Have a look at the extension file `ContentAssist.ext`. A comment in this file describes how to customize the default behaviour:

```
/*
 * There are two types of extensions one can define
 *
 * 1) completeMetaType_feature(ModelElement ele, String prefix)
 * This one is called for assignments only. It gets the underlying Modelelement and the current
 * prefix passed in.
 *
 * 2) completeMetaType(xtext::Element grammarEle, ModelElement ele, String prefix)
 * This one gets the grammarElement which should be completed passed in as the first parameter.
 * an xtext::Element can be of the following types :
 *  - xtext::RuleName (a call to a lexer rule (e.g. ID)),
 *  - xtext::Keyword,
 *  - xtext::Assignment
 *
 * Overwrite rules are as follows:
 * 1) if the first one returns null for a given xtext::Assignment or does not exist the second one
 *    is called.
 * 2) if the second one returns null for a given xtext::Keyword or does not exist a default keyword
 *    proposal will be added.
 *
 * Note that only propals with wich match (case-in-sensitive) the current prefix will be proposed
 * in the editor
 */
```

6.8.2. Navigation

The implementation for the navigation actions is implemented via extensions, too. As for Code completion the same pattern applies: There is a `GenNavigation.ext` extension file in the `src-gen` folder which can be overwritten or specialized using the `Navigation.ext` file in the `src` folder (reexporting the generated extensions).

There are two different actions supported by *Xtext*:

6.8.2.1. Find References

This action can be invoked via **Ctrl-Shift-G** or via the corresponding action context menu. The default implementation returns the cross references for a model element.

The signature to overwrite / specialize is:

```
List[UIContentNode] findReferences(String s, Object grammarelement, Object element) : ...;
```

A `UIContentNode` is a metaclass used by *Xtext*. An UIContentNode represents an element visualized in Eclipse.

Here is the declaration of `UIContentNode` (pseudo code):

```
package tree;

eclass UIContentNode {
  UIContentNode parent;
  UIContentNode[] children;
  String label;
  String image;
  emf::EObject context;
}
```

A content node can have children and / or a parent (the tree structure is not used for find references). The label is used for the label in Eclipse, and the image points to an image relative to the `icons` folder in the editor project. The icon instances are automatically cached and managed.

The context points to the actual model element. This is used to get the file, line and offset of the declaration. If you do not fill it, you cannot click on the item, in order to get to it.

6.8.2.2. Go to Declaration

This action can be invoked via **F3** as well as by holding **CTRL**, hovering over an identifier and left click the mouse.

The default implementation goes to the declaration of a cross reference. You can implement or overwrite this action for all grammar elements.

```
emf::EObject findDeclaration(String identifier, emf::EObject grammarElement,
  emf::EObject modelElement) : ...;
```

Have a look at the generated extensions, to see how it works.

6.8.3. Outline View

The outline view is constructed using a tree of `UIContentNode` objects (see above).

Each time the outline view is created, the following extension is called:

```
UIContentNode outlineTree(emf::EObject model)
```

It is expected to be declared in `Outline.ext`, which by default exports a generic implementation from `GenOutline.ext` (the same pattern again).

You can either reuse the generic extension and just provide a *label()* and *image()* extension for your model elements (should be added in `EditorExtensions.ext`).

However, if you want to control the structure of the outline tree you can overwrite the extension *outlineTree(emf::EObject model)* in `Outline.ext`.

You can define multiple outlines for a single DSL, each representing a different viewpoint on your model. Each viewpoint needs has a unique name. Override the `getViewpoints()` extension to return a list of all viewpoint names. You can customize each viewpoint using the same extensions as above suffixed with the viewpoint's name (spaces are replaced by underscores. Example:

```
List[String] viewpoints() : { "Entities Only" };

UIContentNode outlineTree_Entities_Only(emf::EObject model) :
 let x = model.createContentNode() :
  x.children.addAll(model.eContents.outlineTree_Entities_Only())
  // return the node, not its children
  -> x;

UIContentNode outlineTree_Entities_Only(Model model) :
 let x = model.createContentNode() :
  // add child nodes for elements of type Entity only
  x.children.addAll(model.types.typeSelect(Entity).outlineTree_Entities())
  -> x;

create UIContentNode createContentNode(emf::EObject model) :
 setLabel(model.label()) ->
 setImage(model.image()) ->
 setContext(model);
```

You can switch the viewpoints in the running editor by means of the drop-down menu behind the small triangle symbol in the action bar of the outline view.

6.8.4. Syntax Highlighting

The default syntax highlighting distinguishes between comments, string literals, keywords, and the rest.

You can adjust the specific display styles for keywords with regard to the context they appear in by overriding the `fontstyle(String keyword, EObject metaClassName)` extension in the `Style.ext` extension file. The predefined method `createFontstyle` will help you creating the `FontStyle` object, e.g.

```
FontStyle fontstyle(String keyword, Node node) :
 node.eContainer.metaType==Node
  ? createFontStyle(true, false, false, false,
   createColor(200,200,200), createColor(255,255,255))
  : createFontStyle(true, false, false, false,
   createColor(255,0,0), createColor(255,255,255))
;
```

If you just want to specify which words to be coloured as keywords you can extend the `[basepackage.][Languagename]Utilities.java` class from the editor plugin. You need to overwrite the following method.

```
public String[] allKeywords()
```

> **Important**
>
> Do not change the `Utilities` class directly, because it will be overwritten the next time your start the generator.

Each `String` returned by the method represents a keyword.

The utilities method is created within the `[LanguageName]EditorPlugin.java`. So, make sure that you change the following lines, as well:

```
// OLD -> private MyLangUtilities utilities = new MyLangUtilities();
private MyCustomUtilities utilities = new MyCustomUtilities();
public LanguageUtilities getUtilities() {
  return utilities;
}
```

6.9. Cookbook

This part of the documentation deals with the discussion and solution of different requirements and problems.

6.9.1. Cross-References to Models of the Same DSL/Metamodel

Since version 4.3, Xtext provides first class support for cross-references to elements in other model files. This allows to spread a single model across several files or to reuse parts of a model by referencing it. The former mechanism using the built-in model registry does not work any longer.

The following example illustrates how to implement cross-resource references to models of the same DSL. Referencing elements of a foreign DSLs is shown in Section 6.10.2, “Cross-references to Models of a Different

DSL/Metamodel". First of all, you have to enable model import in your DSLs grammar. This is achieved by declaring a rule that matches the `URI` token.

```
Model:
 (imports+=Import)*
 (elements+=Element)*;

// declare an import statement by means of a rule matching the URI token
Import:
 'import' model=URI;

Element:
 'element' name=ID '{'
  (ref+=Reference)*
 '}';

Reference:
 'ref' element=[Element];
```

This way, we can use the `import` keyword followed by a URI in our models to enable referencing. Note that `import` is just an arbitrary keyword and all the magic comes with the `URI` Token. It loads all elements from the given model and makes them available in the `allVisibleElements()` extension. And how does it look like on the model level? Consider the following model file:

```
// refModel.dsl
Element externalElement {
}
```

To reference the element `externalElement` in another file you'll just have to import the `refModel.dsl` and then reference

```
// model.dsl
// import the model to be referenced by matching the URI token
import "platform:/resource/myplugin/model/refModel.dsl"

Element internalElement {
}

Element myElement {
 ref internalElement  // reference by ID
 ref externalElement  // reference by ID, as if it was in the same file
}
```

6.10. Experimental Features

The features described in this section are mainly intended for advanced users. They are likely to change in the future.

6.10.1. Instantiate Existing Metamodels

In some cases you may want to use Xtext to create a new concrete textual syntax for an existing Ecore model. You'll have to provide a way to make Xtext's rules instantiate existing EClasses rather than the generated ones.

The Ecore model to be referenced must reside in a plug-in that is deployed in your eclipse. This plug-in must register the Ecore model to the `org.eclipse.emf.ecore.generated_package` or `org.eclipse.emf.ecore.dynamic_package` extension. Additionally, load the imported metamodel to the generation workflow using the `StandaloneSetup` component:

```
<workflow>
   <property file='generate.properties'/>
   <bean class="org.eclipse.mwe.emf.StandaloneSetup">
      <registerGeneratedEPackage value="my.MyPackage"/>
   </bean>
   <component file='org/openarchitectureware/xtext/Generator.oaw' inheritAll='true'/>
</workflow>
```

The next step is to make the existing Ecore model available to your Xtext plug-in. Use the `importModel` statement followed by the nsURI of the model for that purpose, e.g.

```
importMetamodel "http://www.oaw.org/entity" as refModel;
```

imports `entity` and makes its elements available in the namespace alias `refModel`.

Now you make a parser rule instantiate an EClass from the imported model by adding the qualified class name in brackets after the rule name. Assuming you want to instantiate an EClass named `Entity` which has an attribute `name`, simply write

```
MyRule [refModel::Entity] :
 name=ID;
```

If all rules in your grammar instantiate foreign EClasses, the automatic generation of the Ecore model becomes obsolete. Switch it off using the `preventMMGeneration` directive at the beginning of your grammar file:

```
preventMMGeneration
importMetamodel "http://www.oaw.org/entity" as refModel;
```

6.10.2. Cross-references to Models of a Different DSL/Metamodel

These are the steps to take, if you want to implement cross-references to models that comply to a DSL that is different from the referencing model's DSL.

Follow the above instructions to make the referenced metamodel/DSL available in your grammar. The outgoing reference is defined - analogous to the instantiation of foreign model element - using brackets, e.g.

```
Referer :
 (entities+=[refModel::Entity])*;
```

6.10.3. Extension/Partitioning of Grammars

If you want to extend an existing grammar or split a large grammar into multiple pieces, you can use the `importGrammar` statement, e.g.

```
importGrammar "platform:/resouce/my.dsl/src/mydsl.xtxt"
```

Imported rules can be overridden simply by redefinition. Note that for the Xtext artifact generation to succeed, the input grammar combined with its imported grammars must not contain unused rules.

Chapter 7. Generic Editor

7.1. Introduction

The generic editor can be used to edit any EMF models even dynamic instances. The generic editor combines the power and flexibility of EMF's reflective editor with the usability and elegance of Eclipse Forms. The user can also dynamically provide constraints and icon and label providers using oAW.

7.2. How to Use the Generic Editor

The user interface of the generic editor is easy to use. It is based on a master/detail component. The master section shows the model tree, while the individual properties of the selected element can be edited in the details section. In the toolbar there is also a validation button integrated. This button can be used to check the model based on the used validation mechanism (see next paragraph).

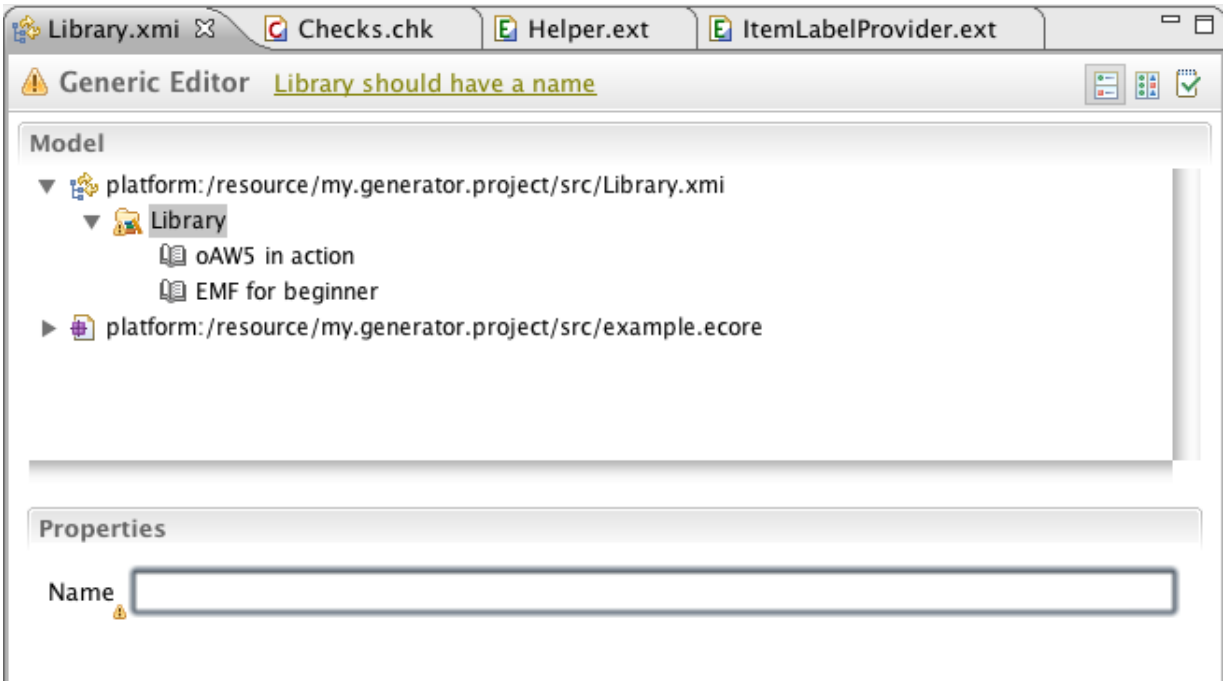


Figure 7.1. Generic Editor

7.3. Working with Dynamic Instances

The editor can of course be used with installed EPackages. In that case the validation is done using the EValidationRegistry and the labels and icons are computed using EMF's *ComposedAdapterFactory.Descriptor.Registry* . As this is exactly how EMF's reflective editor works there is not much added value besides the use of forms instead of the properties view.

Using the generic editor becomes more interesting when it comes to dynamic instances. A dynamic instance is an instance of an Ecore model which has not been installed into the workbench. The use of dynamic instances only makes EMF's code generator obsolete and reduces turnaround times significantly. In order to create a dynamic instance, all you have to do is open (or create) an Ecore file in your workspace, select an EClass and use the context menu to "create a dynamic instance". The action creates a new file and puts an instance of the selected

EClass as it's root element. Using the generic editor one can not only edit theses dynamic models, but also specify constraints. In addition one can define how labels and icons should be computed, all dynamically without the need to install bundles into eclipse or restart the workbench.

7.4. Model Validation

To get your dynamic models validated all you have to do is provide a oAW Check file for the metamodel. The editor will validate the model on startup, on save and each time the check button is pushed. There are some prerequisites about that:

- the project must have the openArchitectureWare nature enabled
- the check file must be named `Checks.chk` , and
- the check file must be placed in `yourEPackageName` package in project's source folder

A typical check example for an entity *Library* could look like this:

```
context Library WARNING "name#Library should have a
name": !this.name.isEmpty();
```

The suffix "name#" in the message says that this warning refer to the feature *name*. With this additional information the generic editor can provide a hyper link and add a marker for the respective feature.

7.5. Customizing Icons and Labels

To customize the presentation of the model elements in the editor you can provide *label()* and *icon()* extensions. Those extensions are found if:

- the project has the openArchitectureWare -Nature enabled
- the extension file is named `ItemLabelProvider.ext` , and
- the extension file must be placed in `yourEPackageName` package in project's source folder
- the extension names are *label* for a label and *icon* for an icon
- they have one parameter with the according type
- corresponding icons must be placed in a folder `/icons/yourEPackageName` (not necessarily on the classpath)

Example:

```
//book String label(Book ctx):
ctx.title.isEmpty()?"Untitled book":ctx.title;

String icon(Book ctx): "book.gif";
```

This example shows how a custom label and icon can be computed for an EClass called *Book* . The extension *label* sets a book's label to title or to "Untitled book" if it is empty. The extension *icon* sets `icons/yourEPackageName/book.gif` image as a book's icon instead of default one.

7.6. Proposals

When setting a reference or an enumeration literal by default the editor provides all elements which would fit (i.e. they are of the reference's type). Sometimes these lists get very long. In order to filter and sort such lists of proposed elements one also can specify an extension For these extensions to be found :

- the project must have the openArchitectureWare nature enabled
- the extension file must be named `Proposals.ext` , and
- the extension file must be placed in `yourEPackageName` package within project's source folder
- the extension's signature is: List[*ReferencedMetaType*] featureName(*MetaType* ctx, List[*ReferencedMetaType*] list)

In the following example one can only select from 'active' authors when setting the author for a book. All the authors are sorted by name:

```
//reference proposal/filter List[Person] author(Book
ctx,List[Person] l):
l.select(e|e.isActive).sortBy(e|e.name);
```

It is also possible to provide code completion for text attributes. The example below shows how to create a *name* -proposal for Entity *Person* , containing John and the names of all Persons already in the model.

```
//attribute proposals
List[String] name(Person ctx):
      {"John"}.addAll(ctx
           .eRootContainer
           .eAllContents
           .typeSelect(Person)
           .collect(e|e.name))
            .toSet()
            .toList();
```

Chapter 8. UML2 Adapter

8.1. Introduction

The UML2 adapter for oAW is available since version 4.1. It is based upon the UML2 2.0 framework of Eclipse and provides a type mapping from the UML 2.1 metamodel implementation to the oAW type system. Hence one can access the whole UML2 metamodel from *Check*, *Xtend*, and *Xpand*. Additionally, and more important, the adapter dynamically maps stereotypes applied to model elements to oAW types and tagged values to oAW properties. You do not have to implement additional Java classes anymore. Just export your models and applied profiles. That's all!

8.2. Installation

First you need to install the UML2 feature from eclipse.org:

http://download.eclipse.org/tools/uml2/updates/site-interim.xml

The oAW uml2adapter is available from the oAW update site:

http://www.openarchitectureware.org/updatesite/milestone/site.xml

(Go to the home page of the project and find the current location if either of the sites do not work)

Restart your Eclipse workbench when Eclipse asks you to do so.

8.3. Setting up Eclipse

You need to configure your project (or the whole workspace) to use the UML2Adapter.

Right-click on the project and choose '*properties*' from the pop-up menu. Therein open the '*openArchitectureWare*' tab, activate the checkboxes (*nature* and *specific metamodel contributors*) and add the *UMLProfiles* metamodel contributor.

8.3.1. Profiles in Eclipse

If you want Eclipse to register (be aware of) your specific profile in order to provide static type checking and code completion in the editors, the profiles (*.profile.uml or *.profile.uml2) need to be on the classpath of the project (e.g. are contained in a `src` folder)

8.4. Runtime Configuration

At runtime, you just need the `org.openarchitectureware.uml2.adapter-4.1.0.jar` (or later). You can use the dependency mechanism of Eclipse from the PDE, or copy or reference the respective JAR file directly. It does not matter, you just have to take care that it is on the classpath.

8.4.1. Workflow

If you have written some *Check*, *Xtend* or *Xpand* files and now want to execute them, you have to provide the right configuration.

You need to configure the UML2 metamodel and a profile metamodel *for each profile* you used directly. A typical configuration looks like this:

```
<workflow>
  <bean class="oaw.uml2.Setup" standardUML2Setup="true"/>
  <component class="oaw.emf.XmiReader">
   ...
  </component>
  <component class="oaw.xpand2.Generator">
   <metaModel class="oaw.uml2.UML2MetaModel"/>
   <metaModel class="oaw.uml2.profile.ProfileMetaModel">
     <profile value="myProfile.profile.uml2"/>
   </metaModel>
   ...
  </component>
<workflow>
```

*Note the bean configuration in the second line. It statically configures the XmiReader to use the right factories for *.uml and *.uml2 files. This is very important.*

If you are invoking several oAW components, you should use the `id` / `idRef` mechanism:

```
<workflow>
  <bean class="oaw.uml2.Setup" standardUML2Setup="true"/>
  <component class="oaw.emf.XmiReader">
   ...
  </component>
  <component class="oaw.xpand2.Generator">
   <metaModel id="uml" class="oaw.uml2.UML2MetaModel"/>
   <metaModel id="profile"
         class="oaw.uml2.profile.ProfileMetaModel">
     <profile value="myProfile.profile.uml"/>
   </metaModel>
   ...
  </component>
  <component class="oaw.xpand2.Generator">
   <metaModel idRef="uml"/>
   <metaModel idRef="profile"/>
   ...
  </component>
```

Chapter 9. UML2 Example

9.1. Setting up Eclipse

Before you can use oAW with Eclipse UML2, you first have to install the UML2 plugins into your Eclipse installation.

9.2. Setting up the project

Create a new openArchitectureWare plugin project. You have to add the following dependencies to the manifest file:

- org.openarchitectureware.uml2.adapter

To tell the oAW Eclipse plugins that this project is a UML2 specific one, you need to specify that in the oAW preferences. Open the project properties, select the openArchitectureWare tab and select the UML2 profiles metamodel.

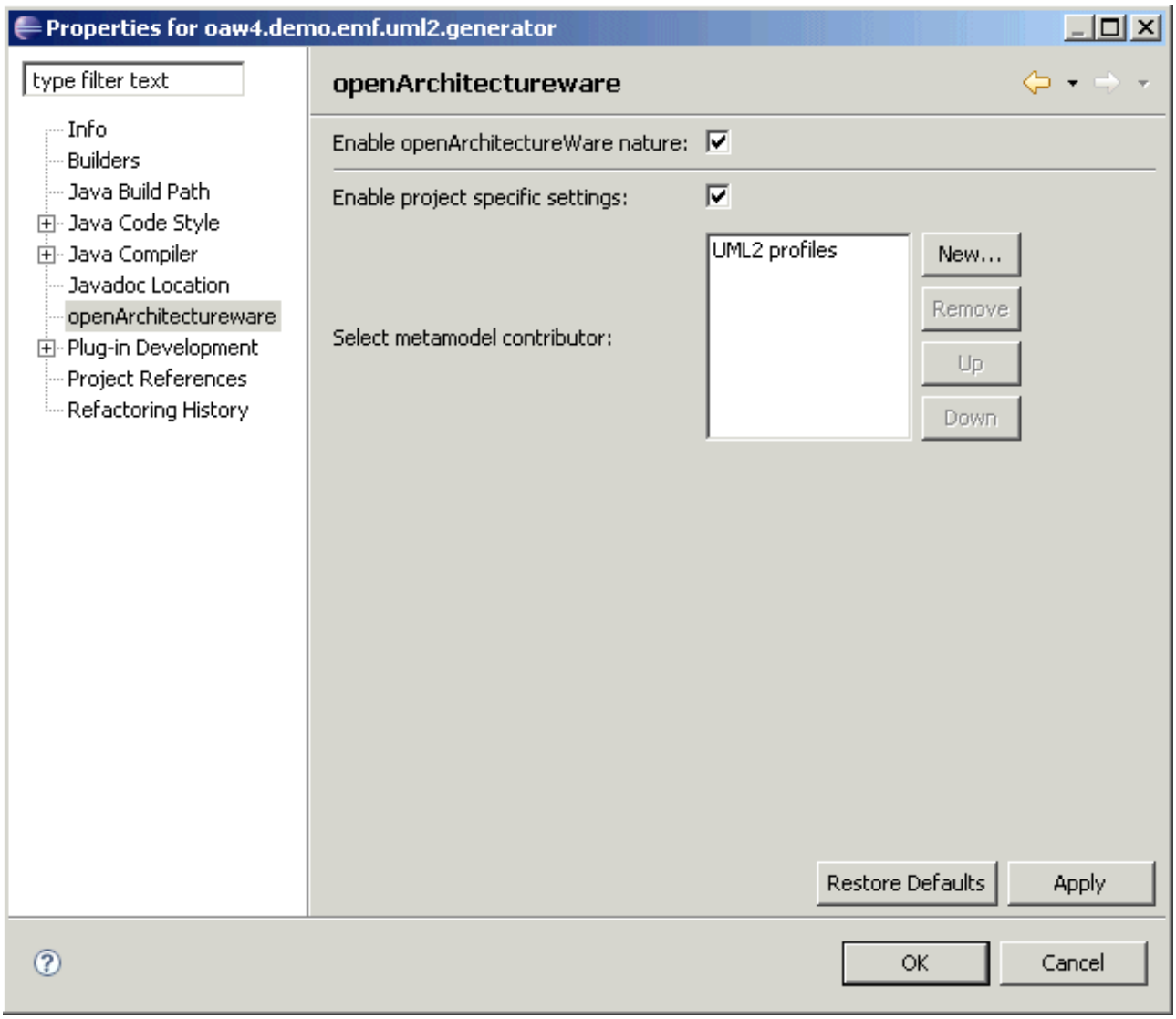


Figure 9.1. Configure UML2 profiles metamodel

Note that if you want to transform an UML2 model into a normal EMF model, you need to add the UML2 metamodel and the EMF metamodels. The order of profiles is important! The UML2 profiles entry must be first in the list.

9.3. Creating a UML2 Model

You start by defining a UML2 model, i.e. an instance of the UML2 metamodel. In the new Java project, in the source folder, you create a UML2 model that you should call `example.uml`.

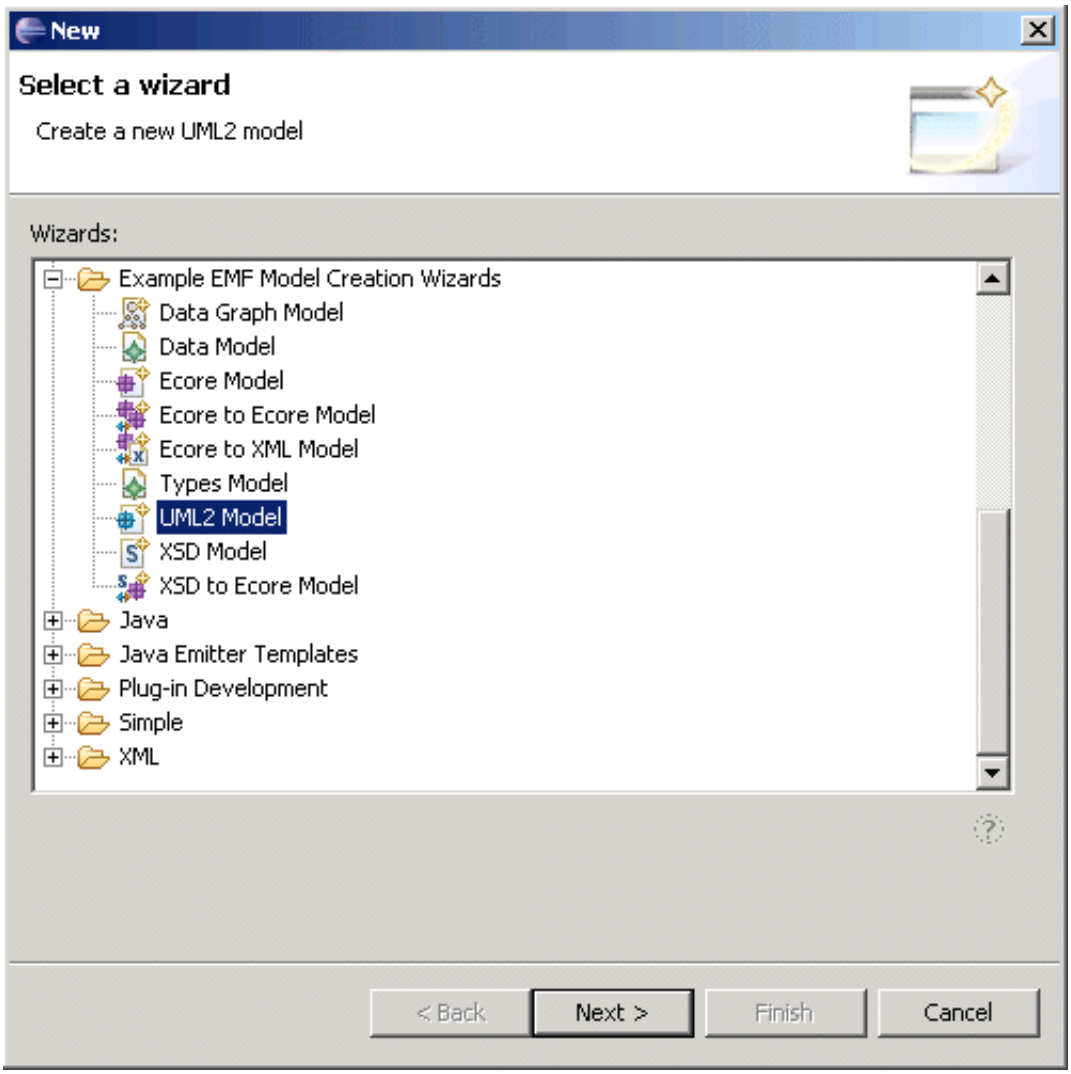


Figure 9.2. Creating a new UML2 model

You then have to select the model object. Make sure its a *Model*, not a *Profile*.

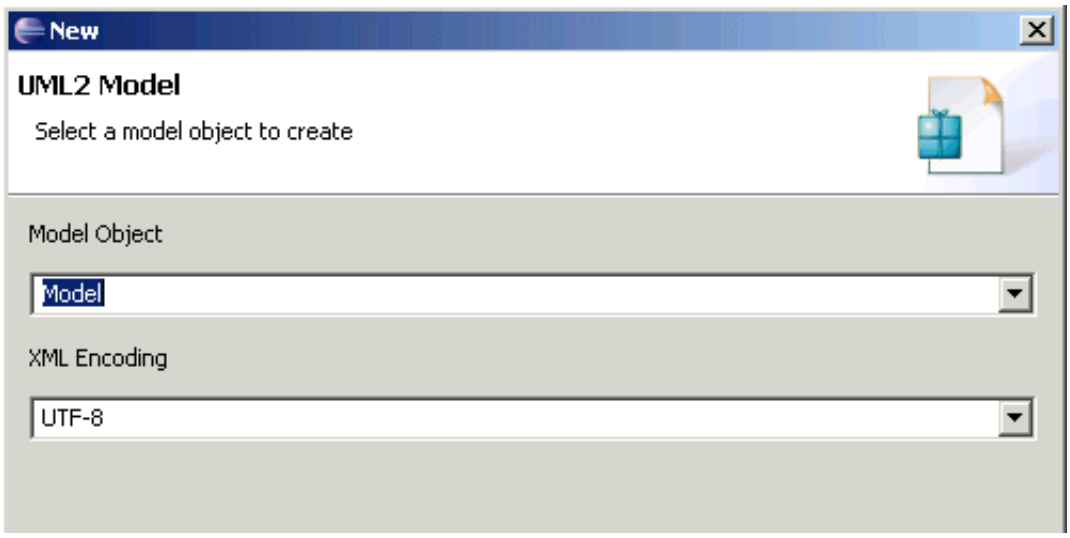


Figure 9.3. Selecting the Model object

9.3.1. Modelling the content

You should then build a model that looks somewhat like this:

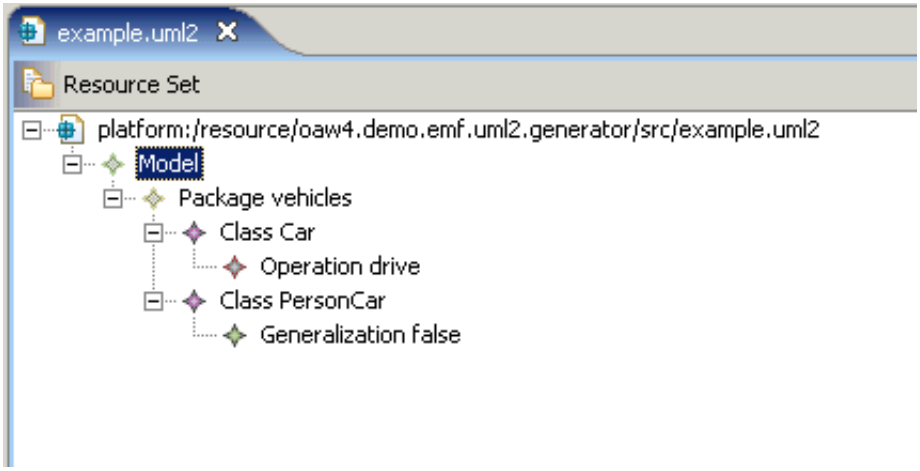


Figure 9.4. Example model

By the way, if you rename the `.uml` file to `.ecore`, you can edit the model using the ecore editors. To inspect the model, they provide a somewhat better view, so you might try!

9.4. Code generation

9.4.1. Defining the templates

Inside the source folder of our project, create a `templates` package. Inside that package folder, create a template file `Root.xpt` that has the following content. First, we define the entry template that is called `Root`. Since we expect a UML model element to be the top element to the model, we define it for `uml::Model`. Note the use of the `uml` Namespace prefix, as defined in the UML2 metamodel. Inside that template, we iterate over all owned elements of the model and expand a template for the packages defined in it.

```
«DEFINE Root FOR uml::Model»
«EXPAND PackageRoot FOREACH ownedElement»
«ENDDEFINE»
```

In the package template, we again iterate over all owned elements and call a template that handles classes. At this point, we expect that only classes are in that package.

```
«DEFINE PackageRoot FOR uml::Package»
 «EXPAND ClassRoot FOREACH ownedType»
«ENDDEFINE»
```

This template handles classes. It opens a file that has the same name as the class, suffixed by `.java`. Into that file, we generate an empty class body.

```
«DEFINE ClassRoot FOR uml::Class»
 «FILE name+".java"»
  public class «name» {}
 «ENDFILE»
«ENDDEFINE»
```

9.4.2. Defining the workflow

In order to generate code, we need a workflow definition. Here is the workflow file; you should put it into the source folder. The file should be generally understandable if you read the oAW EMF Example docs.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<workflow>
```

You need to setup the UML2 stuff (registering URI maps, Factories, etc.). This can be done declaring a bean in before of the `XmiReader` component:

```
 <bean class="oaw.uml2.Setup" standardUML2Setup="true"/>
 <component class="oaw.emf.XmiReader">
  <modelFile value="example.uml"/>
  <outputSlot value="model"/>
 </component>
```

The `XmiReader` reads the model and stores the content (a list containing the model element) in a slot named '`model`'. As usual, you might want to clean the target directory.

```
 <component id="dirCleaner"
  class="oaw.workflow.common.DirectoryCleaner"
  directories="src-gen"/>
```

and in the generator we also configure the EMF metamodel for oAW, together with the UML2 metamodel package, because the UML2 metamodel refers to it.

```
<component id="generator" class="oaw.xpand2.Generator"
 skipOnErrors="true">
 <metaModel class="oaw.type.emf.EmfMetaModel">
  <metaModelPackage  value="org.eclipse.emf.ecore.EcorePackage"/>
 </metaModel>
 <metaModel class="oaw.uml2.UML2MetaModel"/>
 <expand value="templates::Root::Root FOR model"/>
 <outlet path="src-gen/">
  <postprocessor class="oaw.xpand2.output.JavaBeautifier"/>
 </outlet>
</component>
```

If you run the workflow (by right clicking on the .oaw file and select Run As -> oAW workflow) the two Java classes should be generated.

9.5. Profile Support

openArchitectureWare 4 is shipped with a special *UML2Profiles* metamodel implementation. The implementation maps Stereotypes to Types and Tagged Values to simple properties. It also supports Enumerations defined in the profile and Stereotype hierarchies.

9.5.1. Defining a Profile

To define a profile, you can use a variety of UML2-based modelling tools. Assuming they do actually correctly create profile definitions (which is not always the case, as we had to learn painfully), creating a profile and exporting it correctly is straight forward.

In this section, we explain the "manual way", which is good for explaining what happens, but completely useless for practical use. You do not want to build models of realistic sizes using the mechanisms explained below.

You start be creating a new UML2 file (as shown above). In the example we will call it `test.profile.uml`. The root element, however, will be a *Profile*, not a *Package*. Don't forget to actually assign a name to the profile! It should be `test`, too.

As a child of that *Profile*, you then create a *Packaged Element Stereotype* (you will have to scroll a bit in the *Add Child* menu....). For the sake of example, we will call it `test`, too. In our case, we want to make the stereotype be applicable to UML classes – they are defined as part of the UML2 metamodel. So we have to import that metamodel first. So what you do is to select your profile object, and then go to the UML2 Editor menu (in the Eclipse menu bar) and select *Profile -> Reference Metaclass*. Select `uml::Class`. You can then select your stereotype, and select *Stereotype -> Create Extension* from the UML2 Editor menu. Select `uml::Class`. This should lead to the following model. Save it and you are done with the profile definition.

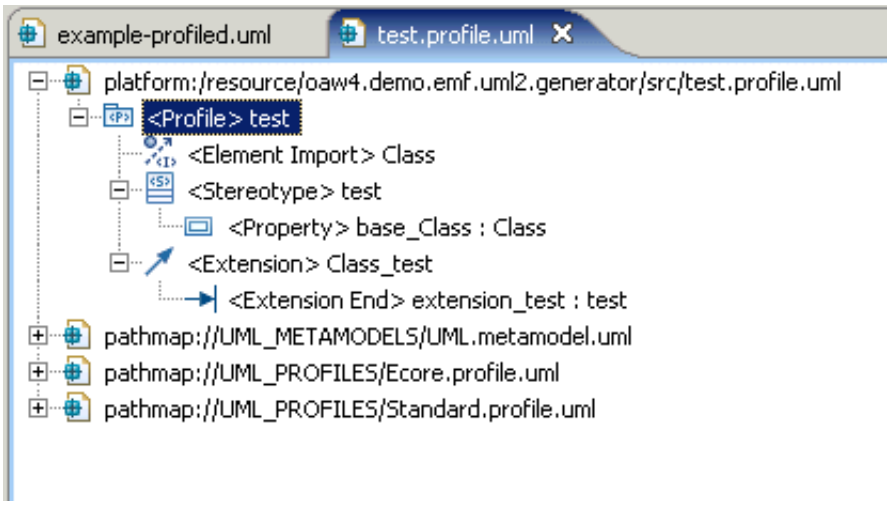


Figure 9.5. Modelling a Profile

9.5.2. Applying the Profile

To make any use of the profile, we have to apply it to some kind of model. To do that, we copy the `example.uml` model to a `example-profiled.uml`. We then open that file and load a resource, namely the profile we just defined. This then looks somewhat like this:

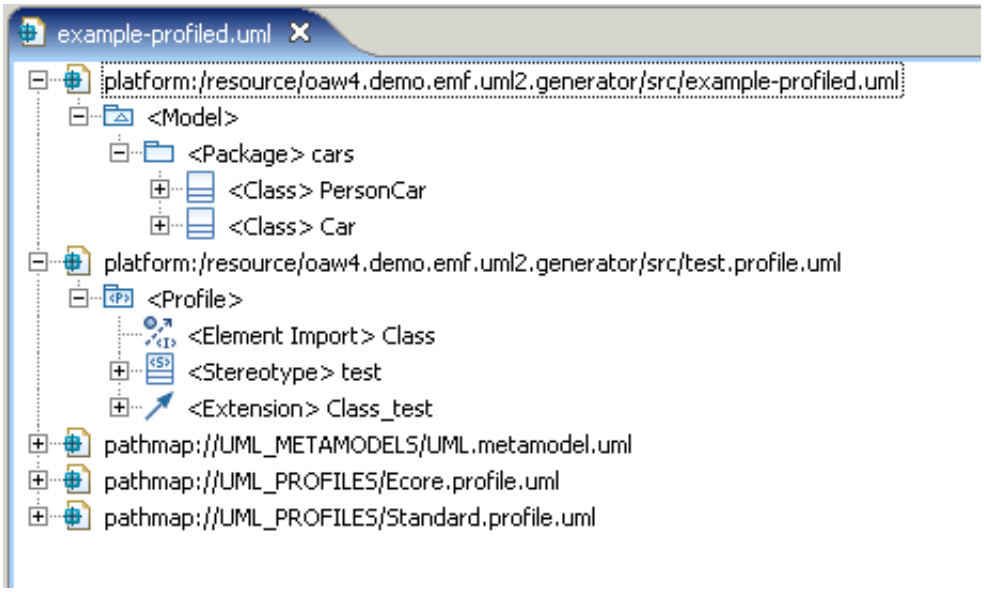


Figure 9.6. Loading the Profile

Now, to make the following stuff work, you first have to select the profile and select the *Profile -> Define* operation from the UML2 Editor menu. This creates all kinds of additional model elements, about which you should not care for the moment.

Now, finally, you can select your `cars` package (the one from the example model) and select *Package -> Apply Profile* from the UML2 Editor menu. Select your test profile to be applied.

For the purpose of this example, you should now apply the test stereotype to the `PersonCar` class. Select the class, and the select *Element -> Apply Stereotype* from the UML2 Editor menu. This should result in the following model:

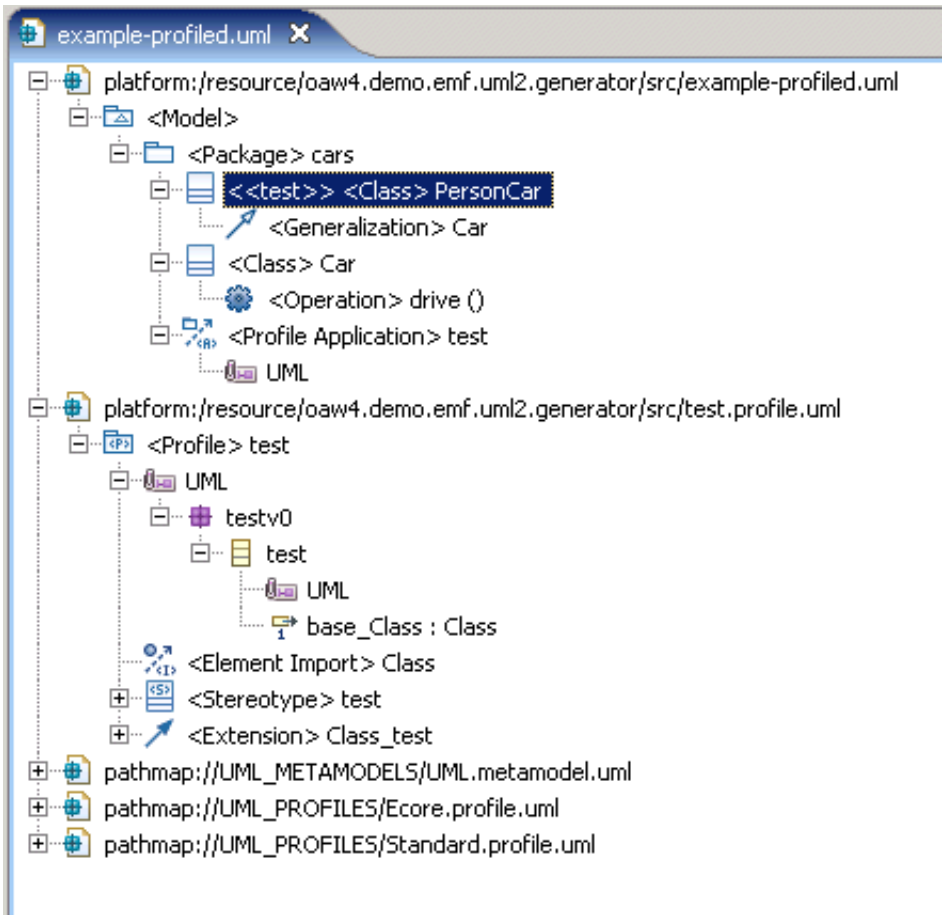


Figure 9.7. Defining the Profile

9.5.3. Generating Code

Note that all the stuff above was not in any way related to oAW, it was just the "bare bones" means of creating and applying a profile to a UML2 model

There are two things we have to change: The workflow (specifically, the configuration of the generator component) needs to know about the profile, and the template needs to generate different code if a class has the test stereotype applied. Let us look at the second aspect first. Here is the modified template (in `RootWithProfile.xpt`):

```
«DEFINE Root FOR uml::Model»
 «EXPAND PackageRoot FOREACH (List[uml::Package])ownedElement»
«ENDDEFINE»

«DEFINE PackageRoot FOR uml::Package»
 «EXPAND ClassRoot FOREACH (List[uml::Class])ownedType»
«ENDDEFINE»

«DEFINE ClassRoot FOR uml::Class»
«FILE name+".java"»
 public class «name» {}
«ENDFILE»
«ENDDEFINE»

«DEFINE ClassRoot FOR test::test»
«FILE name+".java"»

public class «name» {} // stereotyped
«ENDFILE»
«ENDDEFINE»
```

As you can see, the stereotype acts just like a type, and even the polymorphic dispatch between the base type (`uml::Class`) and the stereotype works.

Adapting the workflow file is also straight forward (`workflowWithProfile.oaw`), here is the modified generator component:

```
<component id="generator" class="oaw.xpand2.Generator"
skipOnErrors="true">
 <metaModel class="oaw.type.emf.EmfMetaModel"
  metaModelPackage="org.eclipse.emf.ecore.EcorePackage"/>
 <metaModel class="oaw.uml2.UML2MetaModel"/>
 <metaModel id="profile"
  class="oaw.uml2.profile.ProfileMetaModel">
  <profile value="test.profile.uml"/>
 </metaModel>
 <expand
  value="templates::RootWithProfile::Root FOR model"/>
  <outlet path="src-gen">
   <postprocessor class="oaw.xpand2.output.JavaBeautifier"/>
  </outlet>
</component>
```

The only thing, we have to do is to add a new metamodel that references the profile we just created.

Chapter 10. EMF Validation Adapter

10.1. Introduction

The EMF Validation Adapter integrates the *Check* language of oAW with the validation framework of EMF and thereby with any validation aware model editor. This includes editors generated by EMF and GMF diagram editors. It obsoletes the old GMF2 adapter.

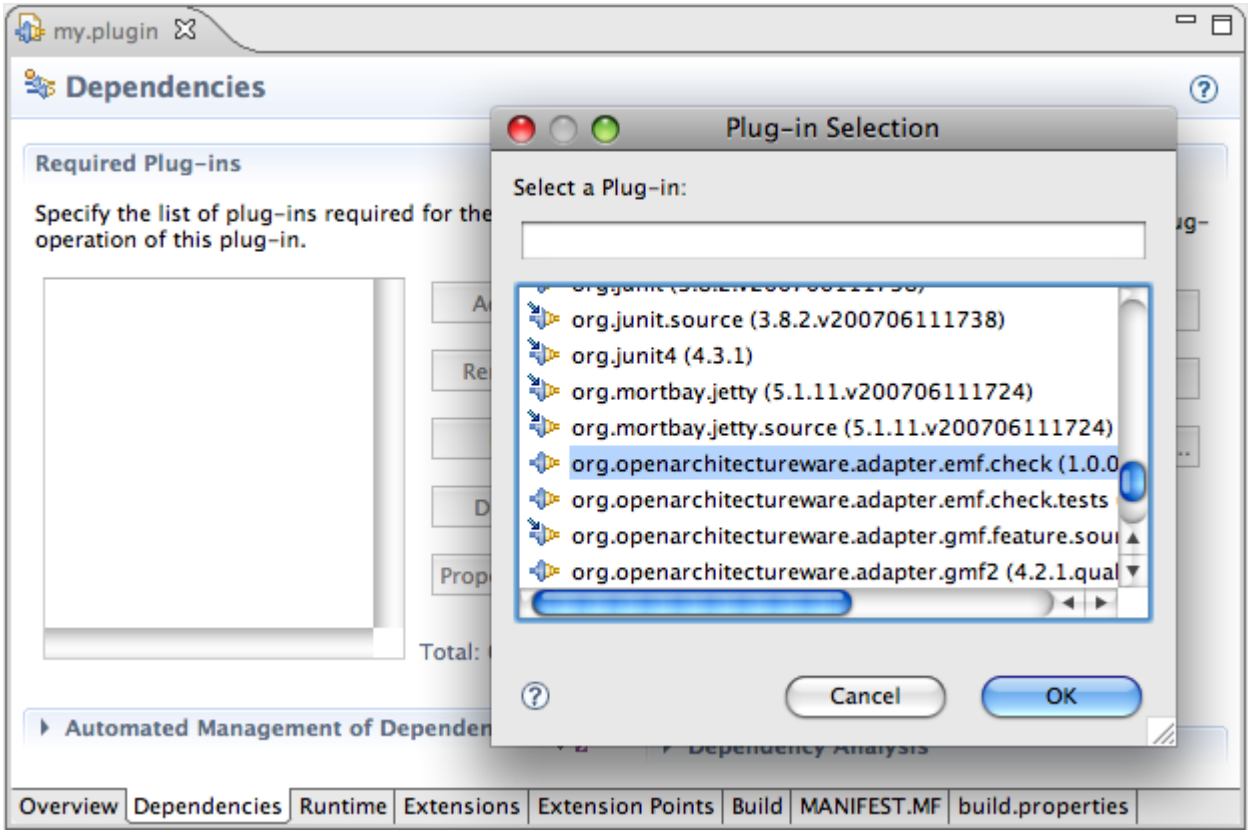
Validation errors and warnings are usually converted into markers by the model editors. These markers appear in the *Problems* view.

10.2. EValidation Adpater Setup

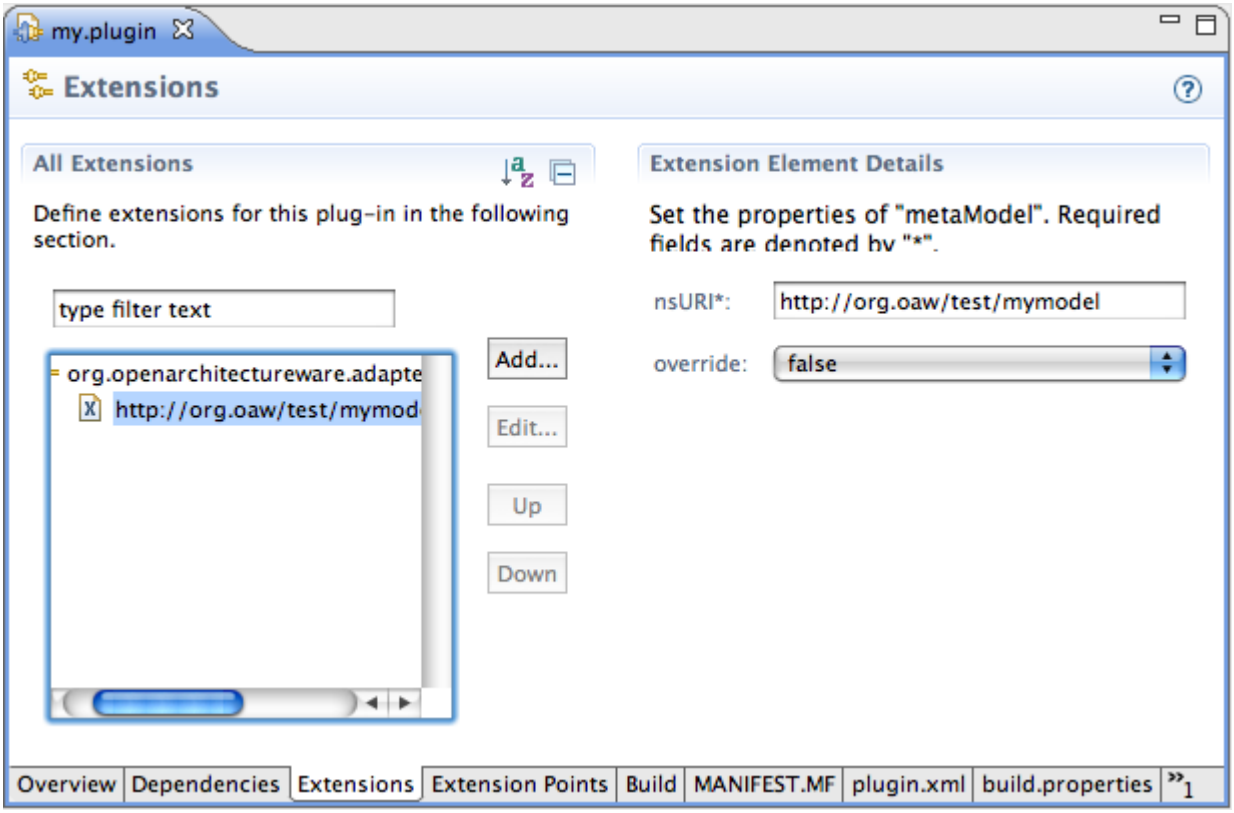
The oAW EMF Validation Adapter is available from the oAW update site.

We assume you have working Ecore model and a Check file in a plug-in in your workspace. The validation adapter is contributed by means of an eclipse extension.

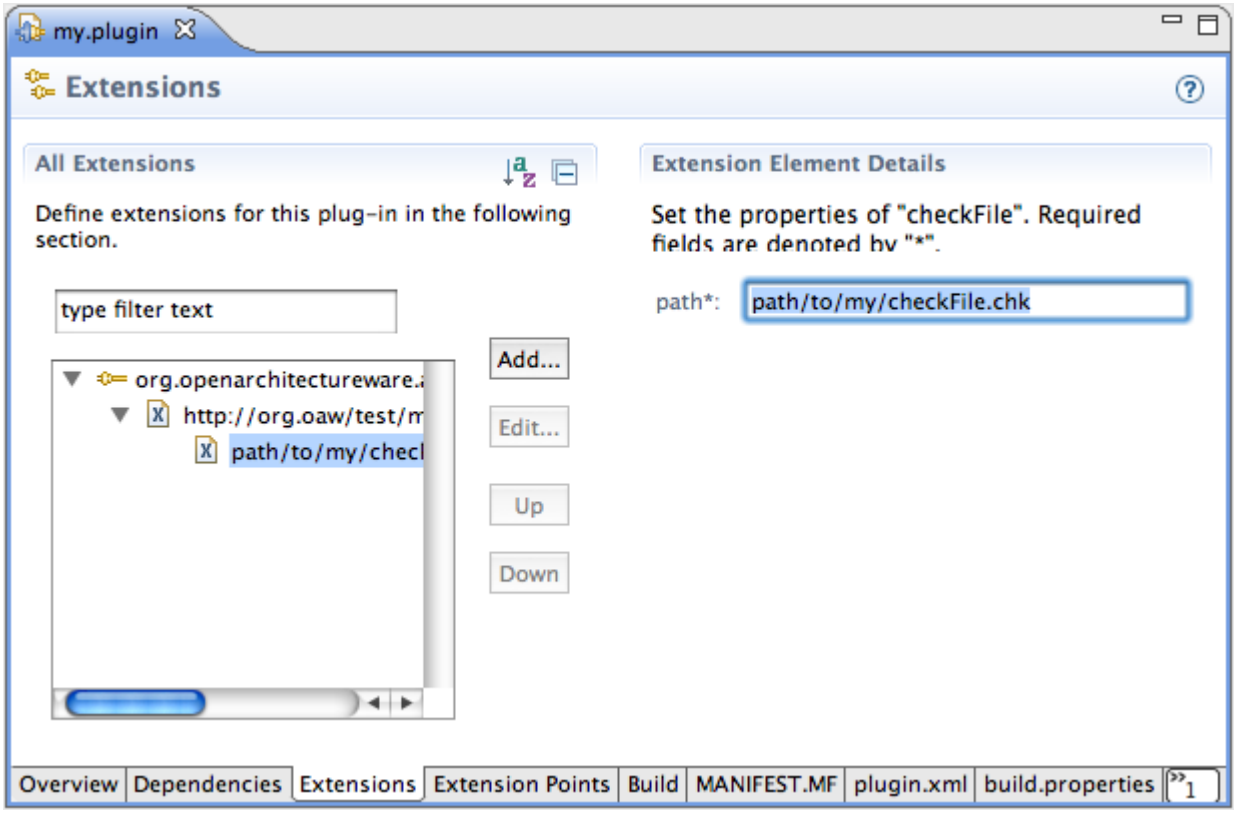
1. Open the MANIFEST.MF editor of your plug-in containing the Check file and add a dependency to the plug-in `org.openarchitectureware.adapter.emf.check` in the dependency tab.



2. Go to the Extensions tab of your plug-in in the MANIFEST.MF editor, click *Add...* and choose the extension point `org.openarchitectureware.adapter.emf.check.checks`. Enter the nsURI of your metamodel. If yu don't remember the nsURI, it is one property of the EPackage of your Ecore model.



3. For each Check file, right click on the metamodel entry, select *New->checkFile* and enter the path to the file. The file extension *.chk* can be ommitted.



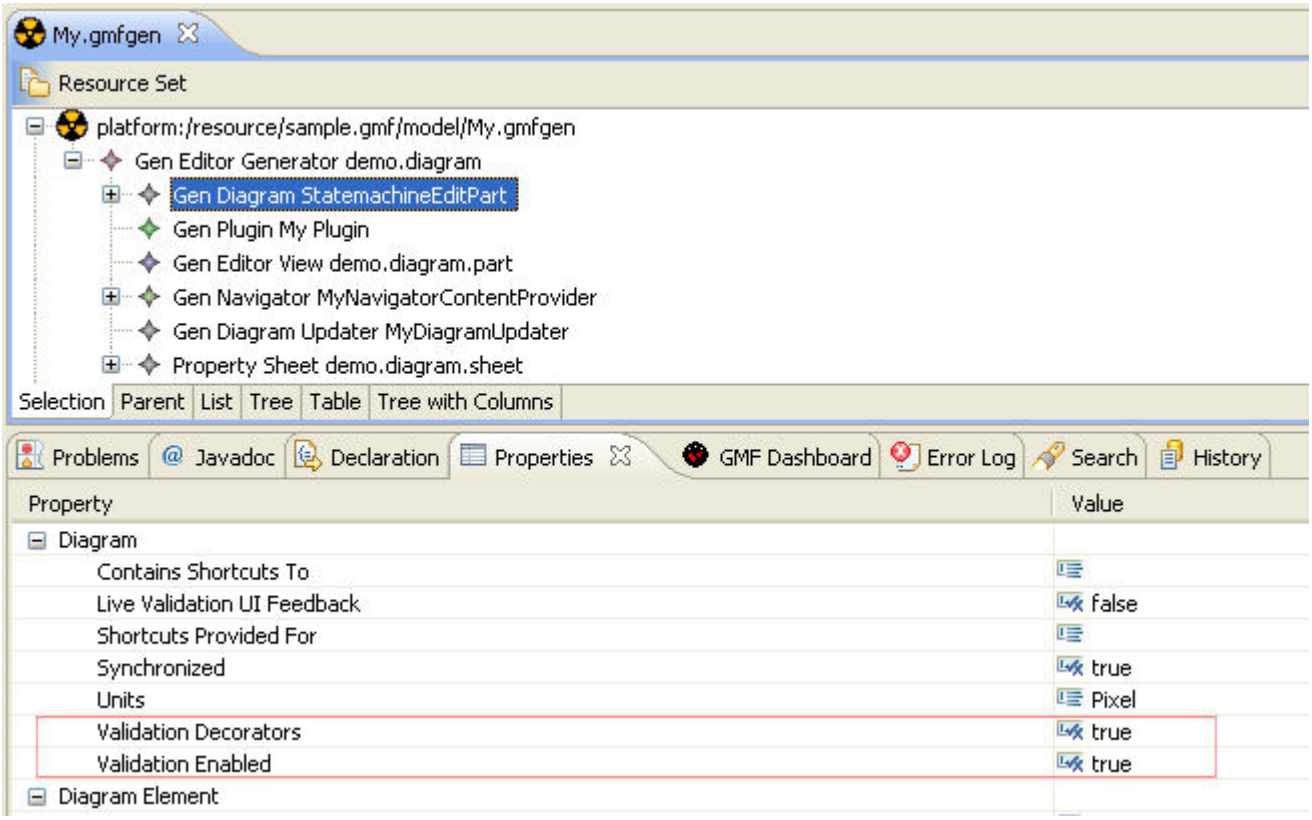
10.3. Setting Up Validation in EMF Editors

No further steps should be necessary. The validation action in the menu bar is only enabled if you select an element in the editor.

10.4. Setting Up Validation in GMF Diagram Editors

In addition to the steps described above, you have to enable validation explicitely. Open the GMF diagram editor's `.gmfgen` file and set the following properties on the your `GenDiagram` element:

- Set *Validation Decorators* in the section "*Diagram*" to *true*
- Set *Validation Enabled* in the section "*Diagram*" to *true*



Chapter 11. RSM/RSA Adapter

11.1. Introduction

The IBM Rational Software Architect / Modeler (RSA/RSM) is a UML modeling tool. It is based on Eclipse and the UML 2.0 project. The UML models stored in a special file format. These model files have the file extension `.emx`, UML profiles are stored in files with the extension `.epx`. The files from version 6.0 of RSA/RSM can be opened with a special API only. With the introduction of version 7.0, the models and profiles can be opened directly with the UML2 2.0 framework.

Usually RSA/RSM profiles are deployed in plugins, the referenced to these profiles are written with a special URI called pathmap (people working with Rational Rose might know this).

The RSA/RSM adapter brings the functionality of the UML2 adapter written by Sven to RSA/RSM. The adapter for oAW is available since version 4.1. It is based upon the UML2 2.0 framework and provides a type mapping from the UML 2.1 metamodel implementation to the oAW type system. Hence, one can access the whole UML2 metamodel from *Check*, *Xtend*, and *Xpand*. Additionally, and more important, the adapter dynamically maps stereotypes, stored in RSA/RSM specific model files (`.epx`), which are applied to model elements, to oAW types. Furthermore, stereotype properties (tagged values) are mapped to oAW properties. You do not have to implement additional Java classes anymore. Just export your models and applied profiles. That's all!

11.2. Installation

Using the RSA/RSM adapter with Eclipse, you first need to install the UML2 feature from eclipse.org.

http://download.eclipse.org/tools/uml2/updates/site-interim.xml

The oAW uml2adapter is available from the oAW update site:

http://www.openarchitectureware.org/updatesite/milestone/site.xml

(Go to the home page of the project and find the current location if either of the sites do not work)

Restart your Eclipse workbench when Eclipse asks you to do so.

11.3. Setting up IBM Rational Software Architect / Modeller

You need to configure your project (or the whole workspace) to use the RSA/RSM adapter.

Right click on the project and choose '*properties*' from the pop-up menu. Open the '*openArchitectureWare*' tab, activate the check boxes (*nature* and *specific metamodel contributors*) and add the *RSA/RSM Profiles* metamodel contributor.

All profiles installed in plugins using the `com.ibm.xtools.uml2.msl.UMLProfiles` extension point are added as metamodels (for more information about creating your own profiles with RSA/RSM see help for "Extending Rational Software Modeler functionality") .

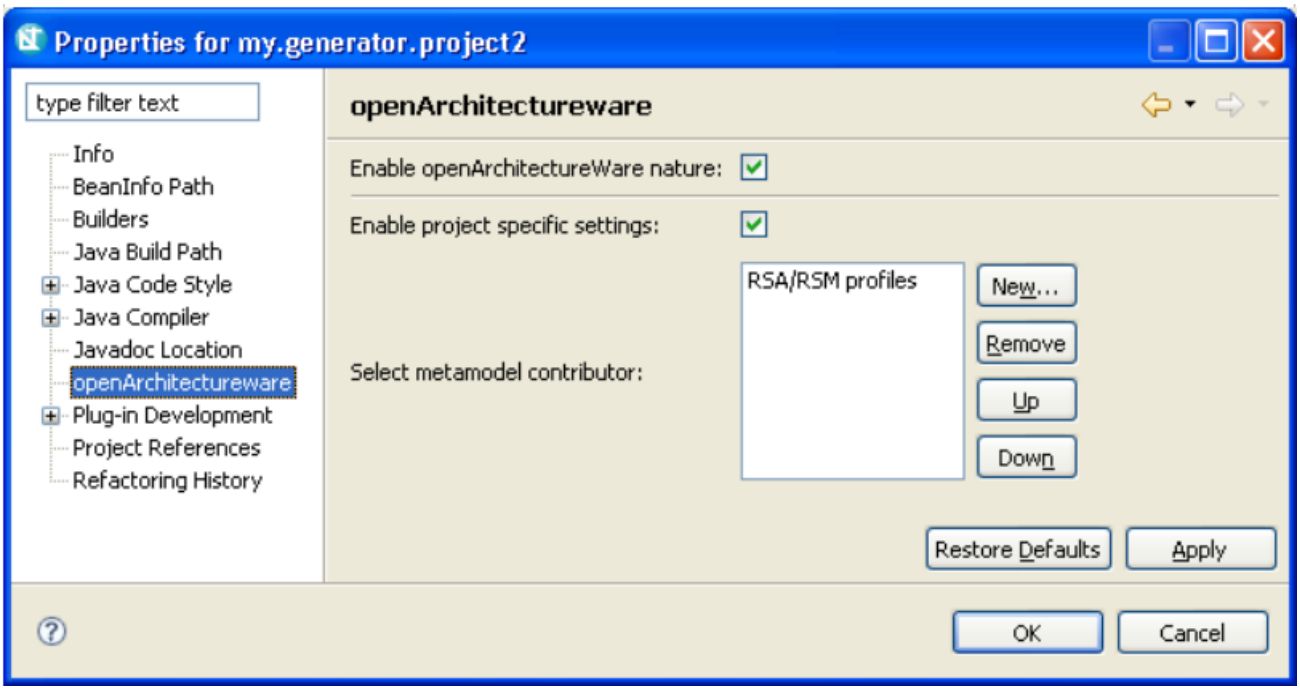


Figure 11.1. Project properties with RSA/RSM profiles added

11.3.1. Runtime Configuration

At runtime, you just need the `org.openarchitectureware.rsdp.adapter-4.1.0.jar` (or later). You can use the dependency of Eclipse mechanism from the PDE, or copy or reference the respective JAR file directly. It does not matter, you just have to take care that it is on the classpath.

11.3.2. Workflow using profiles from plugins

If you have written some *Check*, *Xtend* or *Xpand* files and now want to execute them you have to provide the right configuration.

You need to configure the UML2 metamodel and a profile metamodel for each profile you used directly. A typical configuration looks like this:

```
<workflow>
 <property
  name="umlmda.profile.plugin.path"
  value="C:/winapp/eclipse/v3.3.2/UMLMDA/eclipse/plugins/org.umlmda.u
ml2.profile_2.0.0"/>
 <!-- initializes resourcemaps, urimaps, etc. -->
 <component
   class="org.openarchitectureware.rsdp.workflow.RSASetup" >
  <pathMapEntry>
   <alias value="UMLMDA_PROFILE"/>
   <path value="${umlmda.profile.plugin.path}/profiles"/>
  </pathMapEntry>
  <init value="true"/>
 </component>
 <component class="oaw.emf.XmiReader">
   ...
 </component>
 <component class="oaw.xpand2.Generator">
  <metaModel class="oaw.uml2.UML2MetaModel"/>
  <metaModel class="oaw.uml2.profile.ProfileMetaModel">
   <profile value="pathmap://UMLMDA_PROFILE/UMLMDAProfile.epx"/>
  </metaModel>
    ...
 </component>
<workflow>
```

11.3.3. Workflow using profiles from workspace projects

If the RSA/RSM profiles are installed in a workspace project rather in a plugin, it can be accessed with a workflow below:

```
<workflow>
 <property
  name="umlmda.profile.project.path"
  value="C:/devel/umlmda.rsm7/org.umlmda.uml2.profile"/>
 <!-- initializes resourcemaps, urimaps, etc. -->
 <component
  class="org.openarchitectureware.rsdp.workflow.RSASetup" >
  <umlExtensionProject
   class="org.openarchitectureware.rsdp.workflow.UmlExtensionProje
ct">
   <projectName value="UMLMDAProfile"/>
   <path value="${umlmda.profile.project.path}"/>
  </umlExtensionProject>
  <init value="true"/>
 </component>
 <component class="oaw.emf.XmiReader">
  <modelFile value="${model}"/>
  <outputSlot value="model"/>
  </component>
 <component class="oaw.xpand2.Generator" skipOnErrors="true">
  <metaModel class="oaw.type.emf.EmfMetaModel"
   metaModelPackage="org.eclipse.emf.ecore.EcorePackage"/>
  <metaModel class="oaw.uml2.UML2MetaModel"/>
  <metaModel class="oaw.uml2.profile.ProfileMetaModel">
   <profile
    value="file://${umlmda.profile.project.path}/profiles/UMLMDAP
rofile.epx"/>
  </metaModel>
  ...
 </component>
```

11.3.4. Future enhancements

The next version of the RSA/RSM adapter will support profiles found in the classpath of the generator project. In addition, profiles can be added more selectively.

Chapter 12. Recipe Framework

12.1. Introductory Example and Plugin

Currently, it not feasible in MDSD to generate 100% of an application. Usually, you generate some kind of implementation skeleton into which developers integrate their own, manually written code. For example, the generator generates an abstract base class, from which developers extend their implementation classes – which contains the application logic.

The screenshot above shows an example – and also illustrates the use of the *Recipe* plugin. Let us look at the various items in the screenshot:

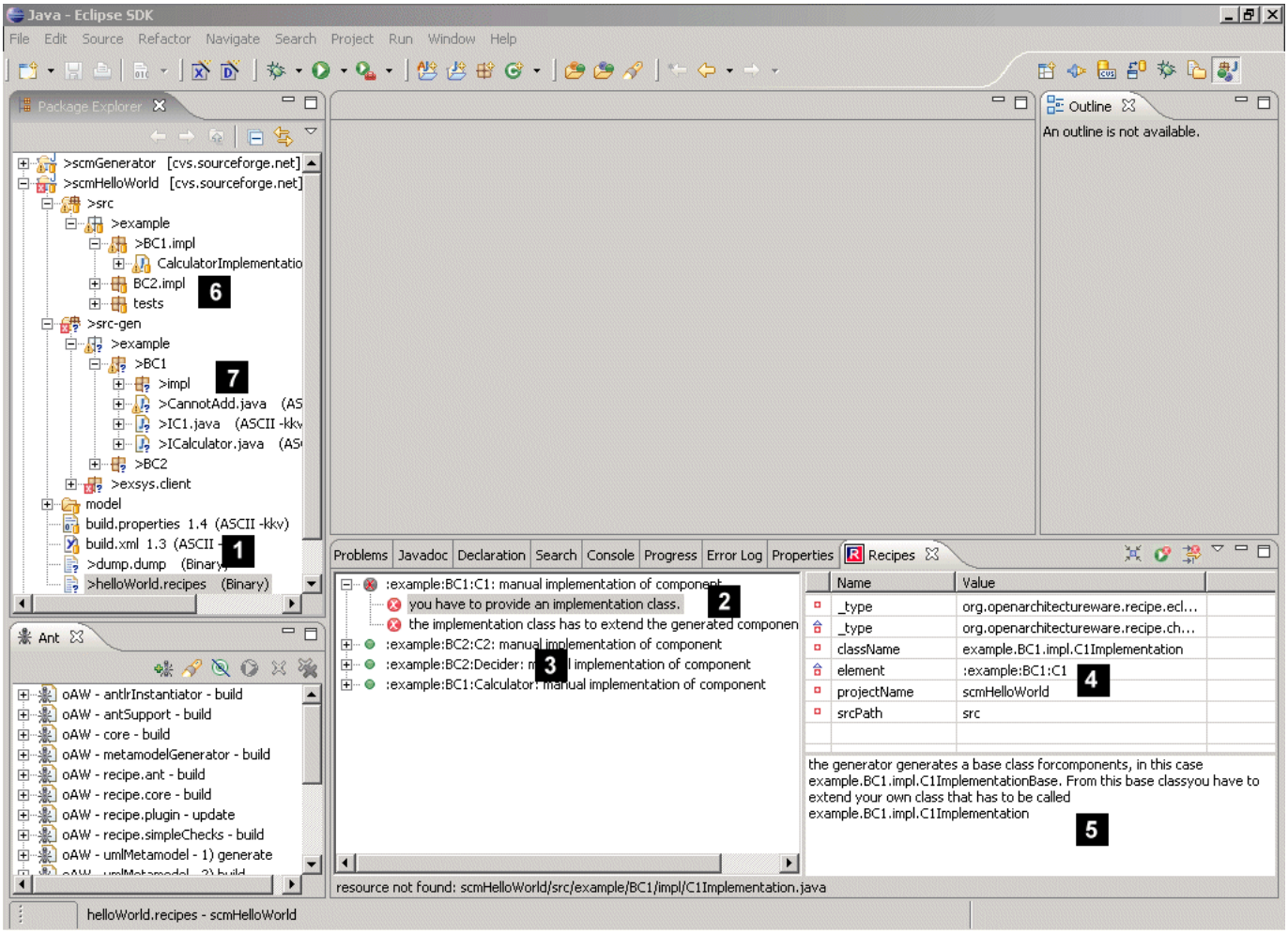


Figure 12.1. Recipe Example

1. The generator creates a so-called `.recipes` file. How this file is created, will be shown below. The recipe file contains the checks that need to be executed on the code base. In the context menu of a `.recipes` file you will find an item called *Open Recipe File* that opens the checks in the respective browser. Note that the file has to have a `.recipes` extension – otherwise the plugin will not open the file.

2. Here you can see failed checks. The messages tell you that you have to write the implementation class, and (the second item) you will have to make sure it extends from the generated base class.

3. Here, you can see a check that worked well.

4. Each check contains a number of parameters; the table shows the parameters of the selected check. For example the name of the class to be generated, the one from which you have to inherit, etc. This is basically for your information. If you double-click on a certain row, the value of the parameter is copied to the clipboard – and you can past it to wherever you need.

5. Here, you can see a descriptive text that explains, what you have to do, in order to fix the failed check.

6. Here, is the manually written code (in the `src` folder). You can see the class`CalculatorImplementation` – this is why the check that verifies its presence is successful. There is no `C1Implementation` why the check that checks its presence fails.

7. This is the generated code folder (`src-gen`). It contains the generated base classes.

There are a couple of options to work with those recipes, as shown in the next illustration:

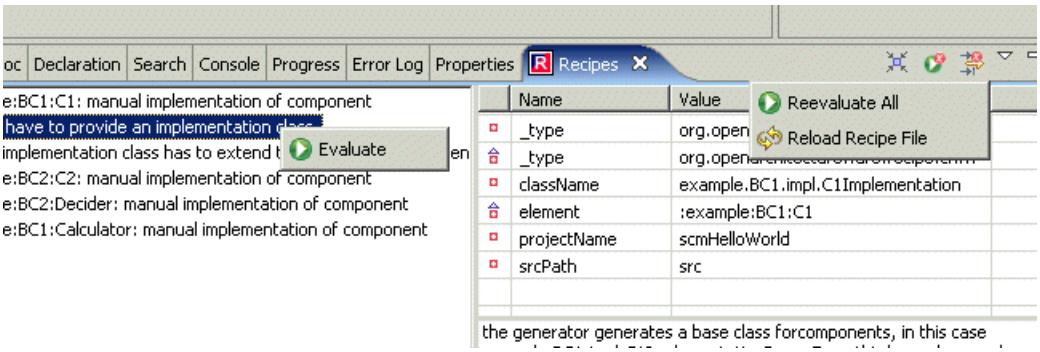


Figure 12.2. Using the Recipe plugin

If you right-click on a check in the tree view, you can reevaluate the check explicitly. At the top-right of the view, there are three buttons. The first on collapses the tree view. The *"play button with the small red cross"* reevaluates all the failed checks - and only those! The third button is a filter button; if selected, the tree view hides the checks that are OK. In the drop down menu of the view (activated with the small downward-pointing triangle), there are two entries: the green *run* button labelled "reevaluate all" reevaluates all checks, even those that have been successful before. And finally, the reload button reloads the recipe file, and reevaluates everything.

There are two important automation steps:

- First of all, you should make sure that when you run the generator – typically through an ant file – the workspace will be refreshed after the ant run (you can configure this in Eclipse). If you do this, the view will automatically reload the recipe file and reevaluate all the checks.

- There are certain kinds of checks that automatically reevaluate if the workspace changes. This means that, for example, if you add the implementation class in the above example, as soon as you save the class file, the check will succeed. The question, which checks will be evaluated automatically, has to be defined in the check definition – see below.

12.1.1. Installing the Plugin

There are two steps: The first one installs the plugin itself, i.e. the Recipe browser view, etc. The respective plugin is `org.openarchitectureware.recipe`. As usual, you install it by copying it into the Eclipse plugin

directory or just by downloading it from the oAW update site. If the plugin is installed in this way, it can only evaluate the (relatively trivial) checks in the `recipe.simpleChecks` project. To check useful things, you will have to extend the plugin – you have to contribute the checks that should be executed. For this purpose, the org.openarchitectureware.recipe plugin provides the `check` extension point. A number of useful checks that can be evaluated in Eclipse are contained in the org.openarchitectureware.recipe.eclipseChecks plugin. You should install that one, too. It also comes automatically from the update site.

In general, this means: whenever you develop your own checks and want to evaluate them in Eclipse, you have to contain them in a plugin and extend the above-mentioned extension point. Otherwise it will not work.

12.1.2. Referencing the JAR files

In order for the workflow to find the recipe JARs, they need to be on the classpath. The easiest way to achieve that is to make your generator project a plugin project and reference the recipe plugins in the plugin dependencies (all the oAW plugins with recipe in their name). This will update your classpath and add the necessary JARs. If, for some reason, you do not want your projects to be plugins, you have to reference the JAR files of the above mentioned plugins manually.

12.2. Executing Recipe Checks

12.2.1. Running Checks within your workflow

You have to write a workflow component that creates the recipes. Your custom workflow component has to extend the `RecipeCreationComponent` base class and overwrite the `createRecipes()` operation. The operation has to return the collection of recipes that should stored into the recipe file. In the workflow file, you then have to configure this component with the name of the application project in Eclipse, the source path where the manually write code can be found, and the name of the recipe file to be written.

Please take a look a the *emfHelloWorld* example. It contains an extensive example of how to use the recipes in oAW 4.

12.2.2. Running Checks within Ant

You can also check the recipes using ant. Of course you cannot use those nice and cool interactive checks – and you also cannot use Eclipse-based checks. They can be in the recipes file, since the ant tasks skips them automatically. The following piece of ant code shows how to run the checks – should be self-explanatory. Note that you have to use all the jar files from the recipe.ant project.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<project name="scm - hello world - generate" default="generate">
  <property file="build.properties" />
  <path id="ant.runtime.classpath">
   <pathelement location="${GENROOT}" />
   <fileset dir="${GENROOT}" includes="*.jar"/>
   <fileset dir="${OAWROOT}/dist" includes="*.jar"/>
   <fileset dir="${RECIPE.CORE.DIR}/dist" includes="*.jar"/>
   <fileset dir="${RECIPE.SIMPLECHECKS.DIR}/dist" includes="*.jar"/>
   <fileset dir="${RECIPE.ANT.DIR}/dist" includes="*.jar"/>
   <fileset dir="${RECIPE.ANT.DIR}/lib" includes="*.jar"/>
   <fileset dir="${RECIPE.ECLIPSECHECKS.DIR}" includes="*.jar"/>
  </path>
  <target name="check" depends="">
   <taskdef name="check"
     classname="org.openarchitectureware.recipe.ant.RecipeCheckTask">
     <classpath refid="ant.runtime.classpath" />
   </taskdef>
   <check recipeFile="L:/workspace/xy/helloWorld.recipes"/>
  </target>
</project>
```

The checks use log4j logging to output the messages. So you can set the log level using the log4j.properties file. The following output shows all the checks being successful:

```
Buildfile: l:\exampleWorkspace-v4\scmHelloWorld\build.xml
check:
  [check] 0    INFO  - checking recipes from file:
     L:/runtime-EclipseApplication/xy/helloWorld.recipes
BUILD SUCCESSFUL
Total time: 1 second
```

If you set the log level to DEBUG, you will get more output. Here, in our case, you can see that all the Eclipse checks are skipped.

```
Buildfile: l:\exampleWorkspace-v4\scmHelloWorld\build.xml
check:
  [check] 0    INFO  - checking recipes from file: L:/runtime-
      EclipseApplication/xy/helloWorld.recipes
  [check] 60   DEBUG - [skipped] resource exists exists -
      skipped - mode was batch only.
  [check] 70   DEBUG - [skipped] resource exists exists -
      skipped - mode was batch only.
  [check] 70   DEBUG - [skipped] resource exists exists -
      skipped - mode was batch only.
  [check] 80   DEBUG - [skipped] resource exists exists -
      skipped - mode was batch only.
  [check] 80   DEBUG - [skipped] resource exists exists --
      skipped - mode was batch only.
  [check] 90   DEBUG - [skipped] resource exists exists -
      skipped - mode was batch only.
  [check] 90   DEBUG - [skipped] resource exists exists -
      skipped - mode was batch only.
  [check] 90   DEBUG - [skipped] resource exists exists -
      skipped - mode was batch only.
BUILD SUCCESSFUL
Total time: 1 second
```

If there are errors, they will be output as an ERROR level message.

12.2.3. Implementing your own Checks

12.2.3.1. Hello World

The following piece of code is the simplest check, you could possibly write:

```
package org.openarchitectureware.recipe.checks.test;

import org.openarchitectureware.recipe.core.AtomicCheck;
import org.openarchitectureware.recipe.eval.EvaluationContext;

public class HelloWorldCheck extends AtomicCheck {
 private static final long serialVersionUID = 1L;
 public HelloWorldCheck() {
  super("hello world", "this check always succeeds");
 }

 public void evaluate(EvaluationContext c) {
  ok();
 }
}
```

A couple of notes:

- You can define any kind of constructor you want – passing any parameters you like. You *have to* pass at least two parameters to the constructor of the parent class: the first one is the name, a short name, of the check. The second parameter is the somewhat longer message. You can call `setLongDescription()` if you want to set the explanatory text.

- You can pass a third parameter to the constructor of the parent class: and that is one of the two constants in `EvalTrigger` . By default, the `EvalTrigger.ON_REQUEST` is used which means that the check is only evaluated upon explicit request. If you pass `EvalTrigger.ON_CHANGE` , the check will be automatically re-evaluated if the Eclipse workspace changes.

- You should define a serial version UID since Java serialization is used for the recipe file.

- In the `evaluate()` method you do the check itself. We will explain more on that later.

12.2.3.2. More sensible Checks

More sensible checks distinguish themselves in two respects:

- First, you will typically pass some parameters to the constructor which you will store in member variables and then use in the `evaluate()` operation.

- You can store parameters for display in the table view in the plugin. You can use the `setParameter(name, value)` operation for that. More on that below.

- The evaluation will contain a certain business logic.

An example:

```
public void evaluate(EvaluationContext c) {
  if ( something is not right ) {
   fail( "something has gone wrong");
  }
  if ( some condition is met ) {
   ok();
  }
```

By the way, you do not need to care about the EvaluationContext. It is only needed by the framework.

12.2.3.3. Eclipse Checks

Eclipse checks are a bit special. If the check were implemented in the way described above, you would have a lot of dependencies to all the Eclipse plugins/jars. You would have these dependencies, as soon as you instantiate the check – i.e. also in the generator when you configure the check. In order to avoid this, we have to decouple the configuration of a check in the generator and its evaluation later in Eclipse:

1. During configuration we do not want any Eclipse dependencies since we do not want to "import" half of Eclipse into out ant-based code generator
2. However, when evaluating the check we obviously need the Eclipse dependencies, otherwise we could not take advantage of Eclipse-based checks in the first place.

An Eclipse check is thus implemented in the following way. First of all, our check has to extend the `EclipseCheck` base class.

```
public class ResourceExistenceCheck extends EclipseCheck {
```

Again, we add a serial version UID to make sure deserialization will work.

```
private static final long serialVersionUID = 2L;
```

In the constructor we decide whether we want to have this check evaluated whenever the Eclipse workspace changes (`EvalTrigger.ON_CHANGE`) or not. We also store some of the parameters in the parameter facility of the framework. Note that we *do not implement* the `evaluate()` operation!

```
   public ResourceExistenceCheck(String message,
    String projectName, String resourceName) {
  super("resource exists exists", message, EvalTrigger.ON_CHANGE);
  setProjectName(projectName);
  setResourceName(resourceName);
 }

 private void setProjectName(String projectName) {
  setParameter("projectName", projectName);
 }

 private void setResourceName(String resourceName) {
  setParameter("resourceName", resourceName);
 }
```

In order to provide the evaluation functionality, you have to implement an `ICheckEvaluator`. It has to have *the same qualified name* as the check itself, postfixed with `Evaluator`. During the evaluation of the check, the class is loaded dynamically based on its name. A wrong name will result in a runtime error during the evaluation of the Eclipse plugin.

```
public class ResourceExistenceCheckEvaluator
implements ICheckEvaluator {

 public void evaluate(AtomicCheck check) {
   String projectName =
     check.getParameter("projectName").getValue().toString();
   String resourceName =
    check.getParameter("resourceName").getValue().toString();
   IWorkspace workspace = ResourcesPlugin.getWorkspace();
   IResource project =
     workspace.getRoot().getProject(projectName);
   if (project == null)
    check.fail("project not found: "+projectName);
   IFile f = workspace.getRoot().getFile(
    new Path(projectName+"/"+resourceName) );
   String n = f.getLocation().toOSString();
   if ( !f.exists() ) check.fail(
    "resource not found: "+projectName+"/"+resourceName);
    check.ok();
 }
```

When implementing the evaluator, you can basically do the same things as in the `evaluate()` operation in normal checks. However, in order to set the `ok()` or `fail("why")` flags, you have to call the respective operations on the check passed as the parameter to the operation.

12.2.3.4. Making checks available to the Eclipse plugin

In order to allow the Eclipse runtime to execute your checks, it has to find the respective classes when deserializing the recipe file. This is a bit tricky in Eclipse, since each plugin has its own class loader. So, assume you want to define your own checks and want to use them in Eclipse; what you have to do is: implement your own plugin that extends a given extension point in the Recipe Browser plugin. The following XML is the plugin descriptor of the org.openarchitectureware.recipe.eclipseChecks.plugin.EclipseChecksPlugin , a sample plugin that comes with the recipe framework and provides a number of Eclipse-capable checks.

```
    <?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
  id="org.openarchitectureware.recipe.eclipseChecks"
  name="%plugin_name"
  version="4.0.0"
  provider-name="%provider_name"
  class="org.openarchitectureware.recipe.\
  eclipseChecks.plugin.EclipseChecksPlugin">
```

Here we now define the jar file that contains our checks (This will be important below)

```
   <runtime>
   <library name="oaw-recipe-eclipsechecks.jar">
     <export name="*"/>
   </library>
  </runtime>
```

The required plugins mainly depend on the implementations of `CheckEvaluator`, of course, however, you have to make sure the dependencies contains the org.openarchitectureware.recipe plugin, since you are going to extend an extension point defined in this plugin.

```
   <requires>
   <import plugin="org.eclipse.ui"/>
   <import plugin="org.eclipse.core.runtime"/>
   <import plugin="org.eclipse.core.resources"/>
   <import plugin="org.openarchitectureware.recipe"/>
   <import plugin="org.eclipse.jdt.core"/>
  </requires>
```

This is the important line: here you specify that you extend the `check` extension point of the *Recipe* browser plugin. If you do not do this, deserialization of the recipe file will fail and you will get nasty errors. And yes, you need the dummy element; because otherwise, the class loading "magic" will not work.

```
   <extension point="org.openarchitectureware.recipe.check">
   <dummy/>
  </extension>
</plugin>
```

When you need to use the checks outside of Eclipse (e.g. in the generator for configuration/serialization purposes) you just add the JAR file of the plugin to your generator classpath. You *do not* need to add references to all the JAR files of the Eclipse plugin in the *requires* section, since these things will only be used during evaluation.

12.2.4. Framework components

Component	Plugin	Depends on	Description
recipe.core	Yes	-	Framework core. Needed whenever you do anything with recipes
recipe.ant	Yes	recipe.core	Contains the ant task to check recipes. Needed only for recipe evaluation in ant
recipe.simpleChecks	Yes	recipe.core	Contains a number of (more or less useful) sample checks
recipe.plugin	Yes	recipe.core	Contains the Eclipse Recipe Browser view
recipe.eclipsechecks.plugin	Yes	recipe.core	Contains the pre-packaged Eclipse checks.

12.2.5. List of currently available Checks

This table contains a list of all currently available checks. We are working on additional ones. Contributions are always welcome, of course! This list might, thus, not always be up to date – just take a look at the code to find out more.

Type	Classnames	Purpose
Batch	*org.openarchitectureware.recipe.checks.file.FileExistenceCheck*	Checks whether a given file exists
Batch	*org.openarchitectureware.recipe.checks.file.FileContentsCheck*	Checks whether a given substring can be found in a certain file
Eclipse	*org.openarchitectureware.recipe.eclipseChecks.checks. JavaClassExistenceCheck*	Checks whether a given Java class exists

Type	Classnames	Purpose
Eclipse	*org.openarchitectureware.recipe.eclipseChecks.checks. JavaSupertypeCheck*	Checks whether a given Java class extends a certain superclass
Eclipse	*org.openarchitectureware.recipe.eclipseChecks.checks. ResourceExistenceCheck*	Checks whether a given Eclipse Workspace Resource exists

Chapter 13. UML2Ecore Reference

13.1. What is UML2Ecore?

Building metamodels with the internal tools of EMF is tedious. Using the tree view based editors does not scale. Once you are at more than, say, 30 metaclasses, things become hard to work with. The same is true for the Ecore editor of GMF. Since you cannot easily factor a large metamodel into several diagrams, the diagram get cluttered and layouting becomes almost impossible.

One way to solve this problem is to use UML tools to draw the metamodel and then transform the UML model into an Ecore instance.

The oAW uml2ecore utiliy transforms a suitably structured Eclipse UML2 model (which can be created using various tools) into an Ecore file.

Note that this tool also serves as a tutorial for writing model-to-model transformations. This aspect, however, is documented elsewhere. This document only shows how to use the uml2ecore tool.

13.2. Setting up Eclipse

You need an installation of oAW 4.3 including the UML2 support. Run the UML2 example (available for download on the oAW download page http://www.eclipse.org/gmt/oaw/download) to verify that you have all the UML2 stuff installed.

The only additional thing required is that you install the *UML2Ecore* plugin into your Eclipse installation. The plugin is part of the oAW 4.3 distribution.

13.3. UML Tool Support

Of course, you could use the tree editors supplied by UML2 for drawing the UML2 model that should be transformed into Ecore. However, this is useless, since then you are back to square one: tree editors. So you need to use an UML tool that is able to export the model in the *Eclipse UML2 2.0* format. For example, MagicDraw 11.5 (or above) can do that; this tool (MD 11.5) is also the one we tested the `uml2ecore` utility with.

Note that we do not use any profiles in the `uml2ecore` utility. Although using profiles might make the metamodel a bit more expressive, we decided not to use a specific profile, to reduce the potential for compatibility problems (with the various tools).

13.4. Setting up your metamodel project

You should first create a new Generator project (select *File->New->Other->openArchitectureWare/Generator Project*).

Then open your UML2 tool of choice (we will use MagicDraw here) and draw a class diagram that resembles your metamodel. Here is the one we have drawn as an example for this document:

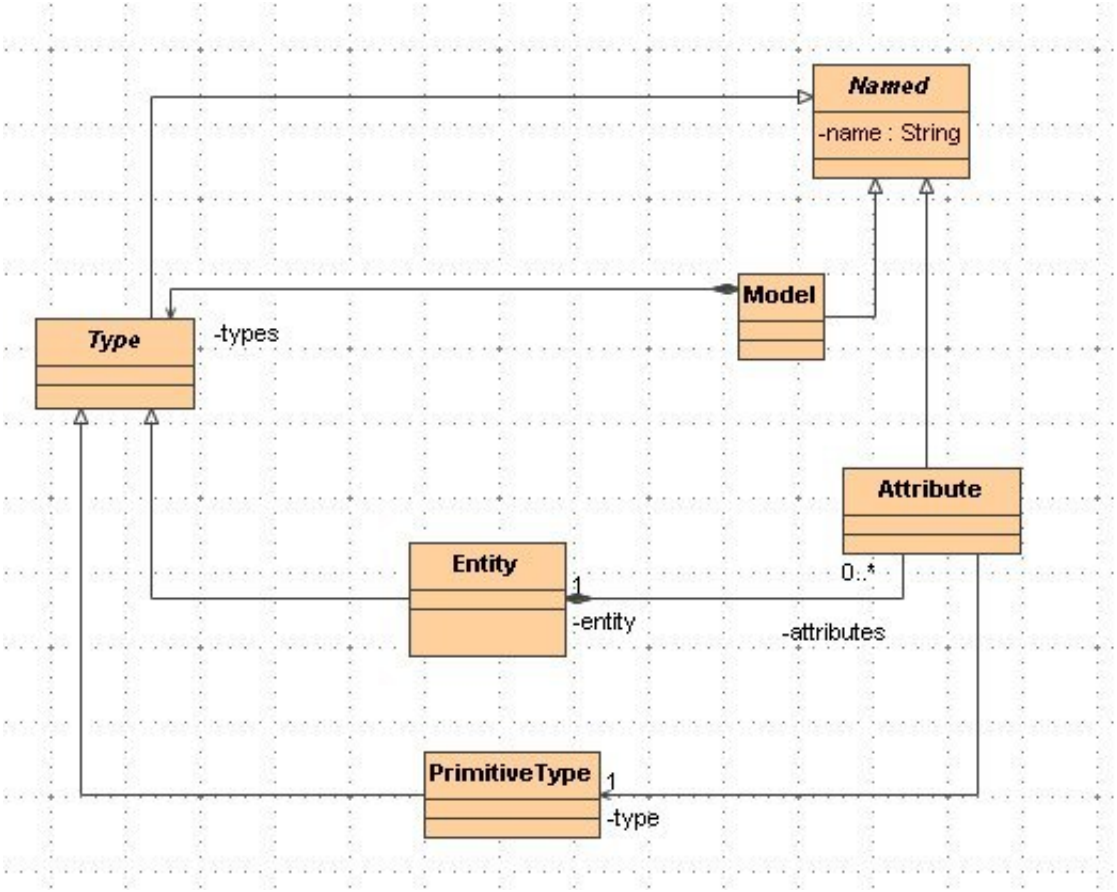


Figure 13.1. UML2Ecore sample metamodel - class diagram

This is the usual metamodel for entities and such. Nothing special. Also, the name of the model defines the name of the Ecore metamodel; here is the MagicDraw tree view to illustrate this:

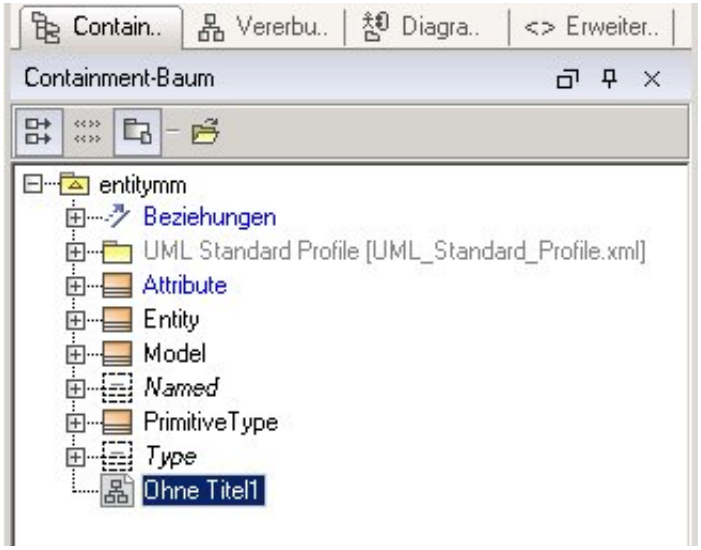


Figure 13.2. UML2Ecore sample metamodel - MagicDraw containment tree

You now have to save/export this model in Eclipse UML2 format. In MagicDraw, you do this by selecting File -> Export -> Eclipse UML2. The exported files have to be in the *root of the src folder* in the metamodel project created above. This is how the project looks like after this step, assuming you would have called your model `entity.uml2`:

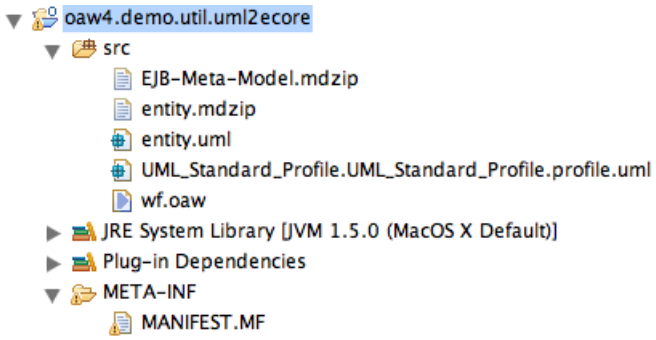


Figure 13.3. Project layout

Before you can actually run the generator, you have to make sure that your project has a plugin dependency to the uml2ecore plugin. Double-Click on the plugin manifest file, select the *Dependencies* tab and click add. Select the org.openarchitectureware.uml2ecore plugin. The result looks like this:

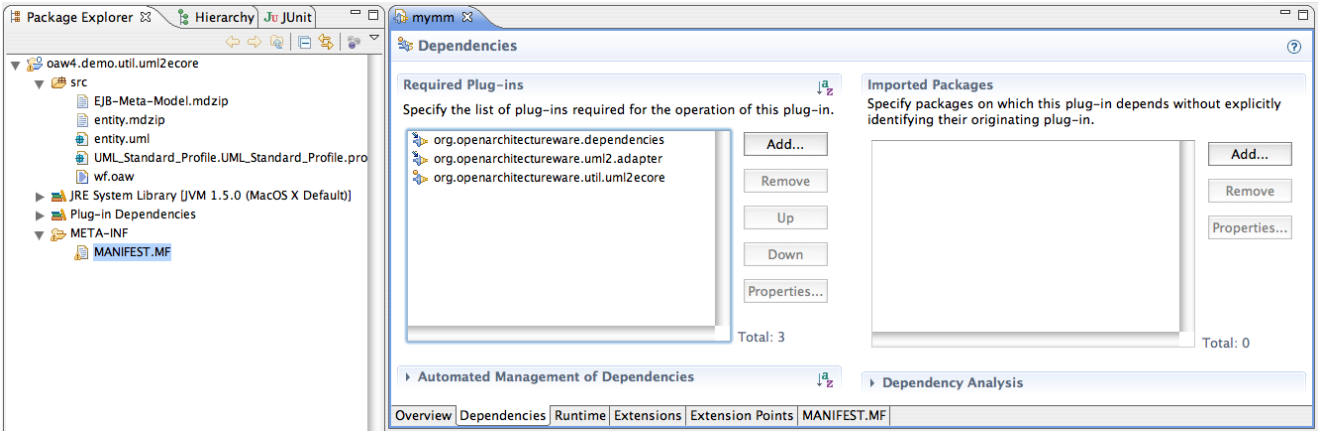


Figure 13.4. Dependencies

In the current version, you also need a dependency to the `org.openarchitectureware.util.stdlib` project. Later versions of the `uml2ecore` plugin will reexport that dependency, so that you will not need to add it you your projects manually.

13.5. Invoking the Generator

As usual, you have to write a workflow file. It also has to reside in your project source folder. Here is how it looks:

```
<?xml version="1.0"?>
<workflow>
<cartridge
 file="org/openarchitectureware/util/uml2ecore/uml2ecoreWorkflow.oaw"
  uml2ModelFile="org/openarchitectureware/uml2ecore/test/data/entity.uml2"
  outputPath="out"
  nsUriPrefix="http://www.voelter.de"
  includedPackages="Data"
  resourcePerToplevelPackage="false"
  addNameAttribute="false"/>
</workflow>
```

As you might expect, it simply calls a cartridge supplied by uml2ecore plugin. You have to define the following properties:

Table 13.1. UML2Ecore - Cartridge properties

Property	Description
uml2ModelFile	The is the name of your UML2 file that contains the model; as usual, the file is looked for in the classpath (that is why you had to move it into the source folder)
addNameAttribute	This determines whether automatic namespace management and naming is turned on (see the end of this document). For the simple example, please set the value to be false. *true/false*
nsUriPrefix	The nsUriPrefix is used to assemble the namespace URI. The name of the metamodel, as well as the nsPrefix, will be derived from the UML model name; so in our example, the nsPrefix and the name of the generated EPackage will be entitymm. The complete namespace URL also required by Ecore is created by concatenating the nsUriPrefix given here, and the name. The resulting namespace URL in the example will thus be http://www.voelter.de/entitymm.
includedPackages	This property etermines which packages the transformer should consider when transforming the model; note that the contents of all the packages will be put into the root `EPackage`.
outputPath	This property etermines where the resulting files are written to.
resourcePerToplevelPackage	It set to `true`, the generator will write a separate Ecore file for each of the top level packages in you UML model filtered by the value of the propery *includedPackages*. Useful for modularizing metamodels (see the end of this chapter).
nameUnnamedNavigableAssociationEnds	If set to `true`, all unnamed navigable association ends will be initialized with the name of the target type.

You can now run this workflow by selecting *Run As -> oAW Workflow*. The name of the generated Ecore file will correspond to the name of the root Model element in the UML model.

13.6. The generated Model

Here is a screenshot of the generated model:

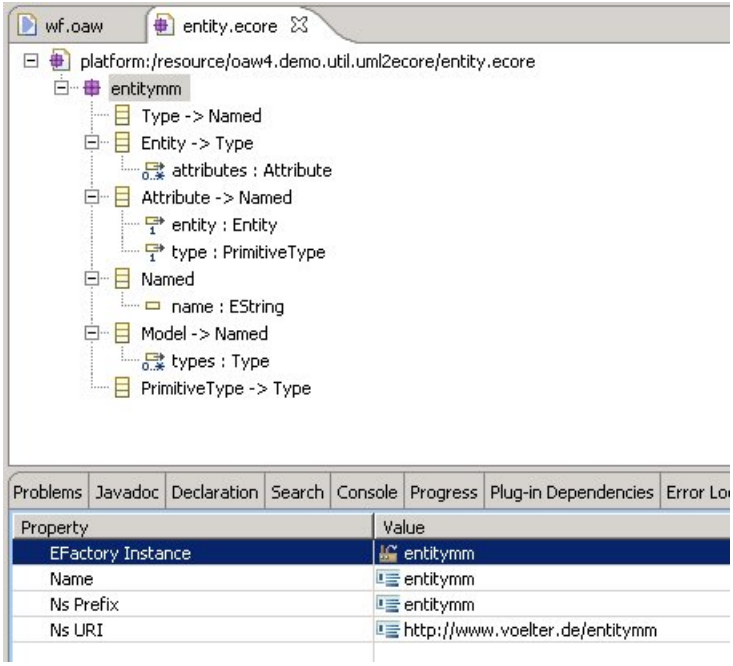


Figure 13.5. Generated Ecore model

The generator also creates a constraints file (called `entitymmConstraints.chk`) which contains a number of constraints; currently these are specifically the checks for the minimum multiplicity in references (an error is reported if the minimum multiplicity is one, but the reference is null or empty, respectively). You can integrate the generated constraints file into your workflow using the usual approach.

13.7. Naming

It is often the case the basically all model elements in a model should have a name. It might not always be necessary from a domain perspective, but it is really useful for debugging. Of course, you can add a superclass called `Named` with a single attribute `name` to all you classes. However, if you set the *`addNameAttribute`* parameter to be true when calling the uml2ecore cartridge, every class which does not inherit from another class gets an additional `name` attribute. Also, constraints that make sure that names within a namespace (i.e. a containment reference) are unique.

uml2ecore also comes with a utility extension called `org::openarchitectureware::util::stdlib::naming` that can calculate the `namespace()` and the `qualifiedName()` for every model element that has a name.

Another convenience is the parameter *`nameUnnamedNavigableAssociationEnds`*. Often you specify a relationship between two metaclasses and give the target end the name of the target metaclass. Let us say you specify an association from metaclass `Application` to `Component`. It is likely that the target end will be called `component`. By setting the parameter to true all unnamed navigable association ends will get the name of the target class. If there is more than one unnamed association only the first one will be modified. This would lead to a failing constraint afterwards, since at least after this modification all navigable association ends must be named.

13.8. Modularizing the metamodel file

You can modularize the metamodel. If you specify the *`resourcePerToplevelPackage=â##trueâ##`* parameter to the cartridge call, you will get a separate Ecore file for each top level package, as well as a separate constraint check file.

Chapter 14. "Classic" Reference

14.1. Available UML Tool Adapters

Table 14.1. Available UML tool adapters

UML Tool	Adapter Class [a]	Mapping File
ArgoUML	`argouml.ArgoUMLAdapter`	`argouml_xmi12_all.xml`
ARIS	`aris.ARISAdapter`	`aris_xmi11_all.xml`
Artisan	`artisan.ArtisanAdapter`	`artisan_xmi13_all.xml`

Enterprise Architect
 	
2.5
 	
4.x
 	
5.x
	`ea.EAAdapter`	`ea25_xmi11_all.xml`
		
		`ea41_xmi12_all.xml`
		
		`ea5_xmi11_all.xml`

Enterprise Architect 6.1	`EA61UnisysXmi12Adapter`, `EA61Xmi12Adapter`	`ea61_unisys_xmi12_cls.xml`, `ea61_xmi12_cls.xml`
MID Innovator	`innovator.InnovatorAdapter`	`innovator_xmi11_all.xml`
MagicDraw 8 -9	`magicdraw.MagicDrawAdapter12`	`magicdraw_xmi12_all.xml`
MagicDraw 10 - 12	`magicdraw_MagicDrawAdapter21`	`magicdraw_xmi21_all.xml`
Metamill	`metamill.MetamillAdapter`	`metamill31_xmi12_all.xml`
Poseidon 1.6	`poseidon.PoseidonAdapter`	`poseidon16_xmi12_all.xml`
Poseidon 2.x	`poseidon.PoseidonAdapter`	`poseidon20_xmi12_all.xml`
Poseidon 3.0	`poseidon.PoseidonAdapter`	`poseidon30_xmi12_all.xml`
Poseidon 3.1	`poseidon.PoseidonAdapter`	`poseidon31_xmi12_all.xml`
Poseidon 4	`poseidon.PoseidonAdapter`	`poseidon40_xmi12_all.xml`
Poseidon 5	`poseidon.PoseidonAdapter`	`poseidon40_xmi12_all.xml`
Rational Rose/Unisys Plugin 1.3.2 XMI 1.0	`rose.RoseAdapter`	`rose_unisys132_xmi10_all.xml`
Rational Rose/Unisys Plugin 1.3.2 XMI 1.1	`rose.RoseAdapter`	`rose_unisys132_xmi11_all.xml`
Rational Rose/Unisys Plugin 1.3.4/1.3.6 XMI 1.1	`rose.RoseAdapter`	`rose_unisys134_xmi11_all.xml`
Star UML	`staruml.StarUMLAdapter`	`staruml_xmi11_all.xml`
Together	`together.TogetherAdapter`	`together_xmi11_all.xml`
XDE	`xde.XdeAdapter`	`xde2_xmi11_all.xml`

[a] Package Prefix: org.openarchitectureware.core.frontends.xmi.toolsupport.uml

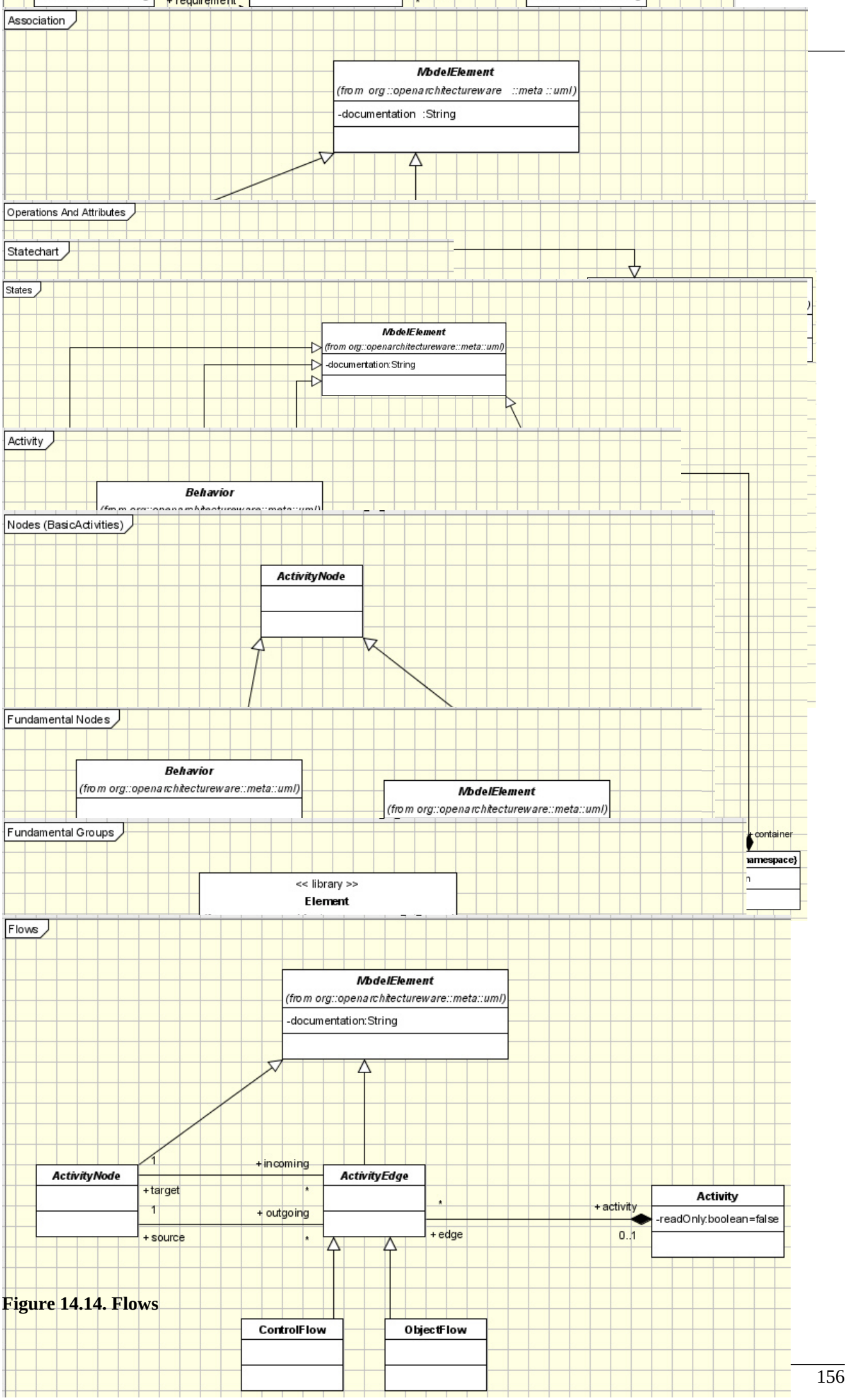


Figure 14.14. Flows

Chapter 15. Classic UML Tutorial

15.1. Introduction

This example shows the usage of openArchitectureWare 4 with integration of an UML tool. In openArchitectureWare 4, we call this "Classic" style, as the underlying metamodel has to be the "Classic" UML metamodel that was introduced by oAW 3. For the example, Magic Draw 11.5 Community Edition is used, but the example can easily be adapted to any supported UML tool. It is strongly recommended to work through this tutorial with MagicDraw in order to minimize environmental problems!

15.2. Installing the sample

Make sure that you installed the openArchitectureWare feature properly in your Eclipse environment.

openArchitectureWare depends on EMF, so check that you have installed it. If you need further information on oAW installation please look at http://www.openarchitectureware.org/staticpages/index.php/documentation.

Instead of working through this tutorial, you can also install the packaged example by downloading the `oaw-samples-classic-uml-4.x.x` package. It contains one Eclipse project, which you have to import into your workspace. To make the projects compile and run, you may have to define to use the *oAW-Classic Metamodel* in the project properties:

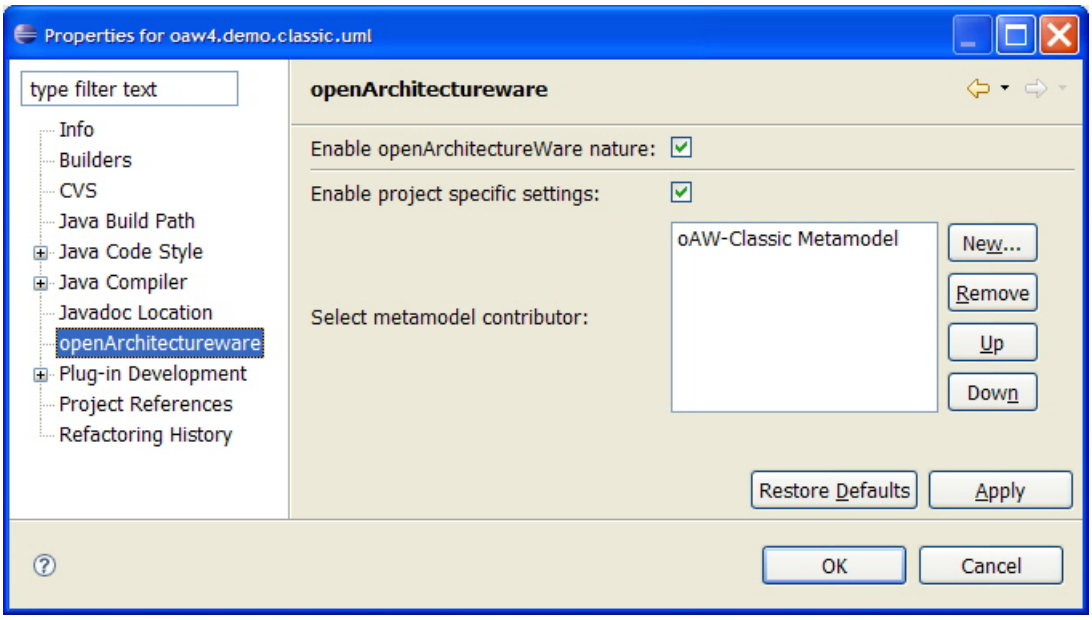


Figure 15.1. Configure oAW-Classic Metamodel

15.3. Example overview

The purpose of this tutorial is to demonstrate the very simplest way to use oAW4 in combination with an (non EMF UML2 capable) UML tool to create code from a model that contains some classes. The project is really simple, so it is the right place to start when you are new to openArchitectureWare 4 and want to use UML tools like MagicDraw, Poseidon, Rational Rose etc.

In this example, we want to generate code from this model:

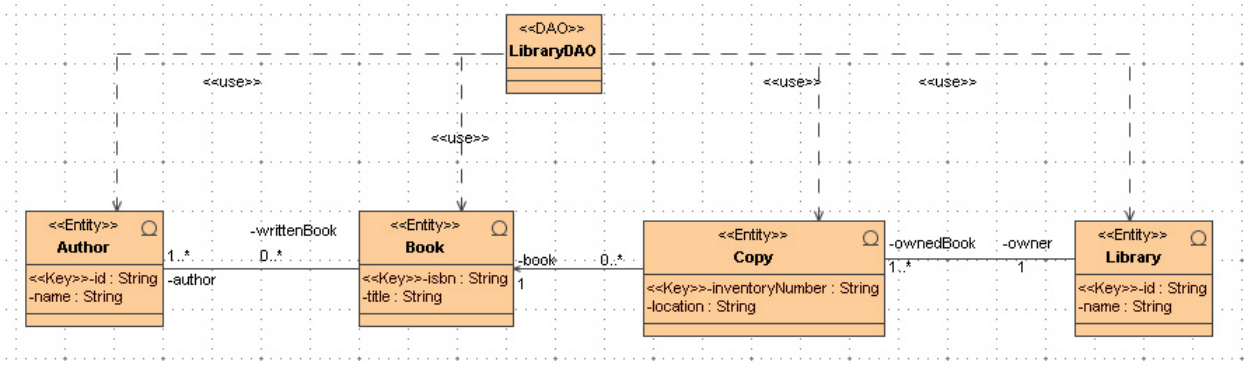


Figure 15.2. Example Model

As a result, we want to create some JavaBean style classes which have properties with getter/setter methods.

15.4. Setting up the project

Create a new Java Project called `oaw4.demo.classic.uml` and select the option to create seperate source and output folders.

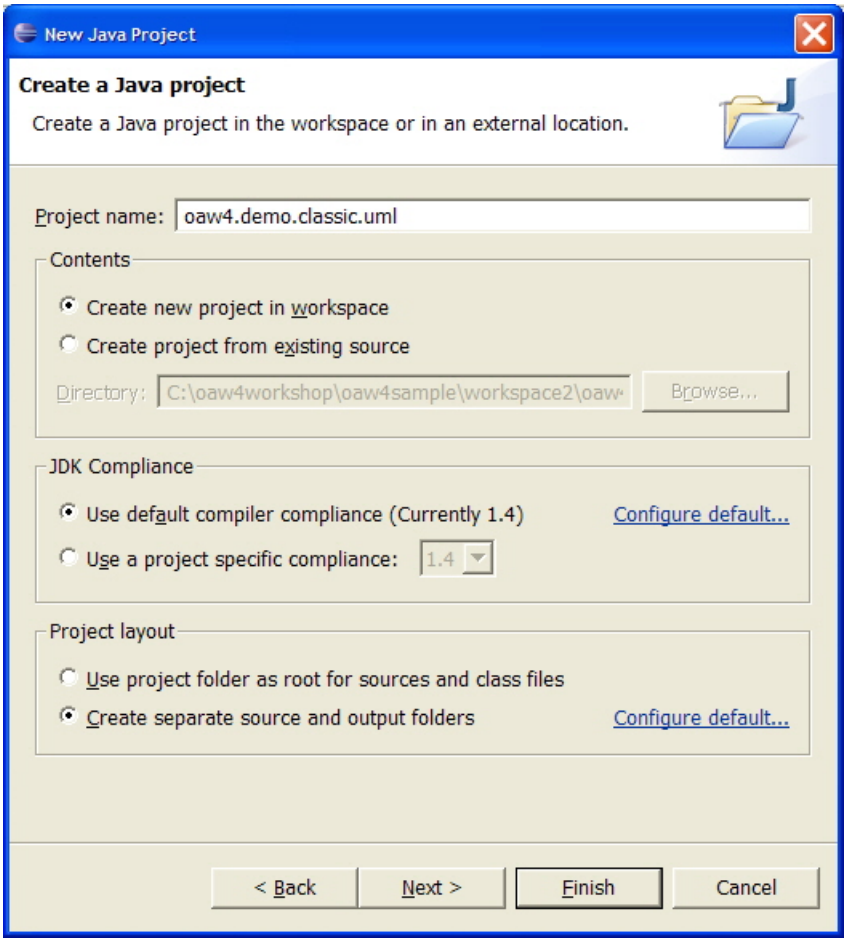


Figure 15.3. Creating the tutorial project

Afterwards, select from the context menu PDE tools -> Convert Projects to Plug-in Projects, since we want to define our dependencies via Eclipse Plug-In dependencies.

Alternatively, you could create a Plug-In project instead of these both steps.

15.5. Defining Dependencies

In the new project, there is now a `META-INF/MANIFEST.MF`. Open it, go to the Dependencies page and add the following dependencies:

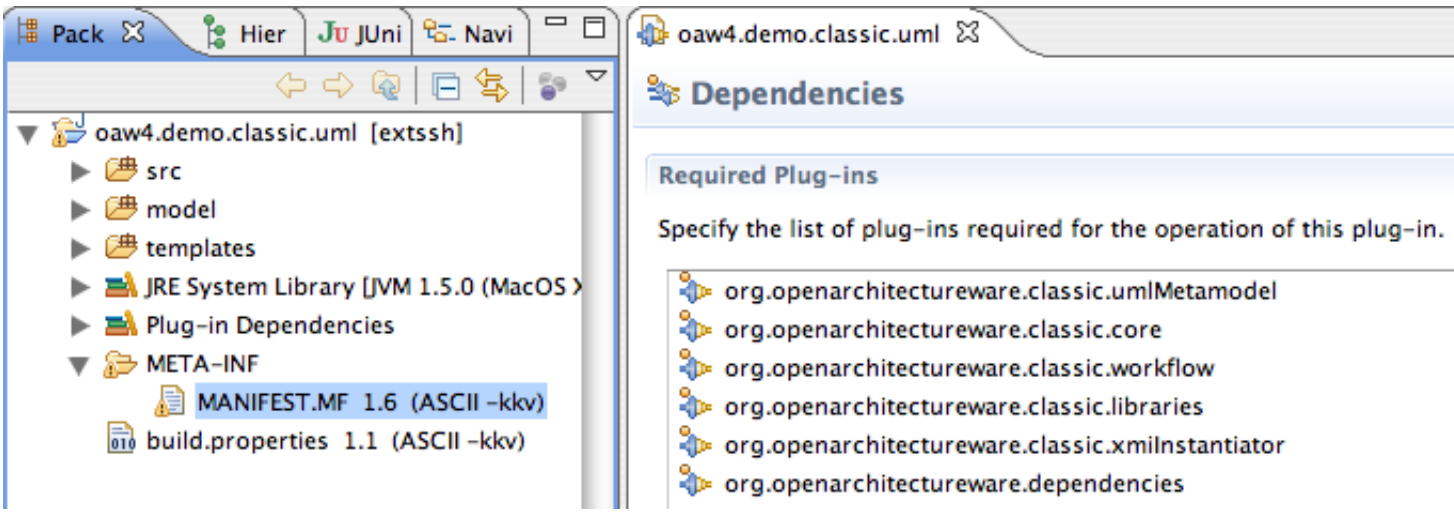


Figure 15.4. Defining plug-in dependencies for oAW Classic

- `org.openarchitectureware.classic.umlMetamodel` : The classic UML metamodel classes
- `org.openarchitectureware.classic.core` : Framework classes for oAW classic
- `org.openarchitectureware.classic.workflow` : oAW classic workflow components and cartridges
- `org.openarchitectureware.classic.xmiInstantiator` : Parser component for UML tools
- `org.openarchitectureware.classic.libraries` : Required 3rd party libraries
- `org.openarchitectureware.core.xpand2` : The Xpand template engine
- `org.openarchitectureware.core.expressions` : oAW language Check for defining constraints

15.6. Create source folders

Create two folders `model` and `templates` in the project root (not in `src`!) and add this folders as classpath folders in the project properties dialog, Libraries tab. By doing this, the model and the templates can be found in the classpath of the project without placing the files in the source folder.

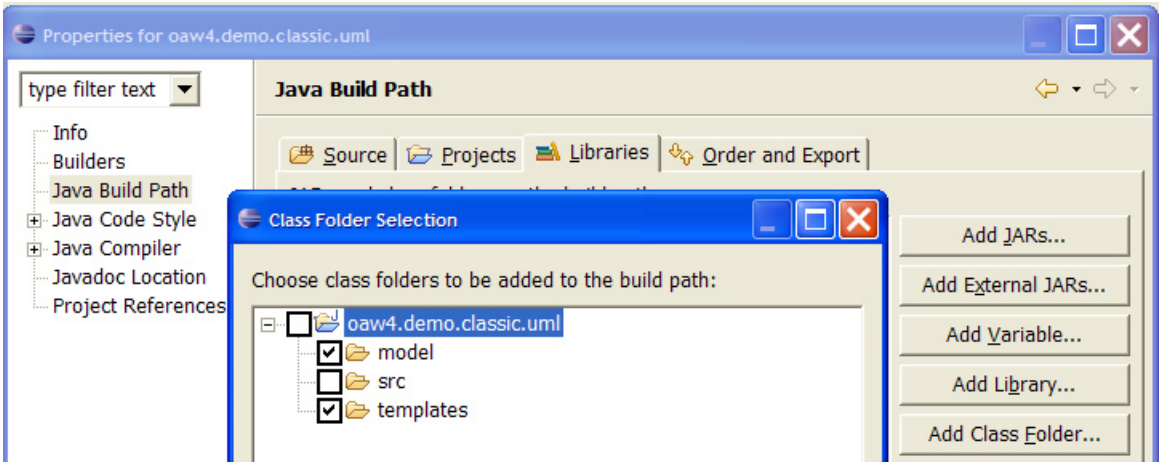


Figure 15.5. Creating source folders

15.7. Create the model

This section explains how to create the model from scratch using MagicDraw 10/11. If you are using another UML tool, this is a guideline to create the model, but the tool can differ in some details. You can skip this section if you have MagicDraw, downloaded the sample project and use the model from the sample project.

Start your UML Tool and create the model from the screenshot above. You have to create the Stereotypes `Entity` and `Key`.

To define the model above follow these steps:

1. Create the stereotype `Entity` with base class `Class` .
2. Create the stereotype `Key` with base class `Property` .
3. Create the stereotype `DAO` with base class `Class` .
4. Create a package structure in your project: `oaw4 / demo / classic / uml / entity`
5. Create a class diagramm in the package `entity`
6. Create a class `Author`
   a. Assign the stereotype `Entity`
   b. Create an attribute `id` of type `String`. Please select for String the datatype from your model, *not from UML Standard Profile*! Assign the stereotype `Key` for this attribute.
   c. Create an attribute `name` of type `String`.
7. Create the class `Book` with stereotype `Entity` . The `Key` attribute is `isbn` of type `String` . The second attribute is the `title` .
8. Draw an association between these two classes.
   a. The association end at class `Author` is named `author`, is navigable and the multiplicity is *1..**.

b. The other association end is `writtenBook` with multiplicity *0..** and is navigable.

9. Create the class `Copy` of stereotype `Entity`

   a. The `Key` attribute is `inventoryNumber` of type `String`.

   b. Add attribute `location` of type `String`.

10. Draw a *directed* association from `Copy` to `Book`

    a. The association end at class `Book` is named `book`, is navigable, with multplicity *1*.

    b. The opposite end at `Copy` is unnamed, not navigable and multiplicity *0..**

11. Create the class `Library` of stereotype `Entity`

    a. The `Key` attribute is `id` of type `String`.

    b. The library has a `String` attribute `name`

12. Draw an association from `Library` to `Copy`

    a. The association end at class `Library` is named `owner`, is navigable, the multiplicity is *1*.

    b. The association end at class `Book` is named `ownedBook`, is navigable and the multiplicity is *1..**.

Save your model packed format in the model folder of your project and give the file the name `AuthorBookExampleMD11.mdzip`. The tool adapter will automatically recognize that it is zipped and read the appropriate ZIP entry.

Create a subfolder `model/md11`. From the `profiles` directory of your MagicDraw installation copy the `UML_Standard_Profile.xml` to there.

15.8. Defining the metamodel

Models are instances of a *Metamodel*. In order to get openArchitectureWare to do something useful it needs to know the used metamodel. Using oAW "Classic" the metamodel is implemented by metaclasses. In UML, they are represented in the model by stereotypes.

The recently defined model already uses the stereotypes `DAO`, `Entity` and `Key`. Entities are some kind of business objects, which have some attributes. They are represented in UML as classes. Exactly one attribute is a special one: a `Key` attribute. `DAOs` are classes which manage `Entities`. We want to express this relationship by using a dependency from `DAO` to `Entity` in our model.

In UML, this metamodel looks like this:

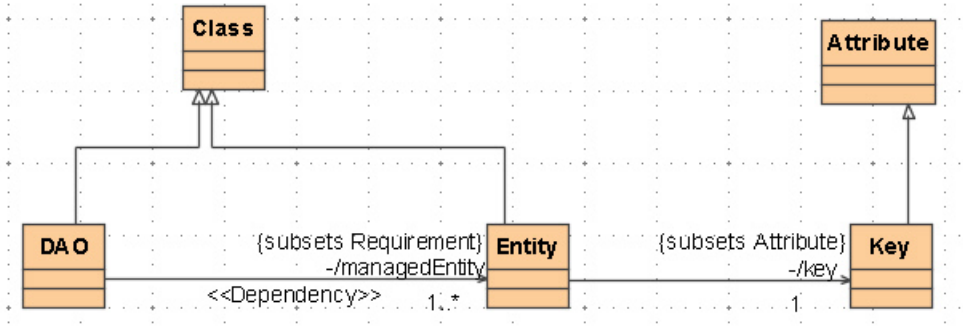


Figure 15.6. Tutorial Metamodel

We have to provide the metaclasses that make up our DSL. The base metaclasses `Class` and `Attribute` are provided by openArchitectureWare within the package `org.openarchitectureware.meta.uml` and its subpackages.

15.8.1. Defining the metaclasses

Create a package `oaw4.demo.classic.uml.meta`. This package will contain our metaclasses which realize the UML profile we want to use. As you can see from the model above we use the Stereotypes `Entity`, `Key` and `DAO`. For these Stereotypes we need metaclasses.

Metaclasses are derived from UML metaclasses. oAW4 provides implementation classes for UML metaclasses. For example, an `Entity` is represented by classes, so, the right metaclass to extend is `Class`, while `Keys` are a specialization from `Attribute`.

In the simplest case, you will only have to create the classes `Entity`, `Key` and `DAO` and derive them from the base metaclasses `Class` and `Attribute` defined in package `org.openarchitecture.meta.uml.classifier`:

```
package oaw4.demo.classic.uml.meta;

public class Entity extends org.openarchitectureware.meta.uml.classifier.Class {
  // nothing to do in the simplest case
}
```

```
package oaw4.demo.classic.uml.meta;

public class Key extends org.openarchitectureware.meta.uml.classifier.Attribute {
}
```

```
package oaw4.demo.classic.uml.meta;

public class DAO extends org.openarchitectureware.meta.uml.classifier.Class {
}
```

15.8.2. Metamappings

By now, the generator will not know that the stereotype `<<Entity>>` has to be mapped to the metaclass `oaw4.demo.classic.uml.meta.Entity`. By default these elements are mapped to their UML base classes, in case of `Entity` this is `Class`.

To map stereotypes to metaclasses a xml file has to be created, called the *metamapping file*.

Create the file `metamappings.xml` in your source folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MetaMap SYSTEM "http://www.openarchitectureware.org/dtds/metamap.dtd">

<MetaMap>
  <Mapping>
   <Map>Entity</Map>
   <To>oaw4.demo.classic.uml.meta.Entity</To>
  </Mapping>
  <Mapping>
   <Map>DAO</Map>
   <To>oaw4.demo.classic.uml.meta.DAO</To>
  </Mapping>
  <Mapping>
   <Map>Key</Map>
   <To>oaw4.demo.classic.uml.meta.Key</To>
  </Mapping>
</MetaMap>
```

15.9. Log4j configuration

Later when running the generator you would like to see some output messages. Therefore you have to define a `log4j.properties`. Create this file in the source folder:

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

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r %-5p %m%n
```

15.10. Creating the generator workflow and the first template

15.10.1. The workflow script

The next step is to create a workflow script. The workflow has to accomplish the following tasks:

- Initialize the environment for using the "classic" metamodel
- Parse the tool-specific xmi file and instantiate the metamodel
- Initialize all instantiated elements
- Run the generator to create code
- Tear-down: Print messages collected through the generator run

For all these tasks, pre-defined workflow-scripts and components can be included in the workflow. The following workflow script does all these tasks and should fit for the first steps. Later on, you may need to customize the workflow script to integrate further components like model modifiers. Therefore, the script itself is not bundled as a cartridge.

```
<?xml version="1.0" encoding="UTF-8"?>
<workflow>
  <property file="workflow.properties"/>

  <cartridge file="org/openarchitectureware/workflow/oawclassic/classicstart.oaw">
   <metaEnvironmentSlot value="me"/>
   <instantiatorEnvironmentSlot value="ie"/>
  </cartridge>

  <component class="org.openarchitectureware.core.frontends.xmi.workflow.XMIInstantiator">
   <instantiatorEnvironmentSlot value="ie"/>
   <modelFile value="${model.xmi}"/>
   <xmlMapFile value="${toolMappingFile}"/>
   <metaMapFile value="${metaMapFile}"/>
   <toolAdapterClassname value="${toolAdapterClassname}"/>
   <moduleFile value="${moduleFile}"/>
   <outputSlot value="model"/>
  </component>

  <cartridge file="org/openarchitectureware/workflow/oawclassic/classicinit.oaw">
   <metaEnvironmentSlot value="me"/>
  </cartridge>

  <component id="dirCleaner"
   class="org.openarchitectureware.workflow.common.DirectoryCleaner">
   <directories value="${srcGenPath}"/>
  </component>

  <component id="generator" class="org.openarchitectureware.xpand2.Generator">
   <metaModel class="org.openarchitectureware.type.impl.java.JavaMetaModel">
     <typeStrategy
      class="org.openarchitectureware.type.impl.oawclassic.OAWClassicStrategy"
      convertPropertiesToLowerCase="false"/>
   </metaModel>
   <expand value="Root::Root FOREACH me.getElements('Model')"/>
   <genPath value="${srcGenPath}/"/>
   <srcPath value="${srcGenPath}/"/>
   <beautifier class="org.openarchitectureware.xpand2.output.JavaBeautifier"/>
   <beautifier class="org.openarchitectureware.xpand2.output.XmlBeautifier"/>
   <fileEncoding value="ISO-8859-1"/>
  </component>

  <cartridge file="org/openarchitectureware/workflow/oawclassic/classicfinish.oaw">
   <instantiatorEnvironmentSlot value="ie"/>
   <dumpfile value="${dumpfile}"/>
  </cartridge>

</workflow>
```

15.10.2. Workflow Properties

The workflow includes a properties file, in which the concrete configuration is stored. Create the file `workflow.properties` like this:

```
# Note: all paths must be found in the classpath!
# the metamappings file
metaMapFile = metamappings.xml

# model.xmi: name of the XMI export
# toolMappingFile: tool mapping file to use
# toolAdapterClassname: tool adapter implementation
# moduleFile: profile files

# MagicDraw 10
model.xmi = AuthorBookExampleMD10.mdzip
toolMappingFile = magicdraw_xmi21_all.xml
toolAdapterClassname =
  org.openarchitectureware.core.frontends.xmi.toolsupport.uml.magicdraw.MagicDrawAdapter21
moduleFile =
  magicdraw/md10/UML_Standard_Profile.xml

#model.xmi = AuthorBookExampleMD11.mdzip
#toolMappingFile = magicdraw_xmi21_all.xml
#toolAdapterClassname =
  org.openarchitectureware.core.frontends.xmi.toolsupport.uml.magicdraw.MagicDrawAdapter21
#moduleFile = magicdraw/md11/UML_Standard_Profile.xml

# path to create the generated output to
srcGenPath = src-gen
# path where the dump file is created
dumpfile = bin/dump
```

15.10.2.1. Setting up for use with other UML tools

This configuration file is designed for use with MagicDraw. Other UML tools can be easily configured by changing the properties *`model.xmi`*, *`toolAdapterClassname`*, *`toolMappingFile`* and *`moduleFile`*. The property *`moduleFile`* specifies additional modules to load and merge, which is currently only evaluated by the MagicDraw adapter.

The existing tool adapter classes and mapping files can be found beneath the package

`org/openarchitectureware/core/frontends/xmi/toolsupport/uml/<TOOL>`

For example, to set up this project for Poseidon 4/5 the appropriate settings are:

```
toolAdapterClassname =
  org.openarchitectureware.core.frontends.xmi.toolsupport.uml.poseidon.PoseidonAdapter
toolMappingFile = poseidon40_xmi12_all.xml
```

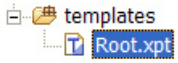
15.10.3. Create the root template

In the workflow the generator has been configured to start with the this definition:

```
<expand value="Root::Root FOREACH me.getElements('Model')"/>
```

This means that the generator will look for a template file `Root.xpt` in the classpath where a definition named `Root` is found for all elements that are selected by the expression `me.getElements('Model')`. In case of UML models there exists solely one element of type `Model` that will be selected.

Now, create the file Root.xpt in the templates folder. Remember: We have configured our Eclipse project that the folder templates is a classpath folder. You will see that the new file is recognized as an oAW template file, as it has a template file icon:



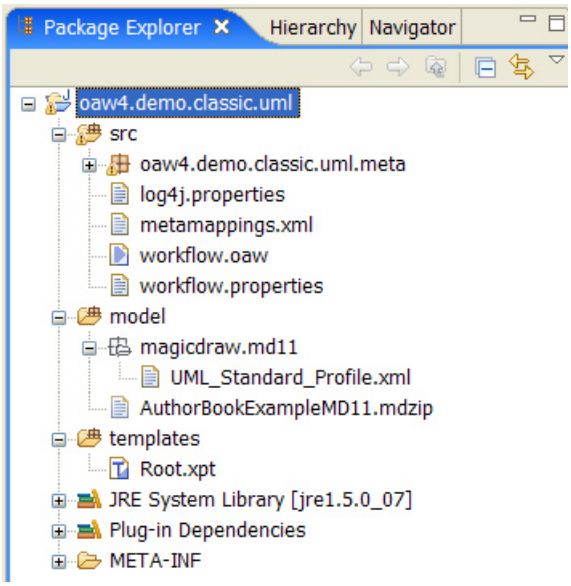
For the beginning the template will be simple:

```
«IMPORT org::openarchitectureware::core::meta::core»

«DEFINE Root FOR Model»
«ENDDEFINE»
```

This template contains only an empty definition for Model elements. The namespace of the `Model` metaclass must be known for the generator, so we import the corresponding package. Otherwise, we would have to fully qualify `Model`. As you could expect, this template does not really do anything yet.

At this time, your project structure should look like this:



15.11. Execute the workflow

We have not defined anything useful in our template yet. However, we now execute the workflow to prove that everything is right configured. Execute the workflow by selecting `workflow.oaw` and select *Run As -> oAW Workflow* from the context menu. The output should be like this:

```
0    INFO  ------------------------------------------------------------------------------
0    INFO  openArchitectureWare v4.3 -- (c) 2005, 2008 openarchitectureware.org and contributors
0    INFO  ------------------------------------------------------------------------------
0    INFO  running workflow:
       C:/dev/ide/workspace/oaw-v4-projects/oaw4.demo.classic.uml/src/workflow.oaw
0    INFO
540  INFO  Starting: workflow org/openarchitectureware/workflow/oawclassic/classicstart.oaw
540  INFO  Starting: org.openarchitectureware.workflow.oawclassic.ClassicOAWSetup
561  INFO  classic oAW environment is set up;
       instantiator environment in: ie, meta environment in me
561  INFO  Starting: org.openarchitectureware.core.frontends.xmi.workflow.XMIInstantiator
561  INFO  Loading XMI from: AuthorBookExampleMD10.mdzip using map: magicdraw_xmi21_all.xml
       and metamap: metamappings.xml
571  INFO  Starting mapping instantiator ...
571  INFO
581  INFO  Parsing metamap metamappings.xml ...
721  INFO  Parsing of metamap took 0.14s
721  INFO
1692 INFO  Parsing design AuthorBookExampleMD10.mdzip ...
1702 INFO  Loading model...
2163 INFO  Initializing XMI support for XMI version 2.1
2163 INFO  Scanning for referenced modules...
2183 INFO  Found reference to module 'UML_Standard_Profile.xml'
2193 INFO  Model loaded in 0.491s
2193 INFO
2193 INFO  Loading modules...
2984 INFO  Scanning for referenced modules...
3064 INFO  Modules loaded in 0.871s
3064 INFO
3074 INFO  Merging...
3114 INFO  Modules merged in 0.04s
3114 INFO
3234 INFO  Parsed design in 1.542s
3234 INFO
3234 INFO  Loading extensions...
3304 INFO  Extensions loaded in 0.07s
3304 INFO
3304 INFO  Loading design...
3314 WARN  ignoring node 65973 (ownedMember): cannot coerce to ModelElement
3314 WARN  ignoring node 66017 (ownedMember): cannot coerce to ModelElement
3314 WARN  ignoring node 66061 (ownedMember): cannot coerce to ModelElement
3314 WARN  ignoring node 66119 (ownedMember): cannot coerce to ModelElement
3314 WARN  ignoring node 68759 (ownedMember): cannot coerce to ModelElement
3314 WARN  ignoring node 70438 (ownedMember): cannot coerce to ModelElement
3324 WARN  ignoring node 65945 (ownedMember): cannot coerce to ModelElement
3334 WARN  ignoring node 65952 (ownedMember): cannot coerce to ModelElement
3334 WARN  ignoring node 65959 (ownedMember): cannot coerce to ModelElement
3334 WARN  ignoring node 65966 (ownedMember): cannot coerce to ModelElement
3405 INFO  Design loaded in 0.101s
3405 INFO
3405 INFO  Applying tool specific design modifications...
3415 INFO  Design modified in 0.01s
3415 INFO
3415 INFO  Instantiating metamodel...
3485 WARN  <<Model>> [no Name@27940994]: ignoring value for nonexisting property Documentation =
[Author:Karsten.
Created:27.02.06 08:47.
Title:.
Comment:.
]
3525 INFO  Metamodel instantiated in 0.1s
3525 INFO
3525 INFO  Instantiated design in 2.954s
3525 INFO
3525 INFO  MetaModel Summary
3525 INFO  -----------------
3525 INFO
3525 INFO  3x Association (org.openarchitectureware.meta.uml.classifier.Association)
3525 INFO  6x AssociationEnd (org.openarchitectureware.meta.uml.classifier.AssociationEnd)
3525 INFO  4x Attribute (org.openarchitectureware.meta.uml.classifier.Attribute)
3525 INFO  1x DAO (oaw4.demo.classic.uml.meta.DAO)
```

Congratulations! You have just set up the whole environment to get openArchitectureWare running. Now, let us do the interesting stuff!

15.12. Looping through the model

Later on, we want to do something with the elements contained in our model. A common pattern in the Root template when using UML models is to loop through the model to find the elements that should be expanded. A model is an instance of `Namespace` and all elements directly contained by the model can be accessed through the association `OwnedElement`. The owned elements can be of various types: Classes, Packages, Datatypes and so on. Packages are also instances of `Namespace`, so they also have a `OwnedElement` association.

We use the feature to recursively resolve any element contained in the model tree. Now extend the `Root.xpt` template file:

```
«IMPORT org::openarchitectureware::core::meta::core»
«IMPORT org::openarchitectureware::meta::uml::classifier»
«DEFINE Root FOR Model»
  «EXPAND Root FOREACH OwnedElement»
«ENDDEFINE»

«DEFINE Root FOR Package»
  «EXPAND Root FOREACH OwnedElement»
«ENDDEFINE»

«DEFINE Root FOR Object»«ENDDEFINE»
```

Note, that we have added a new `IMPORT` statement, because the metaclass `Package` is in package `org.openarchitectureware.meta.uml.classifier`.

First, the definition `Root` FOR `Model` will be called by the generator. This will call the definition named `Root` for all elements, that the `Model` instance contains. Look at the last definition: This is a catcher for all elements that are found while traversing through the tree that are not of any handled type.

When the model contains `Package` instances the definition `Root FOR Package` will be called for these instances. This is polymorphism at work! When the package contains subpackages, the definition is called recursively.

15.13. Creating a template for JavaBeans

As we want to create JavaBean style classes from the entities in our model, we now want to create the template file for this. Create a new folder java within the folder `templates` create the file `JavaBean.xpt` there.

For a first step, this template will simply create a file with the name of the class.

```
«IMPORT org::openarchitectureware::meta::uml::classifier»

«DEFINE BeanClass FOR Class»
  «FILE NameS+".java"»
   public class «Name» {
   }
  «ENDFILE»
«ENDDEFINE»
```

Note, that we have named the definition `BeanClass` and it is defined for elements of type `Class`. So, this template will not only fit for elements of type `Entity`, but can be used for any class.

In the FILE statement, we access the name of the current element by property NameS. This is specific to the oAW classic metamodel. The property NameS returns the name of the element as String. The property Name itself is of type Object for backward compatibility. But in templates we usually want the name as a String object.

15.14. Calling the JavaBean template from Root.xpt

By now, this template is not called. We have to extend the Root.xpt template file. Add a definition Root defined for Entity. We also have to add a new IMPORT, so that oAW can resolve Entity.

```
«IMPORT oaw4::demo::classic::uml::meta»

...

«DEFINE Root FOR Entity»
«EXPAND java::JavaBean::BeanClass»
«ENDDEFINE»
```

As you remember, while looping through the model for each element in the model tree a definition called Root is called. Before adding this, we had a definition for Packages, Models, and all other objects. Now, when evaluating elements of type Entity this definition matches. In this definition, we call the definition named BeanClass from the template file JavaBean.xpt. As the template JavaBean.xpt is defined in the package java, we also have to qualifiy this namespace. As an alternative, we could import the namespace java. The definition is (implicitely) called for the current element, which is of course an Entity.

15.15. The first generated code

Run the generator again (as a shortcut you could type **Ctrl**+**F11**). Now, your project should contain the folder src-gen, which contains four files: Author.java, Book.java, Copy.java and Library.java.

Open Author.java. It does only contain the class definition, since the template was that simple.

```
public class Author {
}
```

15.16. Defining property declarations and accessor methods

Now, we want to extend the JavaBean template to create instance variables, property getters and setters. Generating declarations for instance variables as well as their getter and setter methods is a very recurring task for attributes, so we want this in a central template file.

Create the file Attribute.xpt in folder java.

```
«IMPORT org::openarchitectureware::meta::uml::classifier»

«DEFINE PropertyDeclaration FOR Attribute»
private «Type.NameS» «NameS»;
«ENDDEFINE»

«DEFINE Getter FOR Attribute»
public «Type.NameS» get«NameS.toFirstUpper()» () {
  return this.«NameS»;
}
«ENDDEFINE»

«DEFINE Setter FOR Attribute»
public void set«NameS.toFirstUpper()» («Type.NameS» «NameS») {
  this.«NameS» = «NameS»;
}
«ENDDEFINE»
```

Of course, we also have to call these templates from our `JavaBean` template. We want to call the templates `PropertyDeclaration`, `Getter` and `Setter` for each attribute a class has. So, open your `JavaBean.xpt` template and extend it like follows:

```
«IMPORT org::openarchitectureware::meta::uml::classifier»

«DEFINE BeanClass FOR Class»
«FILE NameS+".java"»
public class «Name» {
  «EXPAND Attribute::PropertyDeclaration FOREACH Attribute»
  «EXPAND Attribute::Getter FOREACH Attribute»
  «EXPAND Attribute::Setter FOREACH Attribute»
}
«ENDFILE»
«ENDDEFINE»
```

Once again, run the generator. Now, your generated files have properties!

```
public class Author {
  private String id;
  private String name;

  public String getId() {
   return this.id;
  }

  public String getName() {
   return this.name;
  }

  public void setId(String id) {
   this.id = id;
  }

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

15.17. Using Extensions

A very powerful feature of openArchitectureWare 4 are extensions. With extensions the *Xpand* template engine can be extended with functions without the need to modify the metamodel.[1] A very common use of extensions are the use of naming conventions, navigation, computation of package, path and filenames for artifacts etc.

openArchitectureWare extensions are declared in files ending with `.ext`. The declaration of extension functions can be by means of oAW expressions or by calling Java functions. The latter have to be declared `public` and `static`.

We will not cover the syntax of expressions very deep, so if you are interested to get more information look at the reference core reference chapters *Xtend* and *Expressions*. Also, in the other tutorials you can see more examples for the usage of expressions.

15.17.1. Declaring functions with the *Xtend* language

For our example, we want to use extensions to introduce some naming conventions for instance variables, parameters, and the computation of the package and path for classes.

Now, create a file `NamingConventions.ext` in the `templates/java` folder. In this extension we declare some functions that are helpful for name conversions:[2]

```
import org::openarchitectureware::meta::uml;
import org::openarchitectureware::meta::uml::classifier;

String asParameter (ModelElement elem) :
  "p"+elem.NameS.toFirstUpper();

String asSetter (ModelElement elem) :
  "set"+elem.NameS.toFirstUpper();

String asGetter (ModelElement elem) :
  "get"+elem.NameS.toFirstUpper();

String asInstanceVar (ModelElement elem) :
  elem.NameS.toFirstLower();
```

In extension files, import statements for used types are necessary, too. The function declarations are single-lined and can use any expressions, calls of other extension functions included.

Open `Attribute.xpt` and make use of these functions. In template files, the usage of extensions must be declared by the «`EXTENSION`» keyword. The modified template file will look like this:

[1] With openArchitectureWare3 it was needed to code functionality of the metamodel in metaclasses

[2] The functions in the Extension file could be defined for type `Attribute` instead of `ModelElement`, but we want to use these functions for other types later on, too.

```
«IMPORT org::openarchitectureware::meta::uml::classifier»
«EXTENSION java::NamingConventions»

«DEFINE PropertyDeclaration FOR Attribute»
  private «Type.NameS» «asInstanceVar()»;
«ENDDEFINE»

«DEFINE Getter FOR Attribute»
  public «Type.NameS» «asGetter()» () {
   return this.«asInstanceVar()»;
  }
«ENDDEFINE»

«DEFINE Setter FOR Attribute»
public void «asSetter()» («Type.NameS» «asParameter()») {
 this.«asInstanceVar()» = «asParameter()»;
}
«ENDDEFINE»
```

Run the generator and open `Author.java`. The file looks almost the same, since the replacements for instance variables, getter and setter method names are equivalent. The only difference are the parameters for setter methods. They are now prefixed with the character `p`.

```
public void setId(String pId) {
  this.id = pId;
}
```

15.17.2. Calling Java functions as extensions

More complex computations are often easier and more understandable by means of the Java programming language. So, the *Xtend* language allows to define functions by referencing a Java function. The methods that should be called, must be declared `public static`.

In our example, we miss the declaration of packages, and files are written into the root directory. We now want to declare some functions that help us to compute the full package name of classes, their corresponding path and the fully qualified name.

Create a new package `oaw4.demo.classic.uml.extend` and a class `ClassUtil`. In this class we define a function `getPackageName()` that computes the full package of a Class (remember that we operate on the class of the metamodel named `(org.openarchitectureware.meta.uml.classifier.)Class`, not `java.lang.Class`).

```
package oaw4.demo.classic.uml.extend;

import org.openarchitectureware.meta.uml.classifier.Class;
import org.openarchitectureware.meta.uml.classifier.Package;

public class ClassUtil {
  public static String getPackageName (Class cls) {
   String result = "";
   for (Package pck=cls.Package(); pck!=null; pck=pck.SuperPackage()) {
     result = pck.NameS() + (result.length()>0 ? "."+result : "");
   }
   return result;
  }
}
```

We declare this function in `NamingConventions.ext` and two more functions that use it. Add these function declarations:

```
String packageName (Class cls) :
  JAVA oaw4.demo.classic.uml.extend.ClassUtil
  .getPackageName(org.openarchitectureware.meta.uml.classifier.Class);

String packagePath (Class cls) :
  packageName(cls).replaceAll("\\.", "/");

String fqn (Class cls) :
  packageName(cls).length>0 ? packageName(cls)+"."+cls.NameS : cls.NameS;
```

The additional functions are used to compute the path to files based on a class and to get the full qualified name of a class. We want to make use of these functions to declare the package, compute the desired file path and to extend from the superclass, if a class has one (in our example model this is not the case yet). If no superclass exists, the Bean class should declare to implement the `Serializable` interface.

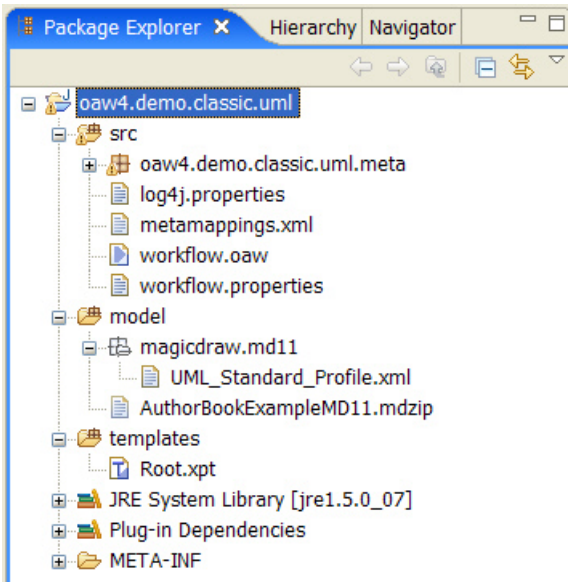
Open the `JavaBean.xpt` template file and modify it like follows:

```
«IMPORT org::openarchitectureware::meta::uml::classifier»
«IMPORT java»
«EXTENSION java::NamingConventions»

«DEFINE BeanClass FOR Class»
«FILE packagePath()+"/"+NameS+".java"»
package «packageName()»;
public class «Name»
  «IF hasSuperClass»extends «SuperClass.fqn()» «ENDIF»
  implements java.io.Serializable {

...
```

Once again, run the generator and refresh your `src-gen` folder. Now, the classes are generated in the expected directory structure:



The generated classes now contain the correct `package` statement:

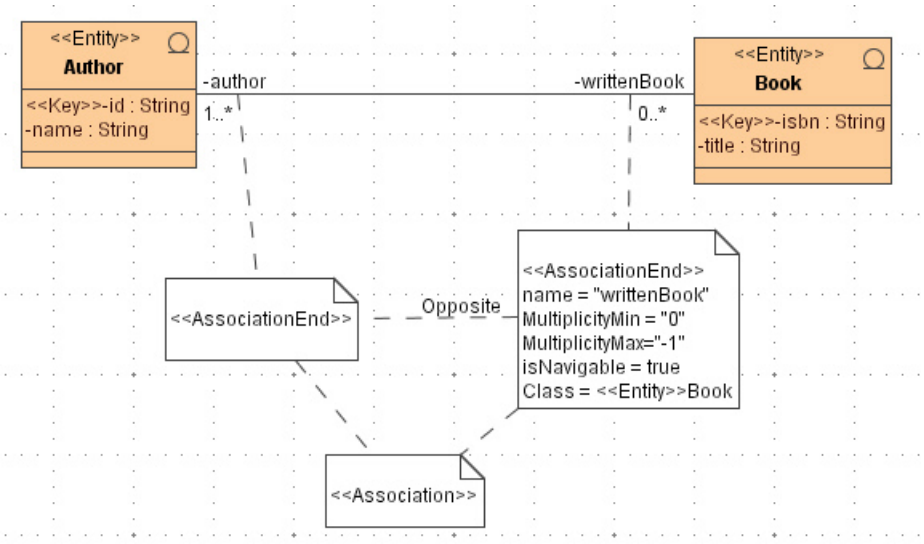
```
package oaw4.demo.classic.uml.entity;

public class Author implements java.io.Serializable {
  ...
```

15.18. Working with associations

In the model, we have defined some associations of different kind. In the next step, we evaluate these associations and create references, accessor methods and modification methods. As you can see, an m:n association between `Author` and `Book` exists. There also exists an association from `Library` to `Copy` with the multiplicity 1:n. The association from `Book` to `Copy` is not navigable. These associations must be handled differently.

Ordinary associations in UML consist of three elements: Two *association ends* and one *association* containing them. From the view of a class, a referenced class is at the *opposite* end. Normally it must be considered whether the association, i.e. the opposite association end, to the referenced class is navigable.



Also, the templates for associations between classes are very common, so we define them in a central template file `Association.xpt` in the `templates/java` folder.

We want to keep the usage of the association template for the calling class template simple and hide this complex stuff about association ends. So, we define simple entry points for `Class` elements. We also make use of further extensions to facilitate template development.

15.18.1. Association Extensions

The extensions for functions used for associations are put into a separate file. Create the file `templates/java/Associations.ext` with this content:

```
import org::openarchitectureware::meta::uml::classifier;
extension java::NamingConventions;

String fqn (AssociationEnd ae) :
  !ae.isMultiple ? ae.Class.fqn() : "java.util.Collection<"+ae.Class.fqn()+">";

String iterator (AssociationEnd ae) : "java.util.Iterator<"+ae.Class.fqn()+">";
```

As you can see, we will create Java 5 style collections for to-many associations. If you do not want to use generics, you could easily replace or modify the extension file and all associations will change to your style.

We see in this extension file, that other extensions can be referenced using the `extension` keyword. In this case, we need the function `fqn()` that was defined for classes. For associations, another function `fqn()` is defined, but

now for AssociationEnds. For to-one associations, the fully qualified name of the associated class is returned, for to-many references, we return a generic collection for that class. In the template code, classes and association ends will both use the fqn() function to print out the referenced type.

15.18.2. Writing a template for associations

Now, it is time to write the template for associations. Create the file Association.xpt. Now fill in the content. We will explain some statements right after.

```
«IMPORT org::openarchitectureware::meta::uml::classifier»
«EXTENSION java::NamingConventions»
«EXTENSION java::Associations»

«DEFINE ReferenceVariables FOR Class»
  «FOREACH AssociationEnd.Opposite.select(ae|ae.isNavigable) AS ae»
   private «ae.fqn()» «ae.asInstanceVar()»;
  «ENDFOREACH»
«ENDDEFINE»

«DEFINE AccessorMethods FOR Class»
  «EXPAND ToOneAccessorMethods FOREACH
   AssociationEnd.Opposite.select(ae|!ae.isMultiple && ae.isNavigable)»
  «EXPAND ToManyAccessorMethods FOREACH
   AssociationEnd.Opposite.select(ae|ae.isMultiple && ae.isNavigable)»
«ENDDEFINE»

«DEFINE ToOneAccessorMethods FOR AssociationEnd»
  public void «asSetter()» («Class.fqn()» «asParameter()») {
   this.«asInstanceVar()» = «asParameter()»;
  }

  public «Class.fqn()» «asGetter()» () {
   return this.«asInstanceVar()»;
  }
«ENDDEFINE»

«DEFINE ToManyAccessorMethods FOR AssociationEnd»
  public void add«NameS.toFirstUpper()» («Class.fqn()» «asParameter()») {
   this.«asInstanceVar()».add(«asParameter()»);
  }

  public void remove«NameS.toFirstUpper()» («Class.fqn()» «asParameter()») {
   this.«asInstanceVar()».remove(«asParameter()»);
  }

  public «iterator()» «asGetter()» () {
   return this.«asInstanceVar()».iterator();
  }
«ENDDEFINE»
```

At first, we see that this template uses both extensions, NamingConventions and Associations.

Next, a template ReferenceVariables is declared. The template makes use of a FOREACH loop of a different kind than we saw before. One more specific thing here is the statement

```
«FOREACH AssociationEnd.Opposite.select(ae|ae.isNavigable) ...
```

This returns all *navigable* opposite end of each association end a class has.

The body of the loop declares a reference variable for an association. For both alternatives (to-one and to-many), the function `fqn()` (the one defined for association ends) is called to determine the right type. In our example, this should result in the following both declarations:

```
// to-one association (Copy.java)
private oaw4.demo.classic.uml.entity.Library owner;

// to-many association (Author.java)
private java.util.Collection<oaw4.demo.classic.uml.entity.Book> writtenBook;
```

Both types are now expressed by this statement – simple, isn't it?

```
private «ae.fqn()» «ae.asInstanceVar()»;
```

The next template definition is also interesting. The definition `AccessorMethods FOR Class` dispatches to the definitions `ToOneAccessorMethods` or `ToManyAccessorMethods`, depending on the cardinality of the association. To distinguish both types, we make use of the select expression which is defined for expression. The statement

```
«EXPAND ToOneAccessorMethods FOREACH
  AssociationEnd.Opposite.select(ae|!ae.isMultiple && ae.isNavigable)»
```

means that the definition `ToOneAccessorMethods` should be expanded for each opposite association end which is not to-many and navigable. So for unnavigable associations no accessor method will be created.

The definitions for the accessor methods have nothing new, so we do not explain them in detail now.

15.18.3. Extending the JavaBeans template

Now, we want to use the new templates from the JavaBeans template so that the classes get instance variables for referenced classes and appropriate accessor methods. Modify the template file `JavaBeans.xpt` by adding these two statements to the class definition:

```
public class «Name» «IF hasSuperClass»extends «SuperClass.fqn()» «ENDIF»{
  «EXPAND Attribute::PropertyDeclaration FOREACH Attribute»
  «EXPAND Attribute::Getter FOREACH Attribute»
  «EXPAND Attribute::Setter FOREACH Attribute»
  «EXPAND Association::ReferenceVariables»
  «EXPAND Association::AccessorMethods»
}
```

15.18.4. Generator result

Run the generator again. After successful generation, open `Book.java`. This file should have the following contents:

```
package oaw4.demo.classic.uml.entity;

public class Book implements java.io.Serializable {
  private String isbn;
  private String title;
  private java.util.Collection<oaw4.demo.classic.uml.entity.Author> author;

  public String getIsbn () {
   return this.isbn;
  }

  public String getTitle () {
   return this.title;
  }

  public void setIsbn (String pIsbn) {
   this.isbn = pIsbn;
  }

  public void setTitle (String pTitle) {
   this.title = pTitle;
  }

  public void addAuthor (oaw4.demo.classic.uml.entity.Author pAuthor) {
   this.author.add(pAuthor);
  }

  public void removeAuthor (oaw4.demo.classic.uml.entity.Author pAuthor) {
   this.author.remove(pAuthor);
  }

  public java.util.Iterator<oaw4.demo.classic.uml.entity.Author> getAuthor () {
   return this.author.iterator();
  }
}
```

In this file, we have a to-many association named `author` to entity `Author`. No code exists for the association to the `Copy` class, since this association is not navigable.

An example for to-one associations can be seen in the `Copy` class:

```
public class Copy implements java.io.Serializable {
  ...
  private oaw4.demo.classic.uml.entity.Library owner;
  ...
  public void setOwner (oaw4.demo.classic.uml.entity.Library pOwner) {
   this.owner = pOwner;
  }

  public oaw4.demo.classic.uml.entity.Library getOwner () {
   return this.owner;
  }
  ...
}
```

The template code is really small, but now you could only have to model classes and their associations and you get the right JavaBeans code from the model. Our JavaBeans template is not specific for entities, you could use it to generate JavaBeans code for just any modelled class. But we only call this template for classes stereotyped with `<<Entity>>` for now. You remember that the model contains another class `<<DAO>> LibraryDAO` for which no code is generated yet.

15.19. Constraint Checking

It is very important that a generator produces correct output. Therefore, the input information must be valid, i.e. the model must be consistent. To prove this, the metamodel is enriched by constraints that check the consistence of the model.

15.19.1. Alternatives for implementing constraints

Constraints can be provided in three different ways:

- Overriding the `checkConstraints()` method of the metaclasses.

  The `checkConstraints()` method is only available for metaclasses based on the "classic" metamodel. It is not recommended to use this alternative. This alternative is for backward compatibility; in oAW3 metaclasses usually used this method to check model constraints.

- Using the new *Check* language.

  oAW 4 has a new language named *Check* that can be used to check model constraints. In this example, we will focus on this method. You may want to read the chapter *Check language* in the core reference for more information.

We want to implement constraint checks by using the *Check* language. The syntax for check files is similar to those for Extensions, as it uses also the oAW Expressions framework. Check files have the file extension `.chk` and have be on the classpath in order to be found.

15.19.2. Constraints to implement in the example

In our example, we want to implement the following constraint:

1. Each navigable association end must have a role name.

2. Unnavigable association ends should have no role name.

Constraint 1 must be fulfilled, otherwise an error message should be printed and the generation process should not be started. If constraint 2 is not fulfilled a warn message should be printed, but the generation process should not be stopped.

15.19.3. Creating the Check file

As the constraints that should be implemented are specific for associations, the check file should be named `AssociationChecks.chk`. Create this file in `templates/java`.

```
import org::openarchitectureware::meta::uml::classifier;

context AssociationEnd ERROR Class.NameS+"->"+Opposite.Class.NameS+":
  Navigable association ends must have a role name" :
  isNavigable ? !isUnnamed : true;

context AssociationEnd WARNING Class.NameS+"->"+Opposite.Class.NameS+":
  Not navigable association ends should have no role name":
  isNavigable ? true : isUnnamed;
```

The syntax is rather simple. Both constraints should be checked for the metaclass `AssociationEnd`, so the appropriate namespace has to be imported. The expression following the colon is the constraint. If it is not fulfilled, the message is printed. When error messages are printed, the workflow will be interrupted.

15.19.4. Checking the model

To execute the constraint checks, the generator workflow has to be extended. The constraint check is configured by a workflow component `CheckComponent`. Insert this code into `workflow.oaw`, right between the cartridge `classicinit.oaw` and the `dirCleaner` component.

```
<component class="org.openarchitectureware.check.CheckComponent">
  <metaModel class="org.openarchitectureware.type.impl.java.JavaMetaModel">
   <typeStrategy class="org.openarchitectureware.type.impl.oawclassic.OAWClassicStrategy"
    convertPropertiesToLowerCase="false"/>
  </metaModel>
  <checkFile value="java::AssociationChecks"/>
  <expression value="me.getElements('ModelElement')"/>
  <abortOnError value="true"/>
</component>
```

Also, the constraint checker has to know which metamodel it should use. It is the same configuration as for the Generator component.[3]The `checkFile` property specifies the qualified name of the check file.

Finally, the `expression` property defines for which elements to constraints should be applied. In our case, we select *all* elements in the `MetaEnvironment`. However, for our special case it would be satisfactory to select only all elements of type `AssociationEnd`. For larger projects, it could be of advantage to select the right subset of elements for improving performance.

15.19.5. Testing the constraints

Start the generator. The generator should run without complaining, because all constraints are fulfilled in the example model for now.

Now open the model; we will change the model in a way that both constraints are not fulfilled.

First, edit the association between `Book` and `Author`. Take the association end named `author` and remove the name. Set the `navigable` flag to `false` for the opposite end named `writtenBooks`.

Save the model and re-run the generator. The generator now produces the expected messages and stops execution, because there is one constraint error and one warning.

```
4316 INFO  Starting: org.openarchitectureware.check.CheckComponent
4667 ERROR Workflow interrupted. Reason: Errors during validation.
4667 WARN  Book->Author: Not navigable association ends should have no role
      name [<<AssociationEnd>> writtenBook]
4667 ERROR Author->Book: Navigable association ends must have a role name [<<AssociationEnd>> ]
```

15.20. Further development of this tutorial

This tutorial is not finished. I plan to extend it in order to show more basic things about oAW usage. If you have any suggestions let me know them. Some features that will be handled in the future are:

[3]It is possible to declare the used metamodel once and reference it the second time.

- Extending the metamodel manually
- Usage of MMUtil functions in metamodel classes
- Generating the metaclasses with the Metamodel Generator
- Demonstration of some oAW expression language features, e.g. collection filter, list operations

Part III. Samples

Chapter 16. Using the Emfatic Ecore Editor

16.1. Introduction

Metamodelling based on the EMF tree views is not a very good solution. A better solution is a textual representation for Ecore models. The IBM Emfatic plugin provides this support.

16.2. Installation

Install the Emfatic plugins from http://www.alphaworks.ibm.com/tech/emfatic.

16.3. Working with Emfatic

If you already have an ecore file (such as our `data.ecore` in our example), you can right-click on the file and select the *Generate Emfatic Source* item. The following code is generated from our `data.ecore` file into a file called `data.emf`. After changing the source in the emf file, you can right-click that file and select *Generate Ecore Model* to transfer it back.

data.ecore *data.emf

```
@namespace(uri="http://www.openarchitectureware.org/oaw4.demo.emf.datamodel",
           prefix="data")
package data;

class DataModel {
  attr String name;
  val Entity[*] entity;
}

class Entity {
  val Attribute[+] attribute;
  attr String name;
  val EntityReference[+] reference;
}

class Attribute {
  attr String name;
  attr String type;
}

class EntityReference {
  ref Entity[1] target;
  attr String name;
  attr boolean toMany;
}

class PersistentEntity extends Entity {
  attr String tableName;
}
```

Figure 16.1. Editing a metamodel with the EMFatic editor.

The syntax should be self-explaining. One remark: To render *containment* relationships, EMfatic uses the *val* keyword, not *ref*, see in the screenshot in the Entities attributes, for example.

Chapter 17. EMF State Machine

17.1. Introduction

This example shows how to implement a state machine generator using EMF and openArchitectureWare. Note that the implementation of the state machine in Java is probably the slowest and clumsiest way to implement a state machine. The focus was not on optimizing the performance of the state machine implementation.

This tutorial does not explain too much, it is rather a guide through the example code. We expect that you know how to work with openArchitectureWare and EMF. If that is not the case, you should read and play with the *emfHelloWorld* example first (the tutorial entitled *Generating Code from EMF Models*).

17.2. Installation

You need to have openArchitectureWare 4.3 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.

In the emf examples package, you can find the following three projects:

- `oaw4.demo.emf.statemachine`: contains the metamodel
- `oaw4.demo.emf.statemachine.generator`: contains the code generator
- `oaw4.demo.emf.statemachine.example`: contains an example state machine as well as a manually written unit test

17.3. Metamodel

The metamodel looks more or less as you would expect from a state machine metamodel. The following is the representation of the metamodel in Emfatic syntax. You can find it in the `oaw4.demo.emf.statemachine/model` package.

```
@namespace(uri="http://oaw/statemachine",
prefix="statemachine")
package statemachine;

abstract class Named {
 attr String name;
}

class State extends AbstractState {
 val Action entryAction;
 val Action exitAction;
}

class StartState extends AbstractState {
}

class StopState extends AbstractState {
}

class Transition extends Named {
 ref AbstractState[1] target;
 val Action action;
 ref Event event;
}

class Action extends Named {
}

class Event extends Named {
}

class CompositeEvent extends Event {
 val Event[*] children;
}

class StateMachine extends Named {
 val AbstractState[*] states;
 val Event[*] events;
}

abstract class AbstractState extends Named {
 val Transition[*] transition;
}
```

From the .ecore file, you have to generate the implementation classes, as it is usual with EMF.

17.4. Example Statemachine

In the `oaw4.demo.emf.statemachine.example/src` folder, you can find an `example.statemachine` file that contains a simple example state machine. You can view it as an EMF tree view after generating the EMF editors.

To generate code from it, run the example.oaw workflow file right next to it. It looks as follows:

```
<workflow>
 <cartridge file="workflow.oaw">
  <modelFile value="example.statemachine"/>
  <srcGenPath value="src-gen"/>
  <appProject value="oaw4.demo.emf.statemachine.example"/>
  <srcPath value="man-src"/>
 </cartridge>
</workflow>
```

As you can see, it only defines a number of parameters and calls another workflow file – the one in the generator project. We will take a look at it below.

17.4.1. Running the example

... is achieved by running the `example.oaw` file. It creates an implementation of the state machine in the *src-gen* folder in the example project. Take a look at the file to understand the implementation of the state machine.

In the *man-src* folder, there is a manually written subclass that implements the actions referenced from the state machine. There is also a unit test that you can run to verify that it works. It also shows you how to use the generated state machine.

17.5. The Generator

17.5.1. Workflow

The workflow file in `oaw4.demo.emf.statemachine.generator/src` has four steps:

- first it reads the model from the XMI file
- then it verifies a number of constraints
- then it generates the code
- and finally it creates a recipes file

17.5.2. Constraints

A number of constraints are defined. Take a look at their definition in *structure.chk* to learn about the constraints check language.

17.5.3. Transformation

There is a transformation called *trafo.ext* in the `src` folder which adds an emergency stop transition to each state.

17.5.4. Templates

In the `src/templates` folder you can find the code generation templates.

17.5.5. Recipe Creation

In `src/recipe`, there is an *SMRecipeCreator* workflow component that creates recipes for the manual implementation of the state machine.Recipe Creation

17.5.6. Acknowledgements

Thanks to the folks at *Rohde & Schwarz Bick Mobilfunk* for letting us use this example.

Chapter 18. Model-To-Model with UML2 Example

18.1. Introduction

This tutorial introduces various somewhat advanced topics of working with openArchitecureWare. We do not generate code, we transform models. Specifically, the tutorial covers the following:

- real model-to-model transformations
- how to test model-to-model transformations
- working with Eclipse UML2 (specifically, transforming away from it)
- using global variables
- implementing JAVA extensions

Note that this tutorial actually documents the *uml2ecore* utility, that can be used to transform metamodels drawn with an UML2 tool into an Ecore instance. This tool is available for download as a plugin on the oAW download page http://www.eclipse.org/gmt/oaw/download.

Please make sure that you first read the *UML2Ecore reference* reference. That document explains what to tool does – this is more or less a precondition to making sense *how* the tool does it (which is what this document explains).

Also note that many of the gory details are actually documented as code comments in the various oAW files of the example. So, make sure you have the code available as you read this document.

18.2. Why this example?

In general, we needed an example for real model-to-model transformations. And obviously, since we already had the *uml2ecore* tool, it makes sense to document this one. Also, including some information of working with UML2 is also important. However, there is one really good reason why looking at this example is important.

If you work with UML2 models, you will soon realize that the UML2 metamodel is very intricate, non-intuitive and complicated. Therefore, it is very good practice to transform away from the UML2 model as early as possible. Especially, if you want to enhance your model using model modifications (remember the emergency-stop feature in the state machine example?) this approach is recommended. While read access to UML2 models is tedious, write access is almost impossible to get correct, and in some aspects also impossible with generic EMF tools such as oAW.

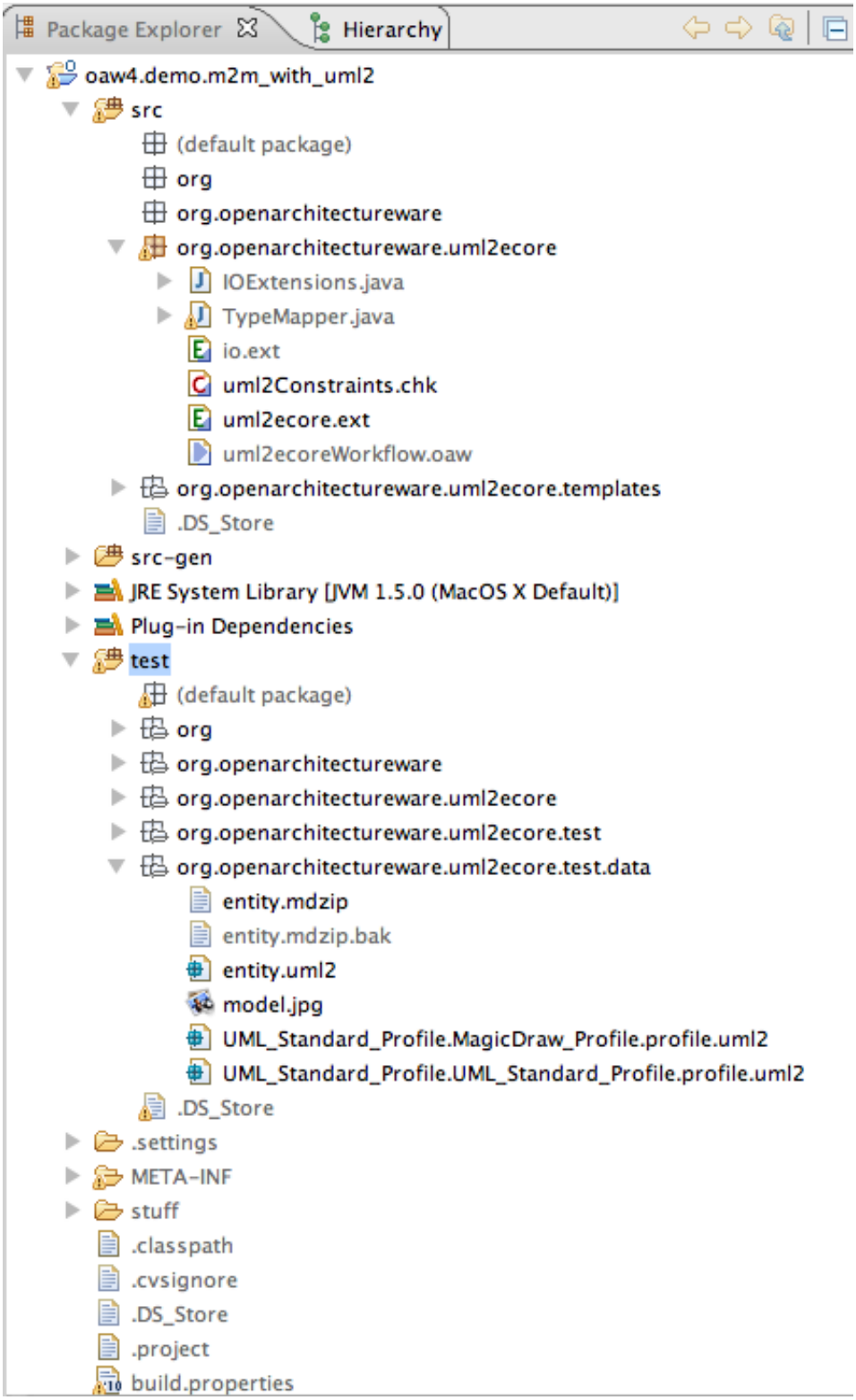
Therefore, if you work with a UML2 model, you should **always** begin by transforming into your own domain-specific metamodel.

In this example, we transform into an instance of Ecore in order to get the metamodel (since the *uml2ecore* tool is used for EMF metamodelling in UML tools). So, in your application projects, the transformation would not create an instance of Ecore, but an instance of your domain-specific metamodel (which, in turn, might have been created with uml2ecore). The only difference to the example given here is that you would use as a metamodel not Ecore, but rather your own one. This has effects on the metamodel declarations in the workflow file; everything else is similar.

18.3. Setting up Eclipse

You need an installation of oAW 4.3 including the UML2 support. Run the UML2 example (available for download on the oAW download page http://www.eclipse.org/gmt/oaw/download) to verify that you have all the UML2 stuff installed.

Then download the sample code for this tutorial and put the project into your workspace. It should look something like this:



18.4. The Building Blocks of the Transformer

The first thing you should look at (*after* reading the uml2ecore user guide!), is the workflow file. It reads the UML2 model, checks a couple of constraints against it and then invokes the transformation. Then, we write the result of the transformation out to an `.ecore` file.

`src/org/openarchitectureware/uml2ecore/uml2ecoreWorkflow.oaw`

You could now take a look at the constraints against the UML model. There are currently quite few of them in the code, but at least it shows how to write constraints against UML2 models

`src/org/openarchitectureware/uml2ecore/uml2Constraints.chk`

The transformation itself is located in `uml2ecore.ext`. While the transformation is not too complicated, it is also not trivial and illustrates many of the facets of the *Xtend* transformation language.

`src/org/openarchitectureware/uml2ecore/uml2ecore.ext`

To see how global variables are used, take a look at the <globalvar ... /> declaration in the workflow file, as well as the `nsUri()` function in `uml2exore.ext` .

To see JAVA extensions in action, take a look at `io.ext` and `IOExtensions.java`

18.5. Using the transformer

Usage of the transformer can be seen from the uml2ecore users guide mentioned a couple of times already. However, you can also take a look at the stuff in the testing section.

18.6. Testing an M2M transformation

M2M transformations can become quite intricate, especially if UML2 is involved. Therefore, testing such a transformation is essential. Note that you should resist the temptation to write tests in the form of generic constraints such as

For each Class in the UML2 model there has to be an EClass in the Ecore model

While this statement is true, it is also on the same level of genericity as the transformation itself. It is therefore not much less complex, and therefore as error prone as the transformation itself.

Instead what you should do is the following:

- Define a reference model. This model should contain all the possible alternatives of creating input models, i.e. should cover the complete variability of the metamodel.
- Then run the transformation on that reference model
- Finally, write *model-specific* constraints that verify the particular result of the transformation.

So in our example, we have a small metamodel modelled in UML using abstract and non-abstract classes, inheritance, attributes, 1:n and 1:1 references, bidirectional and unidirectional. The model, in our case, is drawn with MagicDraw 11.5 and exported as an UML2 model. You find all the model contents in the following location.

Here is a screenshot of the model:

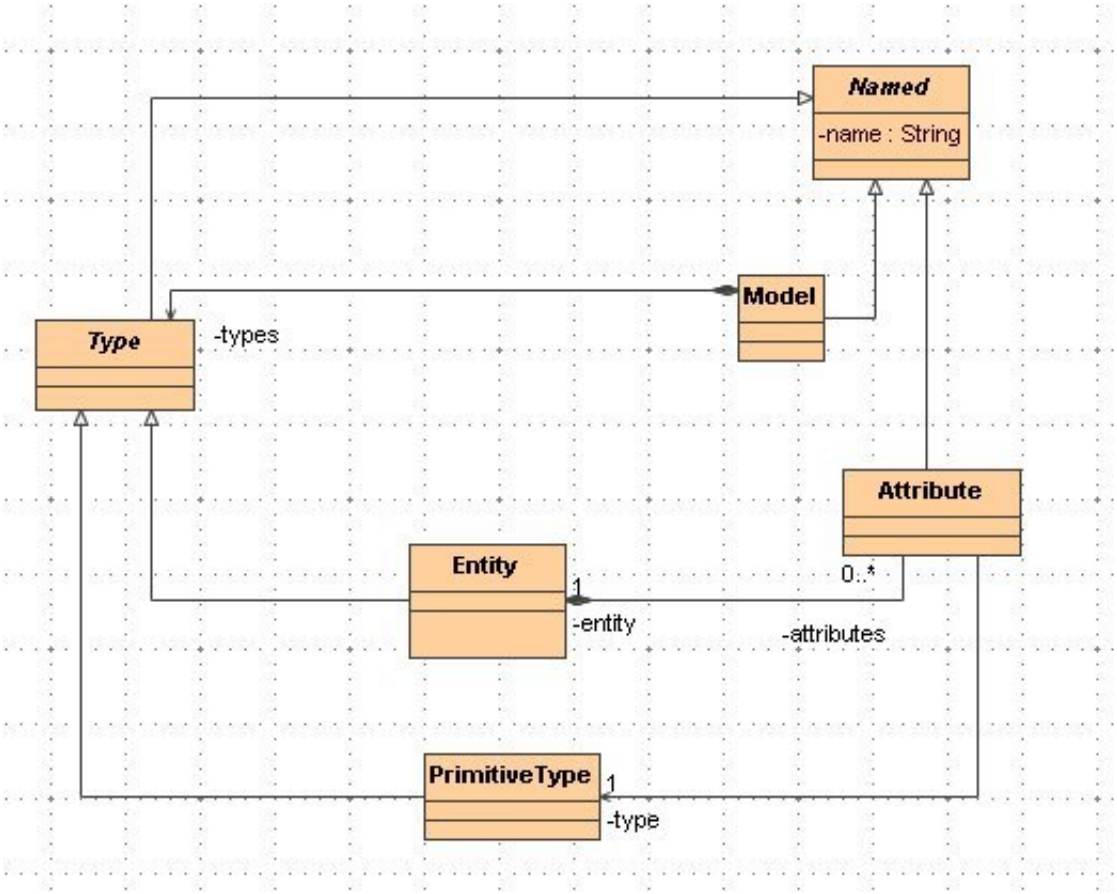


Figure 18.1. Model example

You now run the transformation using a workflow file that looks as follows:

`test/org/openarchitectureware/uml2ecore/test/runtests.oaw`

This workflow basically just invokes the transformation just as any other user of the transformation. However, in addition, after the transformation has finished, we validate test-specific constraints. This is what the additional constraint checker component is good for in the above mentioned workflow.

Now comes the point: The test constraints are not generic! Instead they do have knowledge about the input model (which generic constraints don't, they just operate with knowledge about the metamodel). So this file contains constraints such as

There has to be a class called Entity in the resulting model.

Or:

There must be an EReference owned by Attribute called type, pointing to the class called PrimitiveType.

Make sure that the constraints – again: coverage! – check all relevant aspects of the transformation.

`test/org/openarchitectureware/uml2ecore/test/tests.chk`

Glossary

D

DSL	Domain Specific Language

E

Ecore	A format used by the *Eclipse Modeling Framework to store its models*
EMF	Eclipse Modeling Framework

G

GMF	Graphical Modeling Framework.

M

MDA	Model-driven Architecture

Index

Symbols

A

C

D

E

F

G

I

J

L

N

O

P

R

S

T

W

X