

Xpand Documentation

Xpand Documentation

I. Getting Started	1
1. Getting Started	2
1. Installing the pre-built tutorial	2
2. Tutorial overview	2
3. Defining an EMF metamodel	2
3.1. Creating an EMF project	2
3.2. Defining the (meta)model	2
4. Generating the EMF tooling	5
5. Setting up the generator project	8
6. Defining an Example Data Model	10
7. Using Dynamic EMF	12
8. Generating code from the example model	13
8.1. The workflow definition	13
8.2. Running the workflow	13
8.3. Templates	15
8.4. Running the generator again	16
9. Checking Constraints with the <i>Check</i> Language	17
9.1. Defining the constraint	17
9.2. Integration into the workflow file	17
10. Extensions	18
10.1. Expression Extensions	18
10.2. Java Extensions	18
II. Reference	21
2. Xpand / Xtend / Check Reference	22
1. Introduction	22
2. Type System	22
2.1. Types	22
2.2. Built-In Types	23
2.3. Metamodel Implementations (also known as Meta-Metamodels)	24
2.4. Using different Metamodel implementations (also known as Meta-Metamodels)	25
2.5. Metamodel Reference	25
3. Expressions	28
3.1. Literals and special operators for built-in types	28
3.2. Special Collection operations	30
3.3. <i>if</i> expression	32
3.4. <i>switch</i> expression	32
3.5. Chain expression	32
3.6. <i>create</i> expression	33
3.7. <i>let</i> expression	33
3.8. 'GLOBALVAR' expression	33
3.9. Multi methods (multiple dispatch)	33
3.10. Casting	34
3.11. Xpand keywords and metamodel properties	34
4. <i>Check</i>	34
4.1. Description of the <i>Check</i> language	34
4.2. The workflow component <i>CheckComponent</i>	35
5. <i>Xtend</i>	35
5.1. Xtend files	35
5.2. Comments	36
5.3. Import Statements	36
5.4. Extension Import Statement	36
5.5. Extensions	36
5.6. Java Extensions	38
5.7. Create Extensions (Model Transformation)	39
5.8. Calling Extensions From Java	40
5.9. WorkflowComponent	41
5.10. Aspect-Oriented Programming in <i>Xtend</i>	41
6. <i>Xpand2</i>	44

6.1. Template files and encoding	44
6.2. General structure of template files	44
6.3. Statements of the <i>Xpand</i> language	44
6.4. Aspect-Oriented Programming in <i>Xpand</i>	50
6.5. Generator Workflow Component	51
6.6. Example for using Aspect-Oriented Programming in <i>Xpand</i>	55
6.7. The Problem	55
6.8. Example	55
6.9. More Aspect Orientation	56
7. Profiler	57
3. Built-in types API documentation	59
1. Object	59
2. String	59
3. Integer	60
4. Boolean	61
5. Real	61
6. Collection	62
7. List	63
8. Set	63
9. <code>xpand2::Type</code>	63
10. <code>xpand2::Feature</code>	64
11. <code>xpand2::Property</code>	64
12. <code>xpand2::Operation</code>	64
13. <code>xpand2::StaticProperty</code>	64
14. Void	64
15. <code>xtend::AdviceContext</code>	64
16. <code>xpand2::Definition</code>	65
17. <code>xpand2::Iterator</code>	65
4. Stdlib	66
1. Introduction	66
2. Stdlib extensions	66
2.1. IO extensions	66
2.2. Counter extensions	68
2.3. Properties extensions	69
2.4. Element properties extensions	70
2.5. Issues extensions	70
2.6. Naming extensions	71
2.7. Globalvar extensions	72
2.8. Cloning extensions	72
2.9. Cross references extensions	73
2.10. UID extensions	73
2.11. Mixin extensions	74
2.12. Tracing extensions	75
3. Stdlib workflow components	76
3.1. <code>SystemCommand</code>	76
3.2. <code>SlotCopier</code>	76
3.3. <code>SlotListAdder</code>	77
3.4. <code>SlotPrinter</code>	77
5. Xpand Eclipse Integration	79
1. Introduction	79
2. Installation	79
3. Overview	79
4. File decorations	79
5. Editors	79
5.1. Syntax coloring	79
5.2. Code completion	80
5.3. <i>Xpand</i> tag delimiter creation support	80
6. Preference pages	80

6.1. Metamodel contributors	80
6.2. Global preferences	80
6.3. Preferences per project	81
7. Xpand Nature and Xpand Builder	81
7.1. Problem markers	81
8. Running a workflow	82
III. Tutorials	83
6. UML2 Adapter	84
1. Introduction	84
2. Installation	84
3. Setting up Eclipse	84
3.1. Profiles in Eclipse	84
4. Runtime Configuration	84
4.1. Workflow	84
7. UML2 Example	86
1. Setting up Eclipse	86
2. Setting up the project	86
3. Creating a UML2 Model	86
3.1. Modelling the content	88
4. Code generation	88
4.1. Defining the templates	88
4.2. Defining the workflow	89
5. Profile Support	89
5.1. Defining a Profile	89
5.2. Applying the Profile	90
5.3. Generating Code	91
8. XSD Tutorial	93
1. Setup	93
2. Overview	93
3. Step 1: Create a Project	93
4. Step 2: Define a Meta Model using XML Schema	94
5. Step 3: Create a Model using XML	94
6. Step 4: Create a Template using Xpand	95
7. Step 5: Create a Workflow	95
8. Step 6: Execute Workflow aka Generate Code	96
9. XSD Adapter	97
1. Prerequisites	97
2. Overview	97
3. Workflow Components	97
3.1. XSDDMetaModel	97
3.2. XMLReader	98
3.3. XMLWriter	98
3.4. XMLBeautifier	99
4. Behind the scenes: Transforming XSD to Ecore	99
5. How to declare XML Schemas	99
10. Incremental Generation	101
1. Technical Background	101
2. Using Incremental Generation	101
2.1. The Incremental Generation Facade	101
2.2. The Incremental Generation Callback	102
3. Additional Notes	103
3.1. Limitations	103
3.2. Performance Considerations	103

Part I. Getting Started

Chapter 1. Getting Started

This example uses Eclipse EMF as the basis for code generation. A 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 this tutorial without a complete understanding of EMF, but the tutorial might seem unnecessarily complex to you.

1. Installing the pre-built tutorial

Before you can go through the tutorial and execute the examples, you need to have Xpand installed. Please consider <http://www.eclipse.org/modeling/m2t/downloads/?project=xpand> for details.

2. Tutorial overview

The purpose of this tutorial is to illustrate code generation with Xpand 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 do this for this simple introduction.

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. Sample metamodel

3.1. Creating an EMF project

Create an EMF project as depicted below:

Figure 2. Create EMF project

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

3.2. Defining the (meta)model

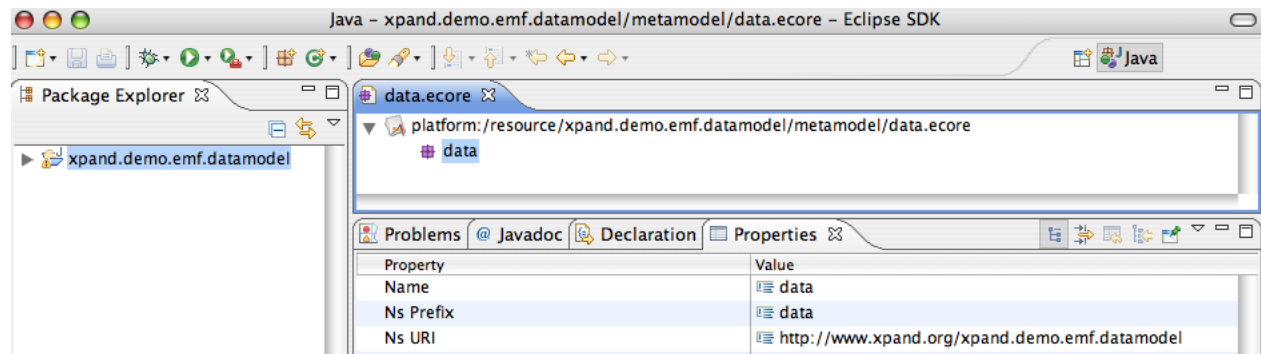
Create a new source folder `metamodel` in that project. Then, create a new Ecore model in that source folder named `data.ecore`. Use EPackage as the model object.

Figure 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.xpand.org/xpand.demo.emf.datamodel`

Figure 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
DataModel			
	entity		
		EType	Entity

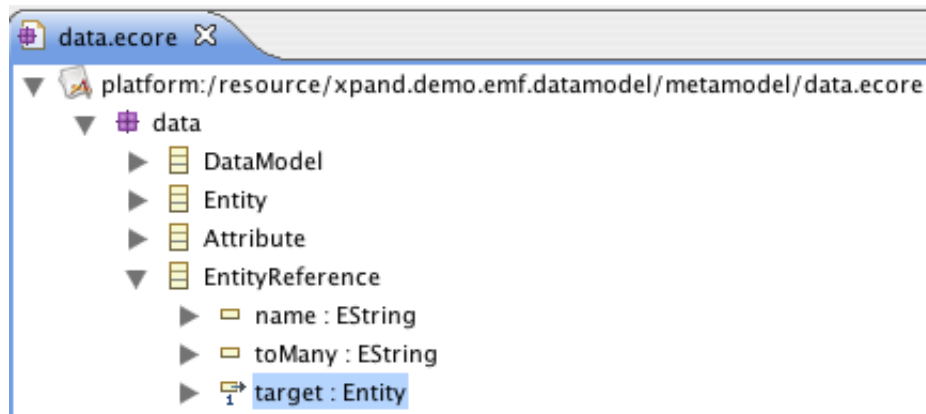
¹To add children, right-click on the element to which you want to add these children and select the type of the child from the list. To configure the properties, open the properties dialog by selecting Show Properties View at the bottom of any of the context menus. Note that this is not an EMF tutorial. For more details on how to build EMF (meta-)models, please refer to the EMF documentation.

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

³Note: there are a couple of -/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
		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 5. Metamodel structure



EMF saves the model we created above in its own dialect of XML. To avoid any ambiguities, here is the complete XML 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.xpand.org/xpand.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"/>
  </eClassifiers>
</ecore:EPackage>
```

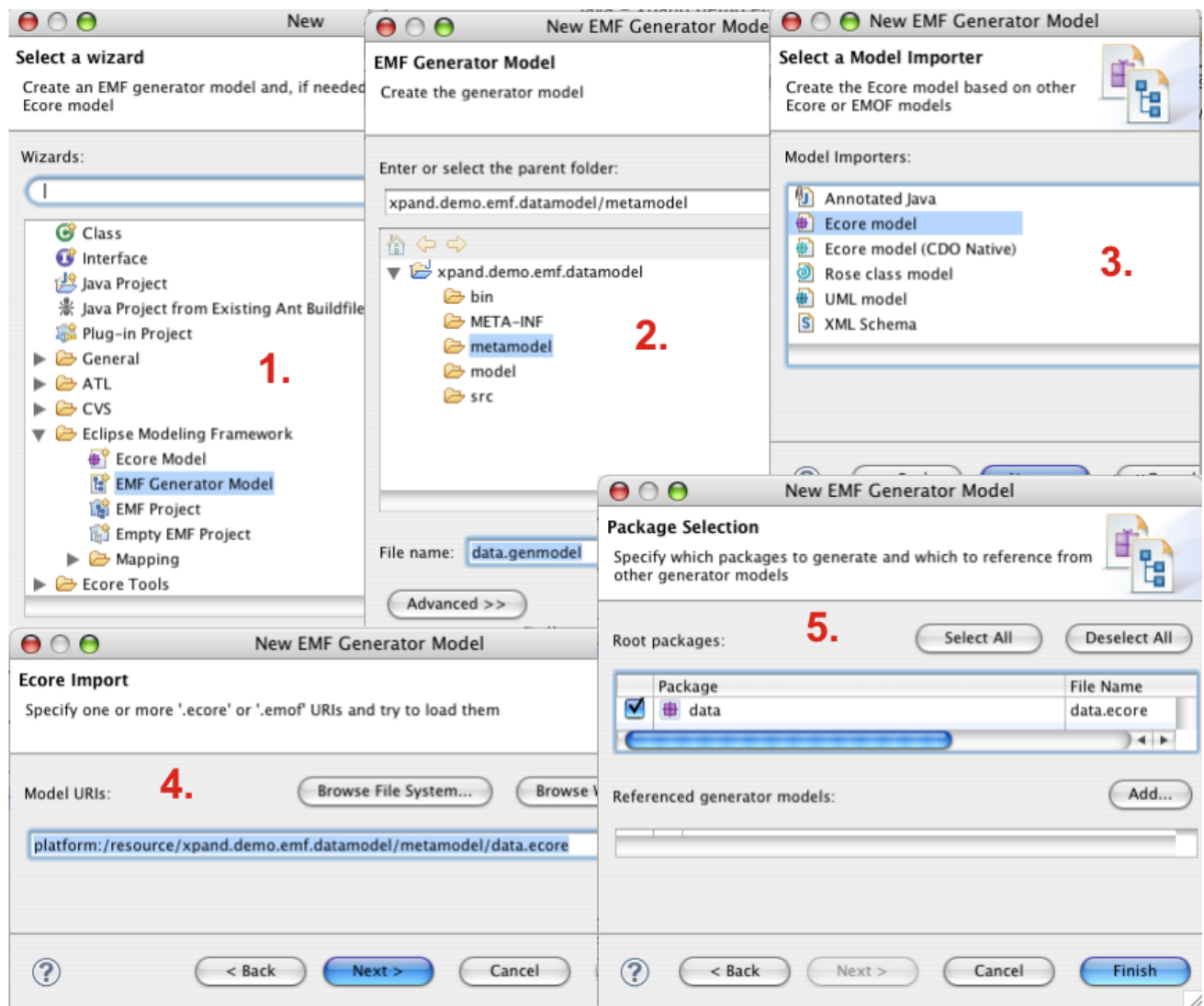
```
<eStructuralFeatures xsi:type="ecore:EReference" name="entity" upperBound="-1"
  eType="#//Entity" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Entity">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
    eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <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>
```

4. Generating the EMF tooling

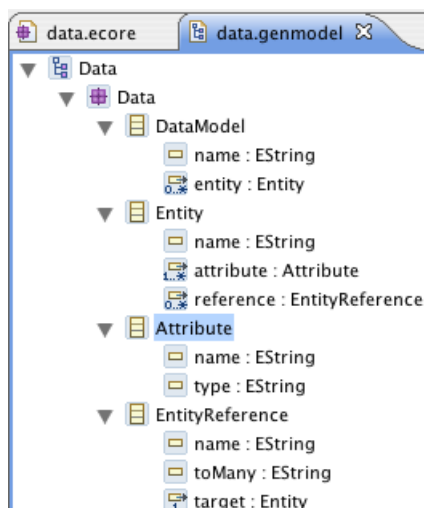
In addition to providing the Ecore meta-metamodel, EMF also comes with support for building (more or less usable) editors. These are generated automatically from the metamodel we just defined. In order to define example models (which we will do below) we have to generate these editors. Also, we have 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 Generator Model`. Follow the following five steps; note that they are also illustrated in the next figure.

1. Select EMF Generator Model
2. Define the name
3. Select the folder
4. Select Ecore model as source
5. Press the *Load* button and then *Finish*

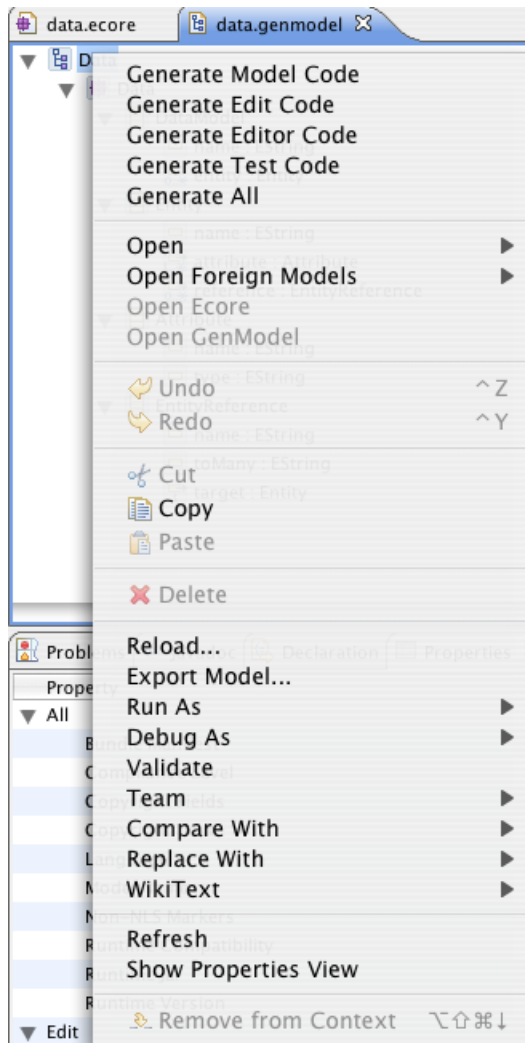
Figure 6. Creating the genmodel

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 7. Structure of the genmodel

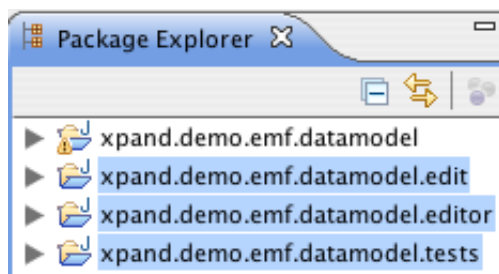
You can now generate the other projects.

Figure 8. Generate editing projects



You now have all the generated additional projects.

Figure 9. Generated projects

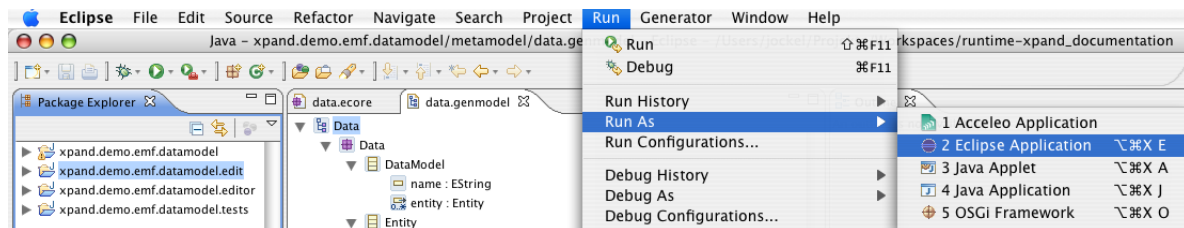


We will not look any deeper at these additional projects for now. However, there is one important thing to point out: The generator also generated the implementation classes for the metamodel. If you take a look into `xpand.demo.emf.datamodel/src` folder, you can find classes (actually, interfaces at 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.

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 `xpand.demo.emf.datamodel.edit` project and choose from the context menu `Run As` → `Eclipse Application`.

Figure 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 `xpand.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.

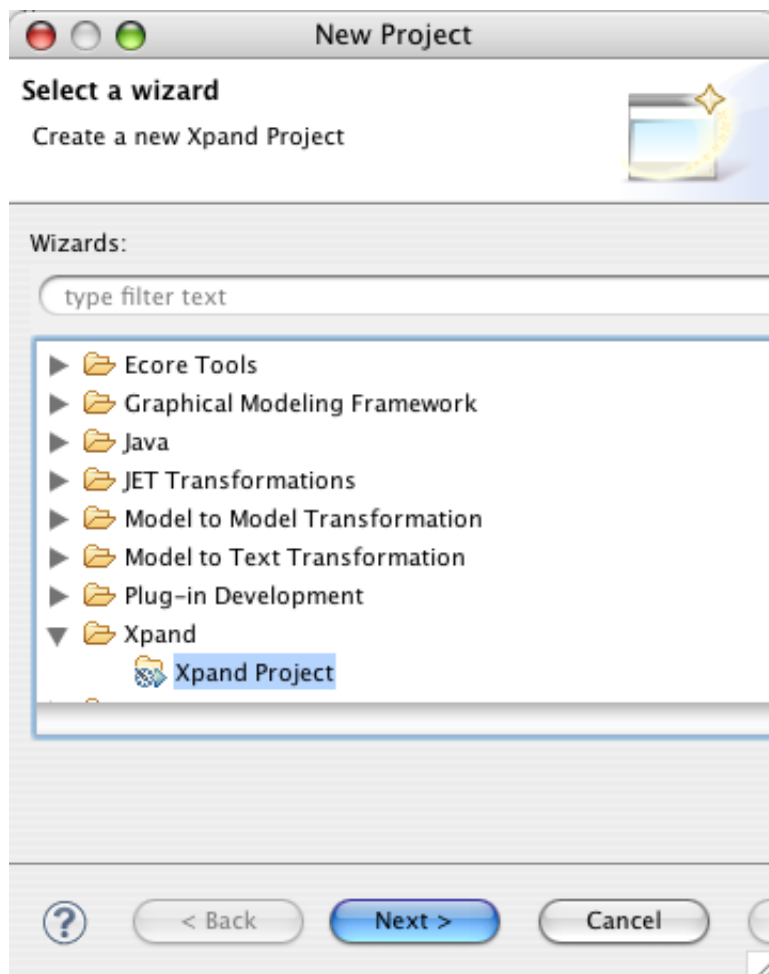
Create a new Xpand Project⁷ called `xpand.demo.emf.datamodel.generator`. Do not choose the option "Generate a simple example".

⁴Window → Preferences → General → Workspace → Text file encoding. This is necessary to have the *guillemet* brackets available.

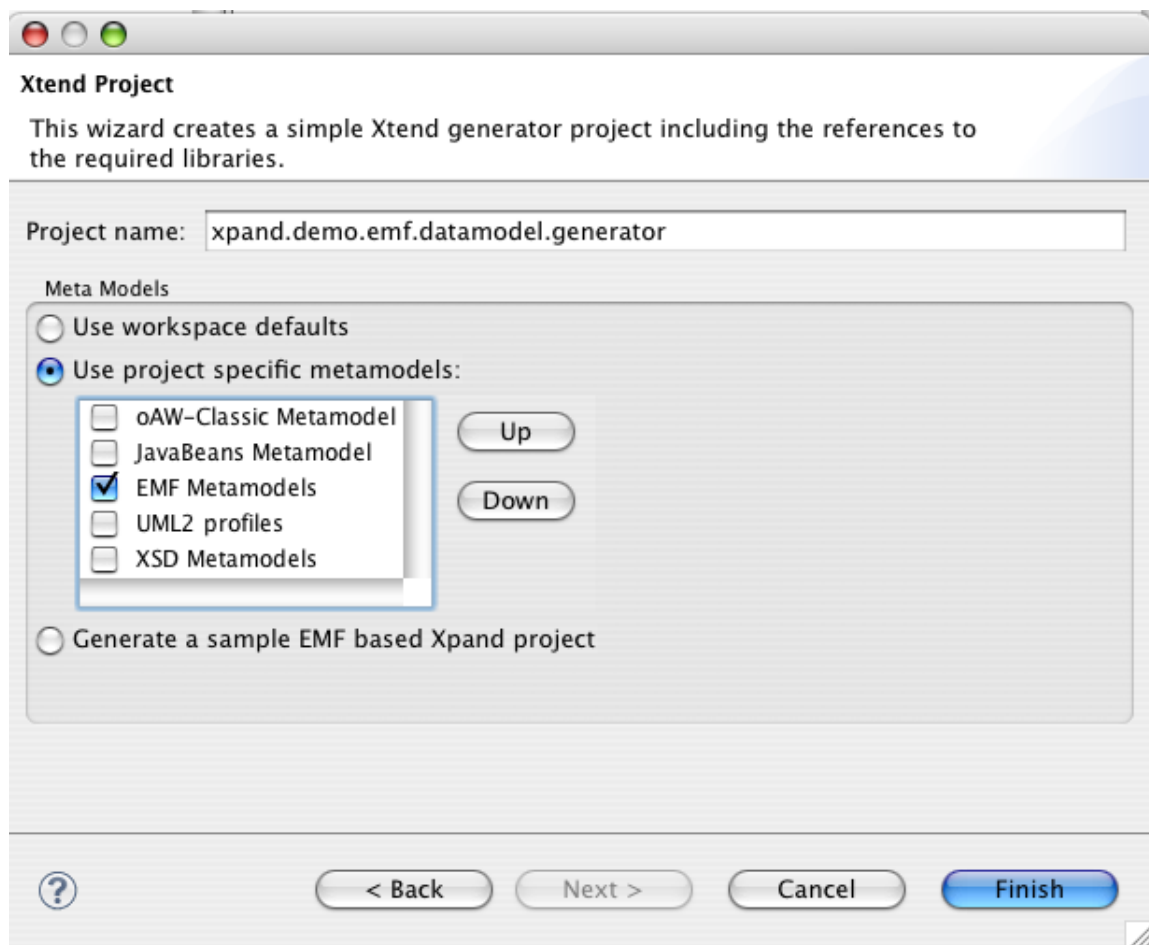
⁵File → Import → General → Existing Project into Workspace

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

⁷File → New → Project → Xpand → Xpand Project

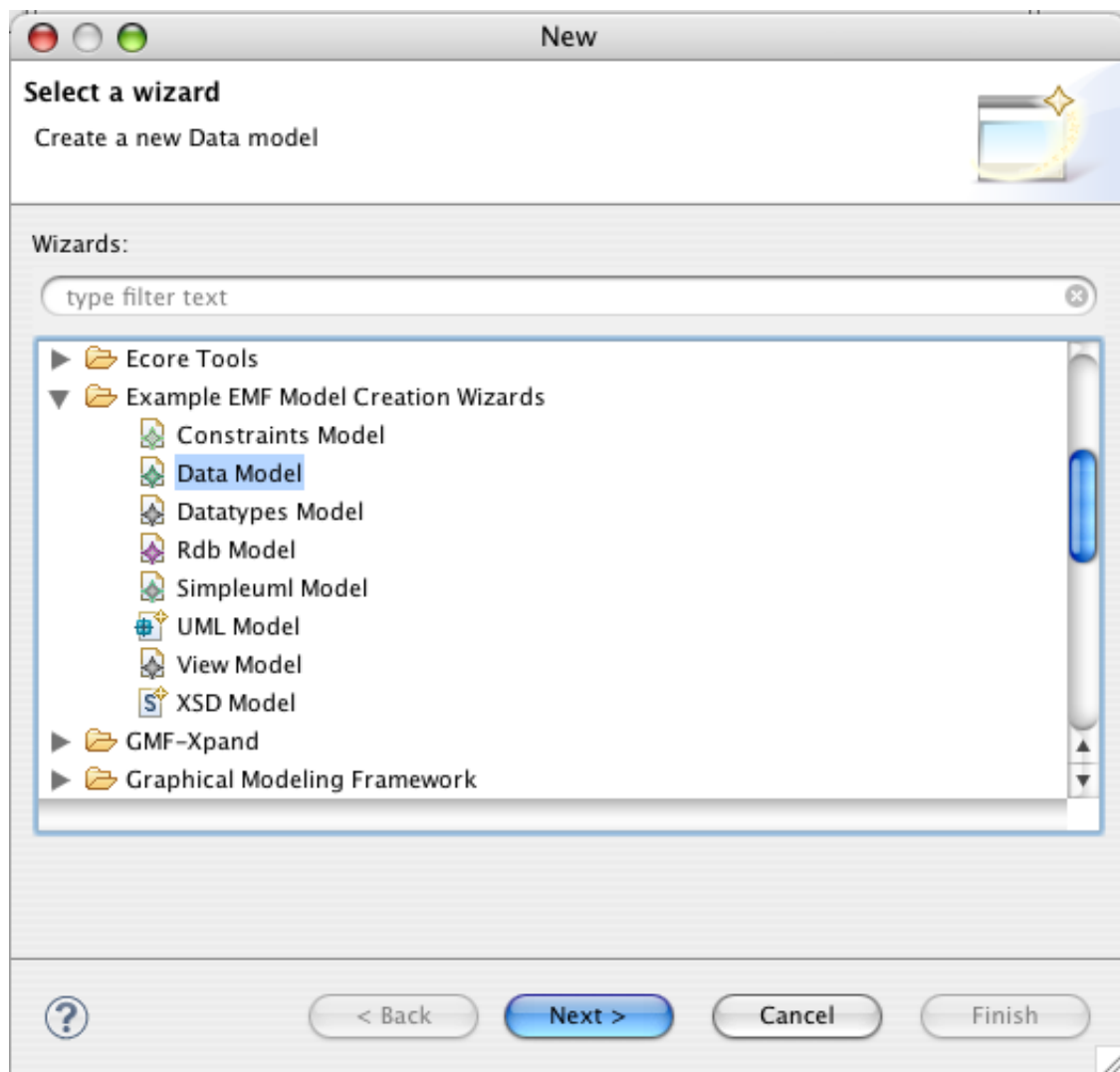
Figure 11. Create new Xpand project

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

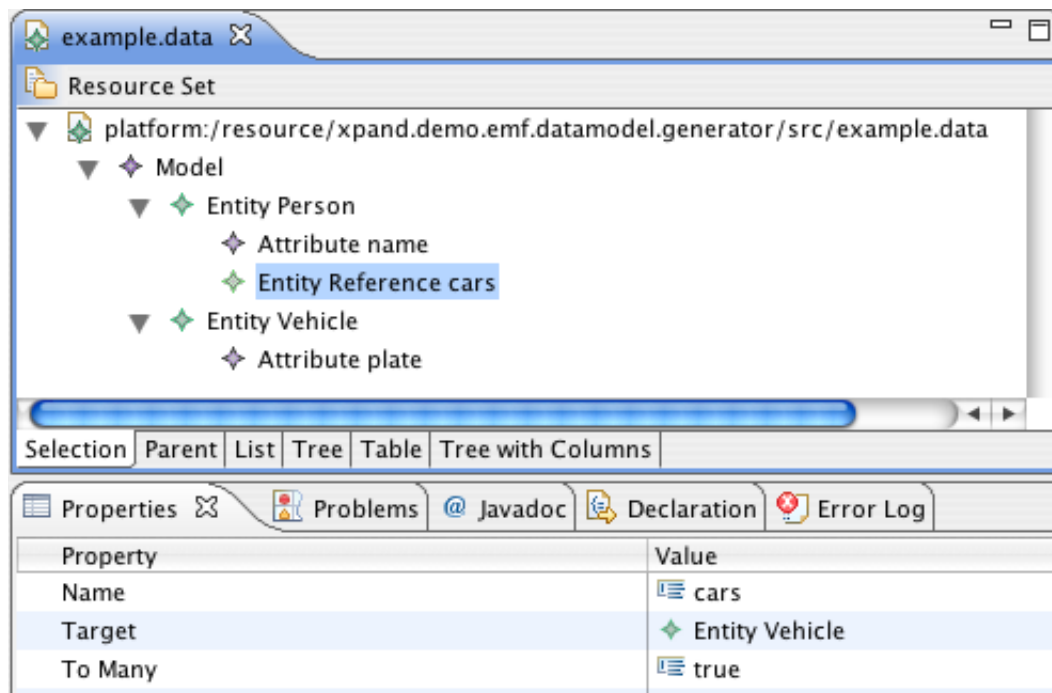
Figure 12. Project properties

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 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 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.xpand.org/xpand.demo.emf.datamodel">
  <entity name="Person">
    <attribute name="name" type="String"/>
    <reference name="cars" toMany="true" target="//@entity.1"/>
  </entity>
  <entity name="Vehicle">
    <attribute name="plate" type="String"/>
  </entity>
</data:DataModel>
```

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 Create Dynamic Instance from the context menu. This opens an editor that can dynamically edit the respective instance. The created file by default has an `.xmi` extension.

Note that Xpand 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.eclipse.xtend.typesystem.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.eclipse.xtend.typesystem.emf.EmfMetaModel">
<metaModelFile value="data.ecore"/>
</metaModel>
```

8. Generating code from the example model

8.1. The workflow definition

To run the Xpand 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 *Modeling Workflow Engine Reference Documentation*.

Create a `workflow.mwe` and a `workflow.properties` in the `src` folder. The contents of these files is shown below:

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

  <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup" >
    <platformUri value="."/>
    <registerGeneratedEPackage value="data.DataPackage"/>
  </bean>

  <component class="org.eclipse.emf.mwe.utils.Reader">
    <uri value="platform:/resource/${modelFile}" />
    <modelSlot value="model" />
  </component>
</workflow>
```

`workflow.properties`:

```
modelFile=platform:/resource/xpand.demo.emf.datamodel/src/example.data
srcGenPath=src-gen
fileEncoding=ISO-8859-1
```

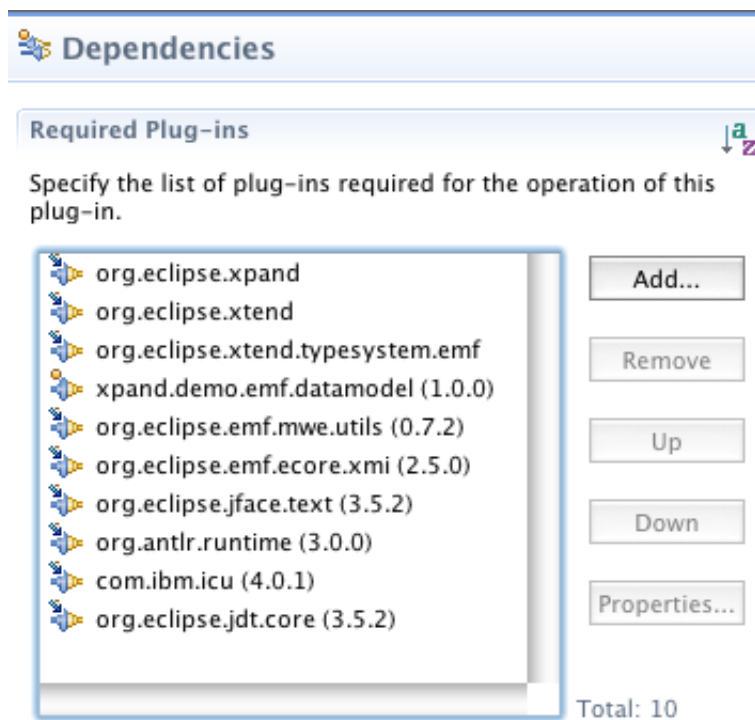
The workflow tries to load stuff from the classpath; so, for example, the `data.DataPackage` class is resolved from the classpath, as is the model file specified in the properties (`modelFile=example.data`)

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).

8.2. Running the workflow

Before you actually run the workflow, make sure your metamodel can be found on the classpath. In our case, this can be achieved by adding the `xpand.demo.emf.datamodel` project to the plug-in dependencies of `xpand.demo.emf.datamodel.generator`. To do this, double click the file `xpand.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 few new dependencies.

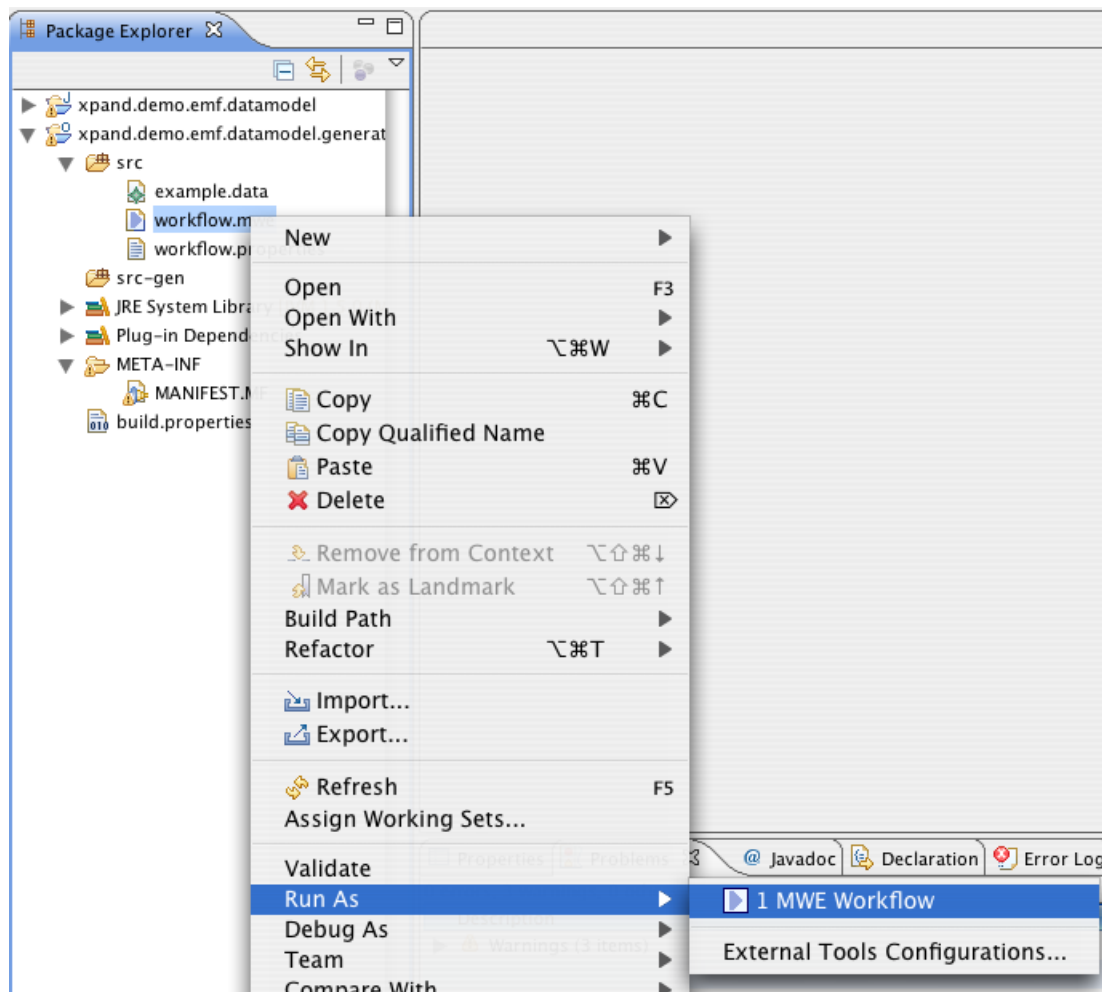
- `xpand.demo.emf.datamodel`
- `org.eclipse.emf.mwe.utils`
- `org.eclipse.emf.ecore.xml`
- `org.eclipse.jface.text`
- `org.antlr.runtime`
- `com.ibm.icu`
- `org.eclipse.jdt.core`

Figure 15. Add metamodel dependency

Do not forget to save the manifest file!

Now, you can run the workflow from within Eclipse:

Figure 16.



The following should be the output:

```
INFO: -----
INFO: EMF Modeling Workflow Engine 0.7.2, Build v200908120417
INFO: (c) 2005-2009 openarchitectureware.org and contributors
INFO: -----
INFO: running workflow: ../xpand.demo.emf.datamodel.generator/src/workflow.mwe
INFO:
14.04.2010 15:49:18 org.eclipse.emf.mwe.utils.StandaloneSetup setPlatformUri
INFO: Registering platform uri '..'
14.04.2010 15:49:18 org.eclipse.emf.mwe.utils.StandaloneSetup addRegisterGeneratedEPackage
INFO: Adding generated EPackage 'data.DataPackage'
14.04.2010 15:49:18 org.eclipse.emf.mwe.core.container.CompositeComponent internalInvoke
INFO: Reader: Loading model from platform:/resource/xpand.demo.emf.datamodel.generator/src/example.data
14.04.2010 15:49:19 org.eclipse.emf.mwe.core.WorkflowRunner executeWorkflow
INFO: workflow completed in 116ms!
```

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»
«ENDDDEFINE»
```

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

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

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

  ..

  <component class="org.eclipse.emf.mwe.utils.Reader">
    <uri value="platform:/resource/${modelFile}" />
    ..
  </component>
</workflow>
```

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

```
<component class="org.eclipse.emf.mwe.utils.DirectoryCleaner">
  <directory value="${srcGenPath}" />
</component>
```

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

```
<component class="org.eclipse.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 specify the class name of the EMF package that represents that metamodel. It has to be on the classpath.

```
<metaModel id="mm"
  class="org.eclipse.xtext.typesystem.emf.EmfRegistryMetaModel">
</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.eclipse.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
```

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;  
}
```

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.

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 ) );
```

9.2. Integration into the workflow file

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

```
<workflow>  
    <property file="workflow.properties"/>  
  
    ..  
  
    <component class="org.eclipse.emf.mwe.utils.Reader">  
        <uri value="platform:/resource/${modelFile}" />  
        ..  
    </component>  
</workflow>
```

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

```
<component  
    class="org.eclipse.xtend.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.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel">  
</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).

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.

10.1. Expression Extensions

Assume we wanted to change the *Attributes* part of the template as follows:

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

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

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

To make this work, we need to define the `setterName()` and `getName()` 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 Xpand; 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 `getName`. 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 getName(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»
```

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 look 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 other Xtend files 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. 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»

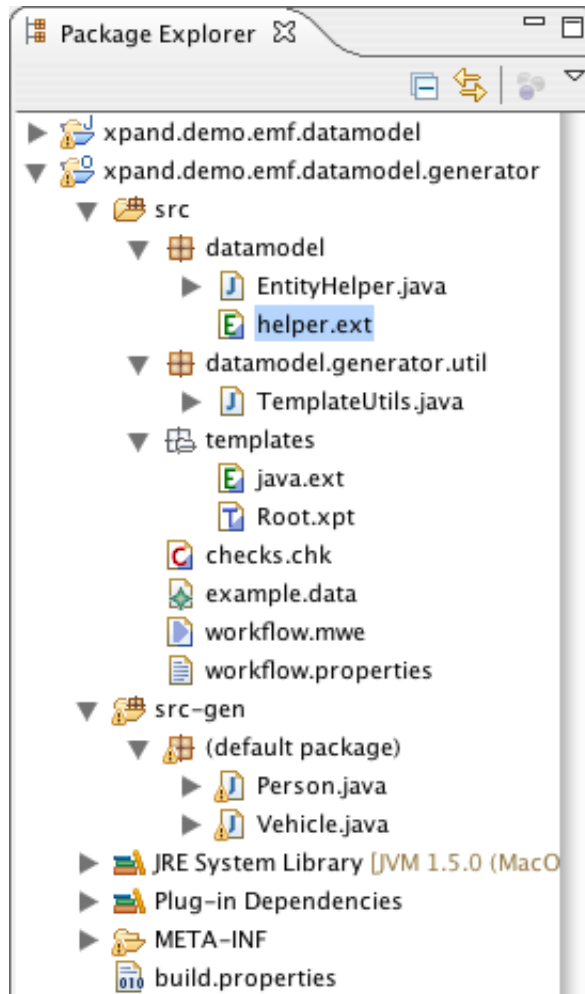
«DEFINE Entity FOR data::Entity»
    «FILE classFileName()»
    // generated at «timestamp()»
    public abstract class «className()» {
        «FOREACH attribute AS a»
            private «a.type» «a.name»;
            public void «a.setterName()»( «a.type» value ) {
                this.«a.name» = value;
            }

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


«ENDDFINE»

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

Figure 17. What has happened so far



Part II. Reference

Chapter 2. *Xpand* / *Xtend* / *Check* Reference

1. Introduction

The *Xpand* generator framework provides textual languages, that are useful in different contexts in the MDSD process (e.g. checks, extensions, code generation, model transformation). Each 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.

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.eclipse.xpand.type` package and is a part of the subproject `core.expressions`.

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).

2.1.1. Type Names

Types have a simple name (e.g. `String`) and an optional namespace used to distinguish 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.

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 `ClassCastException`s, 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]
```

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.

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/>).

2.2.1. Object

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

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.

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*!

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.

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 `xpand`. We have:

- `xpand2::Type`
- `xpand2::Feature`
- `xpand2::Property`
- `xpand2::StaticProperty`
- `xpand2::Operation`

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 *Xpand* type system elements (`Type`, `Property`, `Operation`) are mapped to an arbitrary other type system (Java reflections, *Ecore* or XML Schema).

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 *Xpand* 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 `xpand2::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 *Xpand* 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.

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 *Xpand* 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.

2.3.3. Configuring Metamodel implementations with the workflow

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

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

```
<component class="org.eclipse.xpand2.Generator">
  <metaModel class="org.eclipse.type.emf.EmfMetaModel">
    <metaModelPackage value="my.generated.MetaModelPackage"/>
  </metaModel>
  <metaModel class="org.eclipse.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.emf.tools` project). It points to the generated `EPackages` interface. The second meta model is configured using the `Ecore` file. You do not need to have a generated `Ecore` model for *Xpand* in order to work. The `EmfMetaModel` works with dynamic EMF models just as it works with generated EMF models.

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

With *Xpad* 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="org.eclipse.xpand2.Generator">
  <metaModel class="org.eclipse.internal.xtend.type.impl.java.JavaMetaModel"/>
  <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
    <metaModelFile value="my/java/package/metamodel.ecore"/>
  </metaModel>

  ...
</component>
```

When the 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 *Xpand*). At this point the type `Object` best fits the request, so it will act as the desired type.

The second metamodel returns a type called `org::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 `org::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="org.eclipse.xpand2.Generator">
  <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
    <metaModelFile value="my/java/package/metamodel.ecore"/>
  </metaModel>
  <metaModel class="org.eclipse.internal.xtend.type.impl.java.JavaMetaModel"/>

  ...
</component>
```

The order of the metamodels is important for the work within the *Xpand*-editors. The metamodels to work with can be configured inside the *Xtend/Xpand* -properties dialog. The *Activated metamodel contributors* table is a ordered list. The more specific metamodels have to be placed at the top of the list.

2.5. Metamodel Reference

In the following, each of the built-in metamodels that come with *Xpand* will be documented. Furthermore, there will be some guidelines on how to implement your own metamodel.

2.5.1. EMF Metamodels

This section will describe the metamodels that can be used for EMF models. **Please note that you have to execute one of the setup utility classes, `Setup` or `StandaloneSetup`, in your workflow before you can use one of the EMF metamodels**

2.5.1.1. The EMF Registry Metamodel (`org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel`)

This metamodel looks for the referenced metamodels in the global EMF model registry. This means, when using this metamodel, only models that are registered in this global EMF model registry will be accessible from within the metamodel.

This metamodel provides the following configuration property:

Table 1. Properties of `EmfRegistryMetaModel`

Name of property	Description
<code>useSingleGlobalResourceSet</code>	This boolean property determines the way resource sets are used. If set to <i>true</i> , all model resources will be stored in a single global resource set. Otherwise, a separate resource set will be used for each model resource.

2.5.1.2. The EMF Metamodel (`org.eclipse.xtend.typesystem.emf.EmfMetaModel`)

This metamodel is a specialized version of the EMF registry metamodel. In addition to the features of the former, it allows to specify an unregistered model in different ways that will be added to the metamodel.

This metamodel provides the following configuration properties:

Table 2. Properties of `EmfMetaModel`

Name of property	Description
<code>useSingleGlobalResourceSet</code>	This boolean property determines the way resource sets are used. If set to <i>true</i> , all model resources will be stored in a single global resource set. Otherwise, a separate resource set will be used for each model resource.
<code>metaModelFile</code>	Sets the path to the Ecore file that will be added to the metamodel.
<code>metaModelDescriptor</code>	Adds a model to the metamodel by specifying the name of an EPackage descriptor class.
<code>metaModelPackage</code>	Adds a model to the metamodel by specifying the name of an EPackage.

2.5.2. UML Metamodels

Xpand also provides several metamodels that allow to use UML models in conjunction with this model-to-text generation framework. **Please note that you have to execute the setup utility class `Setup` in your workflow before you can use one of the UML metamodels**

2.5.2.1. The UML2 Metamodel (`org.eclipse.xtend.typesystem.uml2.UML2MetaModel`)

This metamodel is a specialized version of the *EMF metamodel*. It provides access to UML2 models, and it has the following configuration properties:

Table 3. Properties of `UML2MetaModel`

Name of property	Description
<code>useSingleGlobalResourceSet</code>	This boolean property determines the way resource sets are used. If set to <i>true</i> , all model resources will

Name of property	Description
	be stored in a single global resource set. Otherwise, a separate resource set will be used for each model resource.
modelFile	Sets the path to the UML2 model file that will be added to the metamodel.

2.5.2.2. The UML2 Profile Metamodel (org.eclipse.xtend.typesystem.uml2.profile.ProfileMetaModel)

This metamodel allows to apply UML profiles to UML2 models. It has the following configuration properties

Table 4. Properties of ProfileMetaModel

Name of property	Description
useSingleGlobalResourceSet	This boolean property determines the way resource sets are used. If set to <i>true</i> , all model resources will be stored in a single global resource set. Otherwise, a separate resource set will be used for each model resource.
modelFile	Sets the path to the UML2 model file that will be added to the metamodel.
profile	Sets the path to the UML profile that will be applied to the UML2 model.

2.5.2.3. The XMI reader (org.eclipse.xtend.typesystem.uml2.profile.ProfilingExtensions.XmiReader)

The XMI reader component is important when working with UML models. It allows to read out a model stored in an XMI file and put its contents into a model slot.

The `XMIReader` component provides the following configurable properties:

Table 5. Properties of XMIReader

Name of property	Description
metaModelFile	Sets the path to the Ecore file that will be added to the metamodel.
metaModelDescriptor	Adds a model to the metamodel by specifying the name of an EPackage descriptor class.
metaModelPackage	Adds a model to the metamodel by specifying the name of an EPackage.
outputSlot	Sets the name of the model slot where the read model will be stored in.
firstElementOnly	This boolean property determines if only the first model element of the XMI file will be used. If set to <i>true</i> , only the first model element will be used, all other elements will be ignored. Otherwise, all model elements in the XMI file will be used.

2.5.3. The Java Metamodel (org.eclipse.internal.xtend.type.impl.java.JavaMetaModel)

The Java metamodel allows normal Java classes as metatypes for your metamodel. The `JavaMetaClass` uses the strategy pattern to define how the elements are exactly mapped to the metamodel elements. There is a class called `org.eclipse.internal.xtend.type.impl.java.JavaBeansMetaModel` that is preconfigured with a strategy that maps simple Java beans onto the metamodel elements.

The Java metamodel has no configurable properties.

2.5.4. The XSD Metamodel (`org.eclipse.xtend.typesystem.xsd.XSDMetaModel`)

The XSD metamodel provides access to models implemented in the XML Schema Definition language. It has the following configuration properties:

Table 6. Properties of `XSDMetaModel`

Name of property	Description
<code>id</code>	Sets the ID of the current model.
<code>registerPackagesGlobally</code>	This boolean property determines if the model packages will be registered globally. If set to <i>true</i> , the model packages will be registered in the global registry. Otherwise, packages will not be registered.
<code>savePackagesPath</code>	Sets the path where model packages will be saved (in XMI format).

2.5.5. Implementing Your Own Metamodel

The *Xpand* framework also allows you to integrate new metamodel implementations. This section quickly outlines the steps that have to be taken in order to implement a metamodel:

1. Create a class that implements the `MetaModel` interface.
2. In order to be able to integrate your metamodel into the Eclipse UI, you also have to provide a metamodel contributor class for your metamodel implementation that implements either the `MetaModelContributor` or the `MetaModelContributor2` interface.
3. Finally, you have to extend the `org.eclipse.xtend.shared.ui.metaModelContributors` extension point in order to register your metamodel contributor with the Eclipse UI.

3. Expressions

The 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)
```

3.1. Literals and special operators for built-in types

There are several literals for built-in types:

3.1.1. Object

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

equals:

```
obj1 == obj2
```

not equals:

```
obj1 != obj2
```

3.1.2. Void

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

```
null
```

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'
```

3.1.4. StaticProperty literals

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

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

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.

3.1.6. Boolean

The boolean literals are:

```
true
false
```

Operators are:

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

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
```

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
```

3.2. Special Collection operations

Like OCL, the *Xpand* 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.

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}
```

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).

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}
```

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 apply an operation on all elements of the collection and collect the results of the evaluation in a list. 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
namedElements.collect(ne | ne.name.length > 3) // returns a list of boolean
```

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.

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)
```

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)
```

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:

¹since 4.1.2

```
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 *Xpand*. If the values returned from the expression are instances of `java.util.Comparable` the `compareTo` method is used, otherwise `toString()` is invoked and the result is used.

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}
...
```

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'
```

Alternatively, you also could write:

```
if name != null then
  name
else
  'unknown'
```

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:

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

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.

3.6. create expression

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

```
new TypeName
```

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
```

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.

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="org.eclipse.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();
```

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.

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()
```

3.11. Xpand keywords and metamodel properties

When the name of a metamodel property conflicts with an *Xpand* or *Xtend* keyword, the conflict is resolved in favour of the keyword. To refer to the metamodel property in these cases, its name must be preceded by a '^' character.

Example:

```
private String foo(Import ^import) : ^import.name;
```

4. Check

4.1. Description of the *Check* language

Xpand 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 7. 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;
```

- 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.

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.

4.1.1. Guard Conditions

The *Check* language of *Xpand* also provides so called . These conditions allow to apply a check constraint only to model elements that meet certain criteria. Specifying such a guard condition is done by adding an *if* clause to the check constraint. The *if* clause has to be added after the *context* clause as demonstrated by the following example:

```
import data;
context Attribute if name.length > 1 ERROR
  "Attribute names have to start with an 'a'" :
  name.startsWith("a");
```

4.2. The workflow component *CheckComponent*

The workflow component *CheckComponent* allows to integrate model validation constraints using the *Check* into a modeling workflow using MWE.

This component provides the following configuration properties:

Table 8. Properties

Name of property	Description
checkFile	This property allows to add files containing constraints written in the <i>Check</i> language to the validation component. This property only works in conjunction with EMF based models.
expression	This property allows to set a check expression for the validation component. This property only works in conjunction with non-EMF based models.
abortOnError	This boolean property determines if the workflow will be aborted or not if one of the validation constraints fails.
warnIfNothingChecked	If this boolean property will be set to <i>true</i> , a warning will be generated if there were no validation checks. Otherwise, no warning will be issued.

5. Xtend

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

This language provides the possibility to define rich libraries of independent operations and non-invasive metamodel extensions based on either Java methods or *Xtend* expressions. Those libraries can be referenced from all other textual languages, that are based on the expressions framework.

5.1. Xtend files

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

5.2. 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 */
```

5.3. 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!
```

5.4. Extension Import Statement

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

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

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

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. 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();
```

5.5.1. 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.

5.5.2. 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() ->
    {}
;
```

5.5.3. 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.

5.5.4. 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 getName(NamedElement ele) :
    'get'+ele.name.firstUpper()
;
```

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

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...
;
```

5.6. 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.someMethod(java.lang.String)
;
```

The signature is the same as for any other extension. Its syntax is:

```
JAVA fully.qualified.Type.someMethod(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)`

5.6.1. static vs non-static invocation

The implementation of a Java extension is redirected to a public method in a Java class. If the method is not declared static it is required that the Java class has a default constructor. Xtend will instantiate the class for each invocation.

5.6.2. IExecutionContextAware

It is possible with a Java Extension to gain access to the ExecutionContext, which enables to retrieve detailed runtime information on invocation. To use the current ExecutionContext in a Java extension the class must implement `org.eclipse.xtend.expression.IExecutionContextAware`

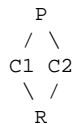
```
public interface IExecutionContextAware {
    void setExecutionContext (ExecutionContext ctx);
}
```

The invoked method must not be static.

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:



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

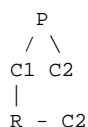
We could write the following extensions in order to transform an Ecore (EMF) model to our metamodel (Package, Class, Reference).

```
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:



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.

5.8. 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
;
```

5.9. 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="org.eclipse.xtend.XtendComponent">
    <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
        <metaModelFile value="metamodel1.ecore"/>
    </metaModel>
    <metaModel class="org.eclipse.xtend.typesystem.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).

5.10. Aspect-Oriented Programming in Xtend

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 a non-invasive way (you do not need to touch the packaged extensions).

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.

5.10.1.1. 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::eclipse::xpand2::* //extensions prefixed with 'org::eclipse::xpand2::'  
*Operation* // extensions containing the word 'Operation' in it.  
* // all extensions
```

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

5.10.1.2. 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
```

5.10.1.3. 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.

5.10.2. 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="org.eclipse.xtend.XtendComponent">  
  <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">  
    <metaModelFile value="metamodel1.ecore"/>  
  </metaModel>  
  <metaModel class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">  
    <metaModelFile value="metamodel2.ecore"/>  
  </metaModel>  
  
  <invoke value="my::example::Trafo::transform(inputSlot)"/>  
    <outputSlot value="transformedModel"/>  
    <advices value="my::Advices,my::Advices2"/>  
</component>
```

5.10.3. Model-to-Model transformation with Xtend

This example uses Eclipse EMF as the basis for model-to-model transformations.

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.

5.10.4. 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="org.eclipse.xtend.XtendComponent">
    <metaModel class="org.eclipse.xtend.typesystem.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.

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.

6. Xpand2

The *Xpand* language 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**+>.

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.

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).

6.3. Statements of the Xpand language

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.*`.

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 *Xtend* 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.

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  
«ENDDDEFINE»
```

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".

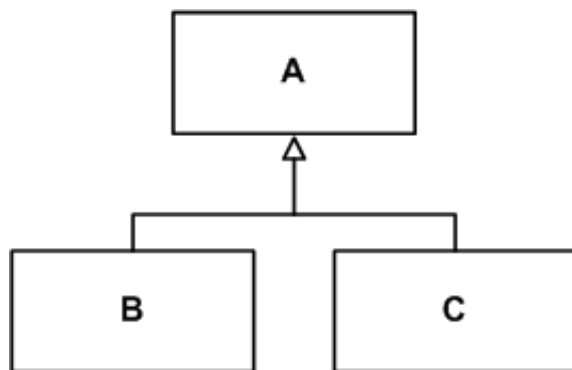
As in Java, a formal parameter list entry consists of the type followed by the name of that parameter.

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 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»  
«ENDDDEFINE»  
  
«DEFINE implClass FOR A»
```

```
// this is the code generated for the superclass A
«ENDDDEFINE»

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

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

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.

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»
```

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] ] [ONFILECLOSE]»
```

The various alternative syntaxes are explained below.

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»
```

6.3.5.2. Lazy evaluation

Appending the ONFILECLOSE statement defers evaluation of the expanded definition until the current file is closed with ENDFILE. This is of use when the state required to create the text is collected during the evaluation of the processed definition.

```
«FILE ...»
...
«EXPAND LazyEvaluatedDefinition FOREACH myCollection ONFILECLOSE»
...
«ENDFILE» «REM»Now 'LazyEvaluatedDefinition' is called«ENDFILE»
```

A typical example for usage of the ONFILECLOSE statement is when you want to create a list of imports in a Java class, but the types that are used should be added when they are used in the templates later.

The state, usually a collection, that is used for the lazy expanded evaluation must be valid until the file is closed. This can be achieved in two ways:

- Span a LET statement around the FILE statement that bounds an empty collection

```
«LET (List[MyType]) {} AS importedTypes»
«FILE ...»
...
«EXPAND ImportStatement FOREACH importedTypes ONFILECLOSE»
...
«importedTypes.add(someType) -> ""-»
...
«ENDFILE»
«ENDLET»
```

- Use a *create* Extension which returns an empty collection and append elements to it. Since it is a create extension the empty collection will be returned on first call and for each subsequent call a reference to this collection will be returned and not a new collection created.

some/path/InsertionPoints.ext:

```
create List[Type] importedTypes (SomeType context) : (List[Type]) {};
```

In Xpand use this as follows:

```
«EXTENSION some::path::InsertionPoints»
«FILE ...»
...
«EXPAND ImportStatement FOREACH importedTypes() ONFILECLOSE»
...
«importedTypes().add(someType) -> ""-»
...
«ENDFILE»
«ENDLET»
```

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»
```

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.

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
```

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»
```

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»
```

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 fqcn»
    the fully qualified name is: «fqcn»;
«ENDLET»
```

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.

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»
```

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).

6.3.14. Controlling generation of whitespace

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.

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 a 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).

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.

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::eclipse::xpand2::* // definitions prefixed with 'org::eclipse::xpand2::'
*Operation* // definitions containing the word 'Operation' in it.
* // all definitions
```

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
```

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
```

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.

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. A simple generator component configuration could look as follows:

```
<component class="org.eclipse.xpand2.Generator">
  <fileEncoding value="ISO-8859-1"/>
  <metaModel class="org.eclipse.xtend.typesystem.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="org.eclipse.xpand2.output.JavaBeautifier"/>
  <beautifier class="org.eclipse.xtend.typesystem.xsd.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.

6.5.1. Main configuration

The first thing to note is that the qualified Java name of the component is `org.eclipse.xpand2.Generator`.

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.

In a team that uses different operating systems or locales it is a good idea to set the file encoding fixed for the *Xpand* project and share the settings. Typical encodings used are UTF-8 or ISO-8859-1, but any encoding having guillemet brackets is fine also.²

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 *Xpand* 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.

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="org.eclipse.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'`.

²On Mac OSX the default encoding is MacRoman, which is not a good choice, since other operating systems are not aware of this encoding. It is recommended to set the encoding to some more common encoding, e.g. UTF-8, maybe even for the whole workspace.

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.

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.eclipse.xpand2.output.JavaBeautifier"/>
<beautifier
  class="org.eclipse.xtend.typesystem.xsd.XMLBeautifier"/>
```

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

```
package org.eclipse.xpand2.output;

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

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

6.5.5.1. JavaBeautifier

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

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.

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"/>
```

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.

6.5.7. VetoStrategy

The *Xpand* engine will generate code for each processed FILE statement. This implies that files are written that might not have changed to the previous generator run. Normally it does not matter that files are rewritten. There are at least two good reasons when it is better to avoid rewriting of files:

1. The generated source code will be checked in. In general it is not the recommended way to go to check in generated code, but sometimes you will have to. Especially with CVS there is the problem that rewritten files are recognized as modified, even if they haven't changed. So the problem arises that identical files get checked in again and again (or you revert it manually). When working in teams the problem even becomes worse, since team members will have conflicts when checking in.
2. When it can be predicted that the generator won't produce different content before a file is even about to be created by a FILE statement then this can boost performance. Of course it is not trivial to predict that a specific file won't result in different content before it is even created. This requires information from a prior generator run and evaluation against the current model to process. Usually a diff model would be used as input for the decision.

Case 1) will prevent file writing after a FILE statement has been evaluated, case 2) will prevent creating a file at all.

To achieve this it is possible to add Veto Strategies to the generator, which are implementations of interface `org.eclipse.xpand2.output.VetoStrategy` or `org.eclipse.xpand2.output.VetoStrategy2`. Use `VetoStrategy2` if you implement your own.

`VetoStrategy2` declares two methods:

- `boolean hasVetoBeforeOpen (FileHandle)`

This method will be called before a file is being opened and generated. Return true to suppress the file creation.

- `boolean hasVeto (FileHandle)`

This method will be called after a file has been produced and after all configured PostProcessors have been invoked. Return true to suppress writing the file.

Veto Strategies are configured per Outlet. It is possible to add multiple stratgy instances to each Outlet.

```
<component id="generator" class="org.eclipse.xpand2.Generator" skipOnErrors="true">
  <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel"/>
  <expand value="templates::Root::Root FOR model"/>
  <fileEncoding value="ISO-8859-1"/>
  <outlet path="src-gen">
    <postprocessor class="org.eclipse.xpand2.output.JavaBeautifier"/>
    <vetoStrategy class="org.eclipse.xpand2.output.NoChangesVetoStrategy"/>
  </outlet>
</component>
```

One `VetoStrategy` is already provided. The `org.eclipse.xpand2.output.NoChangesVetoStrategy` is a simple implementation that will compare the produced output, after it has been postprocessed, with the target file. If the content is identical the strategy vetoes the file writing. This strategy is effective, but has two severe drawbacks:

1. The file has been created at least in memory before. This consumes time and memory. If applying code formatting this usually implies that the file is temporarily written.
2. The existing file must be read into memory. This also costs time and memory.

Much better would be to even prevent the creation of files by having a valid implementation for the `hasVetoBeforeOpen()` method. Providing an implementation that predicts that files do not have to be created requires domain knowledge, thus a standard implementation is not available.

The number of skipped files will be reported by the Generator component like this:

```
2192 INFO - Generator(generator): generating <...>
3792 INFO - Skipped writing of 2 files to outlet [default](src-gen)
```

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.

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.

6.8. Example

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

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 `org.eclipse.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 baseClassName() »
    // 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 accomplish is this: 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 a given framework. The specific code at this place 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**.

6.8.2. Workflow File

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

```
<workflow>  
  <cartridge file="workflow.mwe"/>  
  <component adviceTarget="generator"  
    id="reflectionAdvice"  
    class="org.eclipse.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.

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;  
    }  
}
```

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::eclipse::xpand2::* FOR Entity»
```

matches all templates with namespace *org::eclipse::xpand2::* 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»
```

7. Profiler

The *ProfilerComponent* allows you to measure the time each check, Xtend function or Xpand define needed to be executed in your workflow. It does so by acting as a callback of the *CheckComponent*, *XtendComponent* and *XpandComponent* where the time of each call will be stored in a profiling model. From this data the profiler eventually derives execution times with and without children, callers/callees with the corresponding call counts and finally a call graph with respect to recursive calls. A set of Xpand templates can transform the model to an HTML report or a text file in the GNU GProf format.

Figure 2. Sample HTML output of the profiler

Profiling Results

This profiler result is an adaption of the *GNU gprof* format and has been produced with the *Xpand/Xtend profiler*. Please note that the measured results depend on the *runtime configuration* the templates, checks, and/or expressions have been executed on.

Flat Profile

The following table shows the totals of different measured values. Use *these numbers* to get a first impression of the execution times.

% time	cum. sec.	self sec.	calls	self ms/call	total ms/call	name
86.96	0.89	0.89	2	446.16	447.59	XPD template::Template::javaClass FOR Entity
10.83	1.00	0.11				CHK Entity "Duplicate entity "+(name)
1.14	1.02	0.01				CHK Model "No entities defined"
0.65	1.02	0.01				XPD template::Template::main FOR Model
0.16	1.02	0.00	6	0.27	0.27	XTD template::GeneratorExtensions::setter(metamodel::Feature)
0.13	1.02	0.00	3	0.43	0.43	XTD metamodel::Extensions::entities(metamodel::Model)
0.12	1.03	0.00	6	0.21	0.21	XTD template::GeneratorExtensions::getter(metamodel::Feature)
0.02	1.03	0.00	2	0.10	0.10	XTD metamodel::Extensions::model(metamodel::Type)

Runtime Configuration

The above results are measured against the system time and may vary per execution. Since the runtime configuration and dimensions such as processor clock or heap size greatly influence the execution time you also might consider improving the configuration to improve the results. This sections may give you some hints to do so.

Unfortunately, this report will only provide the following values so far:

Used Memory	39 MB	Please note that this value has been measured at the time of generating this report.
Max Heapsize	254 MB	

Call Graph

The following table represents the *call graph* table similar to GProf. The call graph shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.

#	% time	self	children	called	name
					<spontaneous>
[1]	88.0	0.01	0.90		XPD template::Template::main FOR Model [1]
	0.00	0.00	1 / 3		XTD metamodel::Extensions::entities(metamodel::Model)
	0.89	0.00	2 / 2		XPD template::Template::javaClass FOR Entity
	0.89	0.00	2 / 2		XPD template::Template::main FOR Model [1]
[2]	87.2	0.89	0.00	2	XPD template::Template::javaClass FOR Entity [2]
	0.00	0.00	6 / 6		XTD template::GeneratorExtensions::setter(metamodel::Feature)
	0.00	0.00	6 / 6		XTD template::GeneratorExtensions::getter(metamodel::Feature)
					<spontaneous>
[3]	10.9	0.11	0.00		CHK Entity "Duplicate entity "+(name) [3]
	0.00	0.00	2 / 2		XTD metamodel::Extensions::model(metamodel::Type)
	0.00	0.00	2 / 3		XTD metamodel::Extensions::entities(metamodel::Model)
					<spontaneous>
[4]	1.1	0.01	0.00		CHK Model "No entities defined" [4]
	0.00	0.00	6 / 6		XPD template::Template::javaClass FOR Entity [2]
[5]	0.2	0.00	0.00	6	XTD template::GeneratorExtensions::setter(metamodel::Feature) [5]
	0.00	0.00	2 / 3		CHK Entity "Duplicate entity "+(name) [3]
	0.00	0.00	1 / 3		XPD template::Template::main FOR Model [1]
[6]	0.1	0.00	0.00	3	XTD metamodel::Extensions::entities(metamodel::Model) [6]
	0.00	0.00	6 / 6		XPD template::Template::javaClass FOR Entity [2]
[7]	0.1	0.00	0.00	6	XTD template::GeneratorExtensions::getter(metamodel::Feature) [7]
	0.00	0.00	2 / 2		CHK Entity "Duplicate entity "+(name) [3]
[8]	0.0	0.00	0.00	2	XTD metamodel::Extensions::model(metamodel::Type) [8]

Use the *ProfilerComponent* to wrap other components inside a workflow. Denote a *resultSlot* where the profiler stores the model in the end. Then, refer to this component via *idRef* as a callback. As soon as the component-tag of the profiler closes, it stores the derived profiling model in the given slot. You are free to transform this model or you can re-use one of the templates that come with profiler.

```
<component id="profiler" class="org.eclipse.xtend.profiler.ProfilerComponent">
  <resultSlot value="profilingResult" />

  <component class="org.eclipse.xtend.check.CheckComponent">
    <vetoableCallback idRef="profiler" />
    ...
  </component>
  <component class="org.eclipse.xtend.XtendComponent">
    <vetoableCallback idRef="profiler" />
    ...
  </component>
  <component class="org.eclipse.xpand2.Generator">
    <vetoableCallback idRef="profiler" />
    ...
  </component>
</component>

<component class="org.eclipse.xpand2.Generator" fileEncoding="ISO-8859-1">
  <metaModel idRef="mm"/>
  <expand value="org::eclipse::xtend::profiler::templates::Html::Main FOR profilingResult"/>
  <outlet overwrite="true" path="profiling"/>
</component>
```

Currently, there are two different templates available to render the profiling model

- org::eclipse::xtend::profiler::templates::Html::Main
- org::eclipse::xtend::profiler::templates::GProf::Main

The Xpand Wizard will produce a workflow called *workflowWithProfiler.mwe* that demonstrates the capabilities of the profiler. It puts the result in the folder *profiling*.

Chapter 3. Built-in types API documentation

1. Object

Supertype: none

Table 1. Properties

Type	Name	Description
<code>xpand2::Type</code>	<code>metaType</code>	returns this object's meta type.

Table 2. Operations

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

2. string

Supertype: Object

Table 3. Properties

Type	Name	Description
Integer	<code>length</code>	the length of this string

Table 4. Operations

Return type	Name	Description
String	<code>toLowerCase()</code>	Converts all of the characters in this String to lower case using the rules of the default locale (from Java)
String	<code>+(Object)</code>	concatenates two strings
List	<code>toCharList()</code>	splits this String into a List[String] containing Strings of length 1

Return type	Name	Description
String	toFirstUpper ()	Converts the first character in this String to upper case using the rules of the default locale (from Java)
String	substring (Integer, Integer)	Returns a new string that is a substring of this string.
String	trim ()	Returns a copy of the string, with leading and trailing whitespace omitted. (from Java 1.4)
String	toFirstLower ()	Converts the first character in this String to lower case using the rules of the default locale (from Java)
String	toUpperCase ()	Converts all of the characters in this String to upper 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)
Boolean	startsWith (String)	Tests if this string starts with the specified prefix.
Boolean	matches (String)	Tells whether or not this string matches the given regular expression. (from Java 1.4)
Integer	asInteger ()	Returns an Integer object holding the value of the specified String (from Java 1.5)
Boolean	contains (String)	Tests if this string contains substring.
Boolean	endsWith (String)	Tests if this string ends with the specified prefix.
String	replaceFirst (String, String)	Replaces the first substring of this string that matches the given regular expression with the given replacement.
String	replaceAll (String, String)	Replaces each substring of this string that matches the given regular expression with the given replacement.

3. Integer

Supertype: Real

This type does not define any properties.

Table 5. Operations

Return type	Name	Description
List	upTo (Integer)	returns a List of Integers starting with the value of the target expression, up to the value of the specified Integer, incremented by one, e.g. '1.upTo(5)' evaluates to {1,2,3,4,5}

Return type	Name	Description
Boolean	>= (Integer)	
Boolean	== (Integer)	
Boolean	!= (Integer)	
List	upTo (Integer, Integer)	returns a List of Integers starting with the value of the target expression, up to the value of the first paramter, incremented by the second parameter, e.g. '1.upTo(10, 2)' evaluates to {1,3,5,7,9}
Integer	- (Integer)	
Integer	+ (Integer)	
Boolean	<= (Integer)	
Boolean	< (Integer)	
Integer	* (Integer)	
Integer	- ()	
Boolean	> (Integer)	
Integer	/ (Integer)	

4. Boolean

Supertype: Object

This type does not define any properties.

Table 6. Operations

Return type	Name	Description
Boolean	! ()	

5. Real

Supertype: Object

This type does not define any properties.

Table 7. Operations

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

6. Collection

Supertype: Object

Table 8. Properties

Type	Name	Description
Boolean	isEmpty	returns true if this Collection is empty
Integer	size	returns the size of this Collection

Table 9. Operations

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

7. List

Supertype: Collection

This type does not define any properties.

Table 10. Operations

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

8. Set

Supertype: Collection

This type does not define any properties.

This type does not define any operations.

9. xpanse2::Type

Supertype: Object

Table 11. Properties

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

Table 12. Operations

Return type	Name	Description
xpanse2::StaticProperty	getStaticProperty (String)	
xpanse2::Feature	getFeature (String, List)	
Boolean	isInstance (Object)	
xpanse2::Property	getProperty (String)	
Object	newInstance ()	
Boolean	isAssignableFrom (xpanse2::Type)	
xpanse2::Operation	getOperation (String, List)	

10. `xpand2::Feature`

Supertype: `Object`

Table 13. Properties

Type	Name	Description
<code>String</code>	<code>name</code>	
<code>xpand2::Type</code>	<code>returnType</code>	
<code>String</code>	<code>documentation</code>	
<code>xpand2::Type</code>	<code>owner</code>	

This type does not define any operations.

11. `xpand2::Property`

Supertype: `xpand2::Feature`

This type does not define any properties.

Table 14. Operations

Return type	Name	Description
<code>Void</code>	<code>set (Object, Object)</code>	
<code>Object</code>	<code>get (Object)</code>	

12. `xpand2::Operation`

Supertype: `xpand2::Feature`

This type does not define any properties.

Table 15. Operations

Return type	Name	Description
<code>List</code>	<code>getParameterTypes ()</code>	
<code>Object</code>	<code>evaluate (Object, List)</code>	

13. `xpand2::StaticProperty`

Supertype: `xpand2::Feature`

This type does not define any properties.

Table 16. Operations

Return type	Name	Description
<code>Object</code>	<code>get ()</code>	returns the static value

14. `Void`

Supertype: `Object`

This type does not define any properties.

This type does not define any operations.

15. `xtend::AdviceContext`

Supertype: `Object`

Table 17. Properties

Type	Name	Description
List	paramTypes	
String	name	
List	paramNames	
List	paramValues	

Table 18. Operations

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

16. xpanse2::Definition

Supertype: Object

Table 19. Properties

Type	Name	Description
List	paramTypes	
String	name	
List	paramNames	
xpanse2::Type	targetType	

Table 20. Operations

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

17. xpanse2::Iterator

Supertype: Object

Table 21. Properties

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

This type does not define any operations.

Chapter 4. Stdlib

1. Introduction

Xpand delivers a set of small useful utility extensions and components in the `org.eclipse.xpand.util.stdlib` package. You need to add a dependency to this plugin if you want to use it.

2. Stdlib extensions

This section describes the components and extensions provided by Stdlib. We use the shortcut `oaw.util.stdlib...` for component classes in package `org.eclipse.xtend.util.stdlib` in workflow configurations for convenience.

Note that many functions of the Stdlib make use of static variables in their Java implementation, thus the values are kept through a complete MWE workflow. Also, because of the static implementation, the features are not threadsafe.

2.1. IO extensions

This is an extremely useful library to print information to the logging facility. It is really valuable through transformation processes or for complex expressions to know what exactly expressions are evaluated to.

Extension: `org::eclipse::xtend::util::stdlib::io`

2.1.1. debug (Object o)

Logs an object with DEBUG level to the logger.

Parameters:

- `o` - The object to dump.

Returns: The object `o`

2.1.2. info (Object o)

Logs an object with INFO level to the logger.

Parameters:

- `o` - The object to dump.

Returns: The object `o`

2.1.3. error (Object o)

Logs an object with ERROR level to the logger.

Parameters:

- `o` - The object to dump.

Returns: The object `o`

2.1.4. syserr (Object o)

Prints an object to `System.err`.

Parameters:

- `o` - The object that should be printed. `null` is allowed.

Returns: The object `o`

2.1.5. syserr (Object o, String prefix)

Prints an object to `System.err`.

Parameters:

- `o` - The object that should be printed. `null` is allowed.
- `prefix` - A prefix string for the message.

Returns: The object `o`

2.1.6. syserr (Object o)

Prints an object to System.err.

Parameters:

- o - The object that should be printed. null is allowed.

Returns: The object o

2.1.7. throwError (Object o)

Throws an IllegalStateException.

Parameters:

- o - The exception message.

Returns: Nothing, since an exception is thrown.

2.1.8. Examples

```
import data;
extension org.eclipse.xtend.util.stdlib::io;
create DataModel this duplicate(DataModel s):
    entity.addAll( s.entity.duplicate() ) ->
        setName(s.name);

create Entity this duplicate(Entity old):
    (old.name+" has "+old.reference.size+" references").info() ->
        old.reference.name.info() ->
```

This leads to the following output on the console:

```
922 INFO - Person has 1 references
923 INFO - [autos]
926 INFO - Vehicle has 0 references
926 INFO - []
```

Of course IO extension functions can also be used within Xpand, but if used for logging purposes you have to deal with one side effect: Since the functions return the passed object (the result of an expression, in the simplest case just a string) and Xpand prints out expression results to the opened file, the message will be shown on the console, but also be in the result file. This you might want to avoid, so you can use a small trick for this: after calling a log function use the chaining operator and let the result of the expression be an empty string:

```
«EXTENSION org::eclipse::xtend::util::stdlib::io»
...
«DEFINE javaClass FOR Entity»
«REM»The following expression will dump the feature names without producing output
as side effect«ENDREM»
«features.name.info() -> ""»
```

This will produce this output on the console:

```
1122 INFO IOExtensions - [name, age, address]
1740 INFO IOExtensions - [street, zip, city]
```

Each function returns the object on which they have been called, so you can build chain expressions. Or, in other words, if you have some expression like

```
element.x.y.z.select(t|t.someProp).a
```

you can always embed one of these io functions anywhere such as in

```
element.x.syserr().y.z.select(t|t.someProp.info()).a
```

2.1.9. Controlling the log level

You may want to control the logging level for the messages which are printed via the logging facility. How this is configured in detail depends on the underlying logging framework. Xpand uses the Apache Commons Logging library, which may dispatches to another logging framework, mostly Log4J.

To control the logging level exactly for the IO extensions you have to know the category to which the messages are logged to. It is common to use the class names of the classes that use the logger. In the case of the IO extensions this class is `org.eclipse.xtend.util.stdlib.IOExtensions`.

The following example shows a Log4J configuration file which would disable log levels below warning. This example would only work if the properties file is found at the beginning of the classpath. Make sure that the file would be found before any other Log4J configurations on your classpath. The file must be named `log4j.properties`.

```
log4j.appender.CONSOLE = org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout = org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern = %p %C{1} %m %n
log4j.rootLogger = INFO, CONSOLE
# suppress info messages from IOExtensions
log4j.logger.org.eclipse.xtend.util.stdlib.IOExtensions=WARN, CONSOLE
log4j.additivity.org.eclipse.xtend.util.stdlib.IOExtensions=false
```

2.2. Counter extensions

Sometimes it is necessary to have counters within transformation code. The counter extensions enable to initialize, manipulate and retrieve counters.

Extension: `org::eclipse:xtend::util::stdlib::counter`

2.2.1. `int counterInc (Object o)`

Increments a counter.

Parameters:

- `o` - A key for this counter. If this function is called with a `null` argument an anonymous counter is used. If no counter was registered for the key a new counter instance will be created and initialized with 0.

Returns: The incremented counter.

2.2.2. `int counterDec (Object o)`

Decrements a counter.

Parameters:

- `o` - A key for this counter. If this function is called with a `null` argument an anonymous counter is used. If no counter was registered for the key a new counter instance will be created and initialized with 0.

Returns: The decremented counter.

2.2.3. `int counterReset (Object o)`

Resets a counter.

Parameters:

- `o` - A key for this counter. If this function is called with a `null` argument an anonymous counter is used. If no counter was registered for the key a new counter instance will be created and initialized with 0.

Returns: Always 0.

2.2.4. `int counterGet (Object o)`

Retrieves the current state of a counter.

Parameters:

- `o` - A key for this counter. If this function is called with a `null` argument an anonymous counter is used.

Returns: Current counter value.

2.2.5. Example

```
«DEFINE CounterExtensionsDemo FOR Object»
«FILE "CounterExtensions.txt"»
  First counter:
    get          : «counterGet()»
    inc          : «counterInc()»
```

```

inc      : «counterInc()»
inc      : «counterInc()»
dec      : «counterDec()»
Second (named) counter:
inc      : «counterInc("idx")»
inc      : «counterInc("idx")»
inc      : «counterInc("idx")»
reset    : «counterReset("idx")»
inc      : «counterInc("idx")»
First counter:
inc      : «counterInc()»

«ENDFILE»
«ENDDEFINE»

```

This example will create the following output:

```

First counter:
get       : 0
inc       : 1
inc       : 2
inc       : 3
dec       : 2
Second (named) counter:
inc       : 1
inc       : 2
inc       : 3
reset     : 0
inc       : 1
First counter:
inc       : 3

```

2.3. Properties extensions

You might want to specify configuration values from properties files from your transformation code. The Properties extensions can help you there. Before being able to access the properties through an extension function the properties files must be read and its values stored. This is done through the workflow component *PropertiesReader*, which is described below.

Extension: `org::eclipse::xtend::util::stdlib::properties`

2.3.1. String getProperty (String key)

Retrieves a configuration property.

Parameters:

- `o` - Property key

Returns: Property value, if defined, else null.

2.3.2. Workflow component

The workflow component *PropertiesReader* is used to load properties files. It is possible to configure multiple properties files by adding the `propertiesFile` tag multiple times.

Table 1. Workflow component `org.openarchitectureware.util.stdlib.PropertiesReader`

Property	Type	Mandatory	Description
<code>propertiesFile</code>	String	yes	The properties file to read

2.3.3. Example

Workflow configuration:

```

<component class="org.eclipse.xtend.util.stdlib.PropertiesReader">
  <propertiesFile value="src/config1.properties"/>
  <propertiesFile value="src/config2.properties"/>
</component>

```

`config1.properties:`

```
shapes = box,polygon,ellipse,point
```

Usage in an extension:

```
extension org::eclipse::xtend::util::stdlib::properties;

cached List[String] SHAPES () : getProperty("shapes").split(",").trim();
```

2.4. Element properties extensions

This allows you to temporarily associate name-value pairs with any model element.

Extension: `org::eclipse::xtend::util::stdlib::elementprops`

2.4.1. Void setProperty (Object element, String name, Object value)

Sets the property named name to the value.

Parameters:

- `element` - The model element
- `name` - Property name
- `element` - The property value

Returns: Nothing.

2.4.2. Object getProperty (Object element, String name)

Retrieves a dynamic property from an element.

Parameters:

- `element` - The model element
- `name` - Property name

Returns: The property value.

2.5. Issues extensions

In template code there is no direct access to the Issues instance of the workflow's context possible. The Issues extensions help to report warnings and errors to the Issues instance during transformation.

This should not encourage you to use constraint checking and generally raise errors directly from within the transformations. However, sometimes it is sensible and useful to be able to do that.

Extension: `org::openarchitectureware::util::stdlib::issues`

2.5.1. String reportWarning (String message)

Reports a warning message to the workflow context.

Parameters:

- `message` - A message

Returns: The message.

2.5.2. String reportWarning (Object object, String message)

Reports a warning message and the qualified name of a context object to the workflow context.

Parameters:

- `object` - A context object
- `message` - A message

Returns: The message.

2.5.3. String reportError (String message)

Reports a error message to the workflow context.

Parameters:

- message - A message

Returns: The message.

2.5.4. String reportError (Object object, String message)

Reports a error message and the qualified name of a context object to the workflow context.

Parameters:

- object - A context object
- message - A message

Returns: The message.

2.5.5. Workflow component

The Issues extensions require that the workflow component `org.eclipse.xtend.util.stdlib.ExtIssueReporter` is configured in the workflow before calling the extensions. The purpose of this component is make the workflow's Issues instance available for the extensions.

The `ExtIssueReporter` component does not have any properties.

2.5.6. Example

Workflow configuration:

```
<?xml version="1.0"?>
<workflow>
  ...
  <component class="oaw.util.stdlib.ExtIssueReporter"/>
```

Using from Xtend:

```
import metamodel;
extension org::openarchitectureware::util::stdlib::issues;

demo (Model this) :
  issuesExtensionsDemo()
  ;

issuesExtensionsDemo () :
  reportWarning("Reporting a warn message from Xtend to the workflow");
```

Console output:

```
INFO WorkflowRunner running workflow: workflow/generator.oaw
...
...
INFO CompositeComponent ExtIssueReporter: setting up issue logging from
  within .ext and .xpt files
INFO WorkflowRunner workflow completed in 1101ms!
WARN WorkflowRunner Reporting a warn message from Xtend to the workflow
```

2.6. Naming extensions

The Naming extensions are only usable with EMF models. This one helps with names, qualified names and namespaces. A qualified name is defined as the sequence of primitive names of the containment hierarchy of an element, separated by a dot (e.g. `java.lang.String`). In order for this to work, model elements are expected to have a name attribute of type `EString`.¹

Extension: `org::eclipse::xtend::util::stdlib::naming`

2.6.1. String namespace (Object this)

Returns the namespace, i.e. the qualified name minus the name of the element itself.

Parameters:

- this - A model element

¹It is intended that the `uml2ecore` utility can add such a name attribute to every meta class automatically.

Returns: The qualified namespace name of the element.

2.6.2. String qualifiedName (Object this)

Returns the qualified name (dot separated) of an element by evaluating its containment hierarchy.

Parameters:

- `this` - A model element

Returns: The qualified name of the element.

2.6.3. String loc (Object this)

Tries to build a useful description of an element in the model; very useful for error reporting.

Parameters:

- `this` - A model element

Returns: Location information about the element.

2.6.4. Object findByName (Collection candidates, String name)

Searches the candidates for an element with a specific name.

Parameters:

- `candidates` - A collection of model elements
- `name` - The searched element name

Returns: The searched element or null if no element with that name is contained in the candidates collection.

2.7. Globalvar extensions

Sometimes you might want to share information within a transformation process. One alternative is the use of GLOBALVAR expressions, but this needs that the variables are configured in the workflow. The Globalvar extensions help to store and retrieve objects within a transformation process.

Extension: `org::openarchitectureware::util::stdlib::globalvar`

2.7.1. Example

Usage in Xtend:

```
import metamodel;
extension org::openarchitectureware::util::stdlib::io;
extension org::openarchitectureware::util::stdlib::globalvar;

demo (Model this) :
    globalvarExtensionsDemo1() ->
    globalvarExtensionsDemo2()
;

globalvarExtensionsDemo1 () :
    "Storing global var...".info() ->
    storeGlobalVar("msg", "oAW is cool stuff!");

globalvarExtensionsDemo2 () :
    ("Getting message from global var: "+getGlobalVar("msg")).info();
```

Console output:

```
INFO IOExtensions Storing global var...
INFO IOExtensions Getting message from global var: oAW is cool stuff!
```

This is a simple example storing a string, but of course you can store the result of any expression this way.

2.8. Cloning extensions

The cloning utilities help you to clone a model element and all its children. The `clone(Object)` function clones a single object and its children, whereas the `clone(List)` clones a list of elements. The semantics of cloning is as follows:

- the object passed in as a parameter is duplicated
- all objects referenced via containment references are also duplicated, recursively
- the values of the attributes are duplicated
- non-containing references to other objects are copied while the target is not cloned (a reference to the original is created in the new object)

Extension: org::eclipse::xtend::util::stdlib::cloning

2.8.1. Object clone (Object original)

Clones an object.

Parameters:

- original - The object that should be cloned.

Returns: The cloned object.

2.8.2. List clone (List l)

Clones a list of objects.

Parameters:

- l - Source list.

Returns: The list of cloned objects.

2.9. Cross references extensions

Sometimes there is the need to find objects that reference a specific object. This extension helps to solve this recurring task. This extension can only be used for EMF based models.

Extension: org::eclipse::xtend::util::stdlib::crossref

2.9.1. List[EObject] getReferencingObjects(EObject target)

Retrieves objects that reference a given object.

Parameters:

- target - The target object.

Returns: A list of objects referencing the target.

2.9.2. Example

Usage in Xtend:

```
extension org::openarchitectureware::util::stdlib::crossref;
crossRefDemo (Model this) :
    eAllContents.typeSelect(Datatype).dumpCrossReferences();

dumpCrossReferences (Datatype this) :
    ("Number of cross references to datatype "+name+":"
    + getReferencingObjects().size)
    .info()
    ;
```

Console output:

```
INFO IOExtensions Number of cross references to datatype Integer:1
INFO IOExtensions Number of cross references to datatype String:4
```

2.10. UID extensions

Often it is required to create and retrieve unique identifiers for objects through the transformation process. The UID extensions provide a simple mechanism for this task. Unique identifiers are calculated from the current system time plus an internal counter. The extensions therefore only guarantee that the identifier stays the same within one workflow execution, but will change through different runs. If you need to have unique identifiers that stay the same over every generation run (e.g. for Protected Regions Ids) then you need another mechanism.

If you are loading the model that assigns IDs to EObject (only for EMF based models) the `xmlId()` function will be useful. Especially when using UML2 models this function will return a unique and non-changing identifier for objects.

Extension: `org::openarchitectureware::util::stdlib::uid`

2.10.1. cached String uid(Object o)

Retrieves a unique identifier for an object. Creates a new one on first access.

Parameters:

- `o` - A model element or other object.

Returns: The UID for this object

2.10.2. String createUID(Object o)

Creates a unique identifier for an object.

Parameters:

- `o` - A model element or other object.

Returns: A newly created UID for this object.

2.10.3. String xmlId(ecore::EObject o)

Retrieves an object's identifier. The object must be read from a XMLResource.

Parameters:

- `o` - An object.

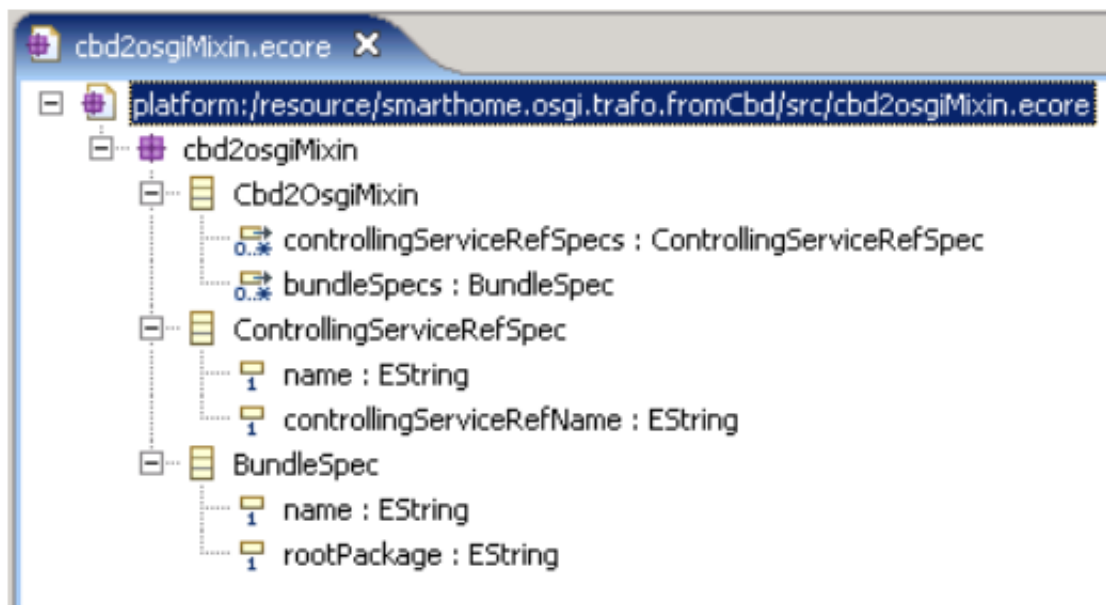
Returns: The object's id. Returns null if the object was not load from a XMLResource.

2.11. Mixin extensions

These utilities help with mixin models. Mixin models are typically simple models that provide additional information about model elements in a source model of a transformation. They can be seen as annotations.

These utilities expect that the mixin models have a very specific structure: A root element, and then any subtree, where the elements have a name attribute. Here's an example:

Figure 1. Mixin model example



The mixin elements are `ControllingServiceRefSpec` and `BundleSpec`. They are owned by the root element, `Cbd2OsgiMixin`. The name is expected to contain the qualified name of the element the annotation

refers to. Once the model is set up like this, and made available to a transformation using the workflow's GLOBALVAR facilities, you can then use the extension functions.

Extension: `org::eclipse::xtend::util::stdlib::mixin`

2.11.1. Object `getMandatoryMixin(Object mixinModel, Object ctx, oaw::Type t)`

Returns the corresponding mixin element for the context object; the mixin must be of type `t` and its name attribute must correspond to the qualified name of the context. If none is found, a workflow ERROR is raised and a null object is returned (so you can call additional operations on it without getting a null evaluation error).

Parameters:

- `mixinModel` - The root element of the mixin model.
- `ctx` - The context object.
- `t` - The type of the mixin model element.

Returns: The mixin model element corresponding to `ctx`.

2.11.2. Object `getOptionalMixin(Object mixinModel, Object ctx, oaw::Type t)`

Same as `getMandatoryMixin()`, but does not raise an error in case nothing is found.

2.12. Tracing extensions

The tracing extensions allow to create trace paths during your model transformations. This is done by creating a trace model which holds references from source to target elements. Traces must be added explicitly to the transformation code.

Extension: `org::openarchitectureware::util::stdlib::tracing`

2.12.1. Void `createTrace(Object from, Object to, String kind, String backKind)`

Creates a trace between two elements.

Parameters:

- `from` - Source element.
- `to` - Target element.
- `kind` - Name for the trace from source to target.
- `backkind` - Name for the trace from target back to source.

Returns: Nothing.

2.12.2. Void `createTrace(Object from, Object to, String kind)`

Creates a trace between two elements.

Parameters:

- `from` - Source element.
- `to` - Target element.
- `kind` - Name for the trace from source to target.

Returns: Nothing.

2.12.3. Void `clearTrace()`

Clears all traces.

Parameters: none

Returns: Nothing.

2.12.4. Object `getSingleTraceTarget(Object from, String kind)`

Finds the target of a trace. This function will report an error if no trace for the source element to the target of the specified kind can be found.

Parameters:

- `from` - Source element.
- `kind` - Trace kind name.

Returns: The target element of that trace.

2.12.5. Boolean `hasTrace(Object from, String kind)`

Proves if a trace of a specific kind exists for some element.

Parameters:

- `from` - Source element.
- `kind` - Trace kind name.

Returns: true, if a trace of that kind exists for the element.

3. Stdlib workflow components

Besides the extensions described in the previous section Xpand's Stdlib provides some workflow components.

3.1. SystemCommand

This component executes a system command.

Table 2. Workflow component `org.eclipse.xtend.util.stdlib.SystemCommand`

Property	Type	Mandatory	Description
<code>command</code>	String	yes	The command to execute.
<code>directory</code>	String	no	Execution directory.
<code>arg</code>	String	no	(multiple) command arguments
<code>env</code>	String	no	(multiple) additional environment entries. Format: [key],[value]

Example:

```
<component class="org.eclipse.xtend.util.stdlib.SystemCommand">
  <directory value="src-gen"/>
  <command value="sh"/>
  <arg value="processdot.sh"/>
</component>
```

Console output:

```
1639 INFO - Running command '[sh, processdot.sh]' in directory [absolutePath] ...
1667 INFO - processing shape_box.dot ...
2597 INFO - processing shape_polygon.dot ...
...
3564 INFO - Execution of command was successful.
```

Windows tip:

When executing a command on windows this is typically done with the `cmd` as command value. It is important that the command terminates, therefore the argument `/c` must be appended as arg value.²

3.2. SlotCopier

This component copies an element from one slot to another. The slot content is not cloned.

Table 3. Workflow component `org.eclipse.xtend.util.stdlib.SlotCopier`

Property	Type	Mandatory	Description
<code>fromSlot</code>	String	yes	Source slot name.

²See <http://www.ss64.com/nt/cmd.html>, <http://www.java-forum.org/de/viewtopic.php?p=469059>

Property	Type	Mandatory	Description
toSlot	String	yes	Destination slot name.
removeTopLevelList	boolean	no	If true the source slot must contain a list and the top level list is removed (i.e. the first element from the list is copied to the destination slot), otherwise it is not removed.

Example:

```
<component class="org.eclipse.xtend.util.stdlib.SlotCopier">
  <fromSlot value="model"/>
  <toSlot value="target"/>
</component>
```

Console output:

```
INFO SlotCopier copying org.eclipse.emf.ecore.impl.DynamicEObjectImpl@1fdbef
(eClass: org.eclipse.emf.ecore.impl.EClassImpl@fc5b01
(name: Model) (instanceClassName: null) (abstract: false, interface: false))
[org.eclipse.emf.ecore.impl.DynamicEObjectImpl]
```

3.3. SlotListAdder

This component copies an element from one slot to a list contained in another slot.

Table 4. Workflow component org.eclipse.xtend.util.stdlib.SlotListAdder

Property	Type	Mandatory	Description
modelSlot	String	yes	Source slot name.
listSlot	String	yes	Target slot name. This slot contains a list of elements.
uniqueNames	boolean	no	If true, names have to be unique, otherwise not. Requires that modelSlot contains an EObject.

Example:

This example adds the content of slot 'model' to the slot 'targetList'. The slot 'targetList' does not contain anything at the time of execution.

```
<component class="org.eclipse.xtend.util.stdlib.SlotListAdder">
  <modelSlot value="model"/>
  <listSlot value="targetList"/>
</component>
```

Console output:

```
INFO CompositeComponent SlotListAdder: adding contents of slot 'model' to the list of
stuff in 'targetList'
...
INFO WorkflowRunner workflow completed in 1503ms!
WARN WorkflowRunner 'targetList' is empty, creating a new list.
[org.eclipse.xtend.util.stdlib.SlotListAdder@7536e7]
```

Note that the warn messages will appear after the workflow finished, since they are reported as a workflow warn issue.

3.4. SlotPrinter

This component prints a workflow context slot content to the log. This can be useful for debugging purposes.

Table 5. Workflow component org.eclipse.xtend.util.stdlib.SlotPrinter

Property	Type	Mandatory	Description
slotName	String	yes	The name of a slot whose content should be dumped.

Property	Type	Mandatory	Description
message	String	no	An optional message that will be prefixed to the log output.
level	String	no	The log level for the message. Valid values are TRACE, DEBUG, INFO, WARN.

Example:

```
<component class="org.eclipse.xtend.util.stdlib.SlotPrinter">
  <slotName value="model"/>
  <message value="DUMP"/>
  <level value="INFO"/>
</component>
```

Console output:

```
INFO SlotPrinter DUMP: (slot: model)org.eclipse.emf.ecore.impl.DynamicEObjectImpl@d22ddb
(eClass: org.eclipse.emf.ecore.impl.EClassImpl@fe0ce9 (name: Model) (instanceClassName: null)
(abstract: false, interface: false))
```

Chapter 5. Xpand Eclipse Integration

1. Introduction

This document describes the various functionalities that the Xpand 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 Xpand framework itself.

2. Installation

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

3. Overview

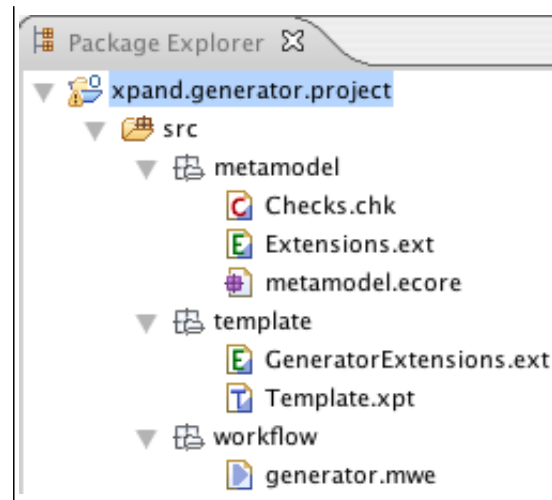
The Xpand UI plugins provide editors for the different languages and a launch shortcut to start workflow files. Let us first have a look at the different Xpand specific files.

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 (.mwe extension)
- Xpand2 templates (.xpt extension)
- Extension files (.ext extension)
- Check constraints (.chk extension)

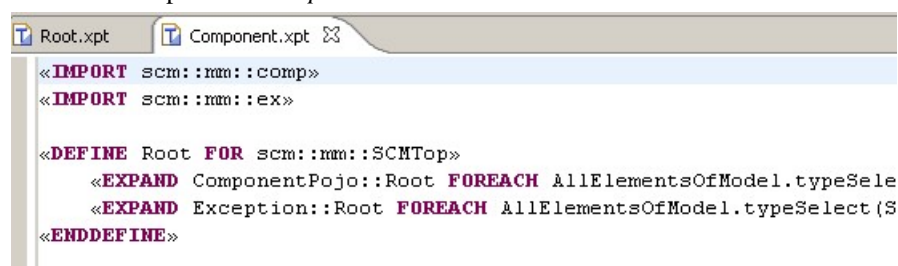


5. Editors

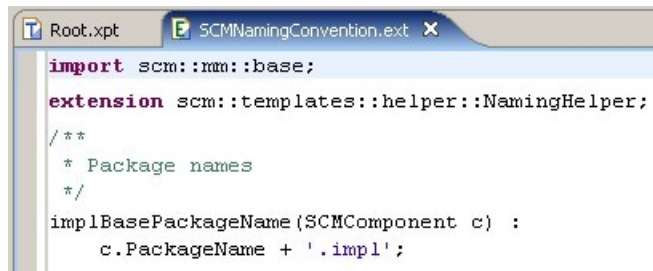
When you double-click on one of the above mentioned file types, special editors will open that provide appropriate syntax coloring.

5.1. Syntax coloring

Here are examples for the Xpand editor:



for the Extensions editor:



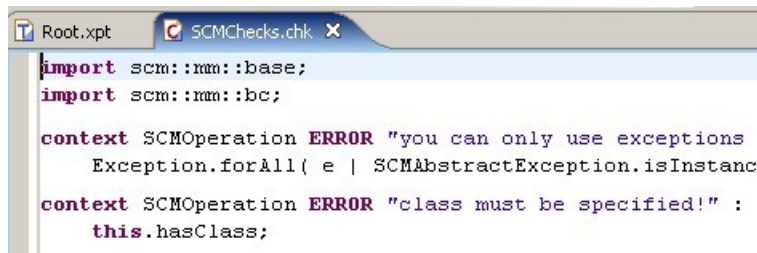
```

import scm::mm::base;
extension scm::templates::helper::NamingHelper;

/**
 * Package names
 */
implBasePackageName(SCMComponent c) :
    c.PackageName + '.impl';

```

and for *Check* editor:



```

import scm::mm::base;
import scm::mm::bc;

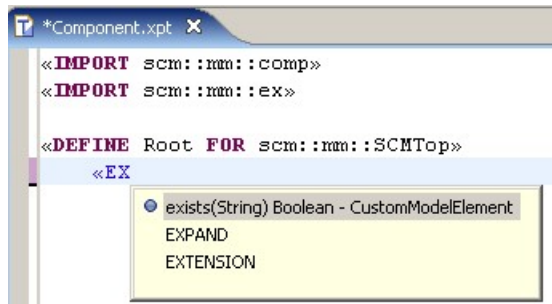
context SCMOperation ERROR "you can only use exceptions
    Exception.forAll( e | SCMAbstractException.isInstant

context SCMOperation ERROR "class must be specified!" :
    this.hasClass;

```

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.



```

<<IMPORT scm::mm::comp>>
<<IMPORT scm::mm::ex>>

<<DEFINE Root FOR scm::mm::SCMTop>>
<<EX

```

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 "«"

Ctrl + > creates "»"

6. Preference pages

6.1. Metamodel contributors

Xpand 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.

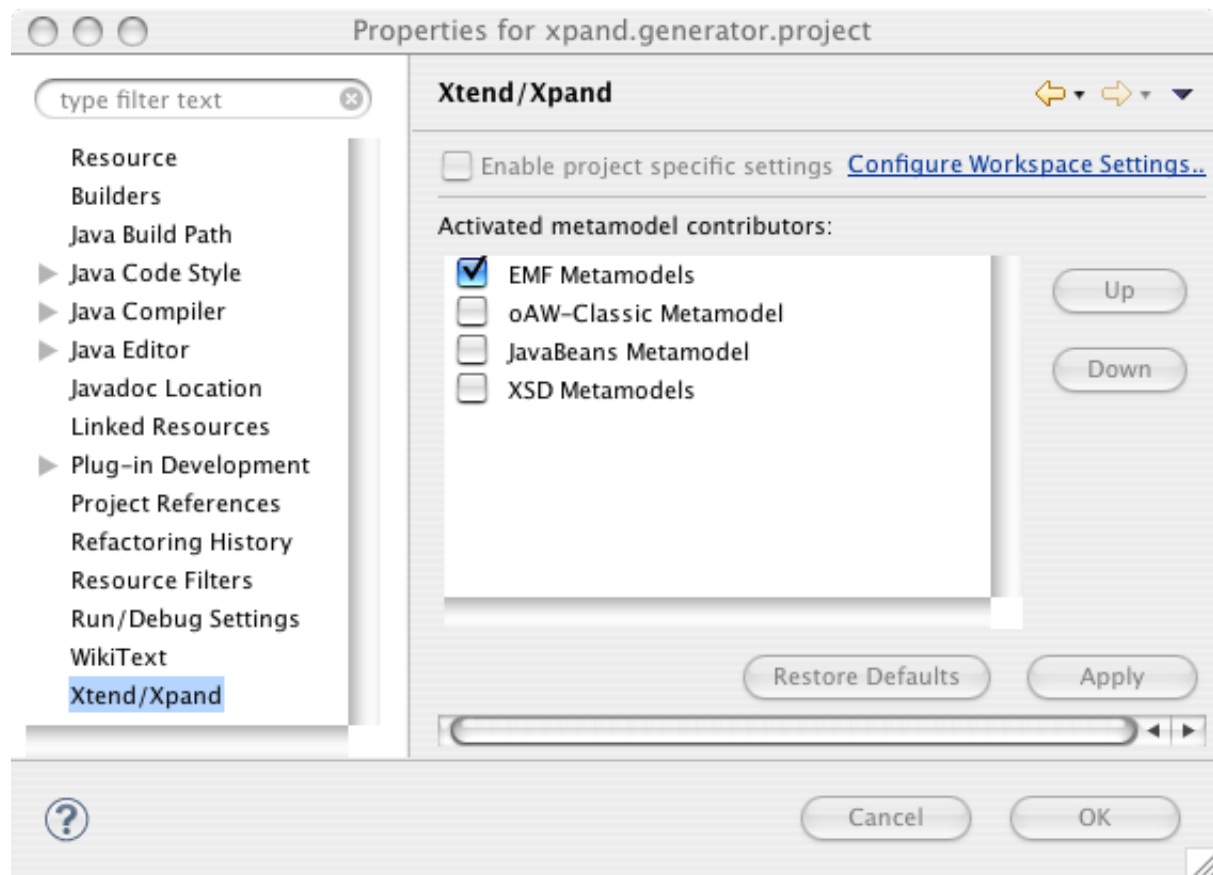
6.2. Global preferences

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.

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.

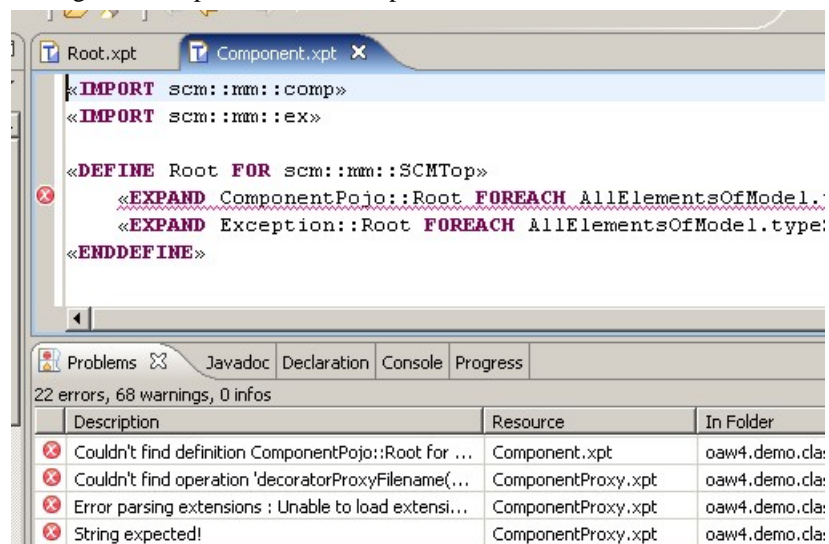


7. Xpand Nature and Xpand Builder

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

7.1. Problem markers

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

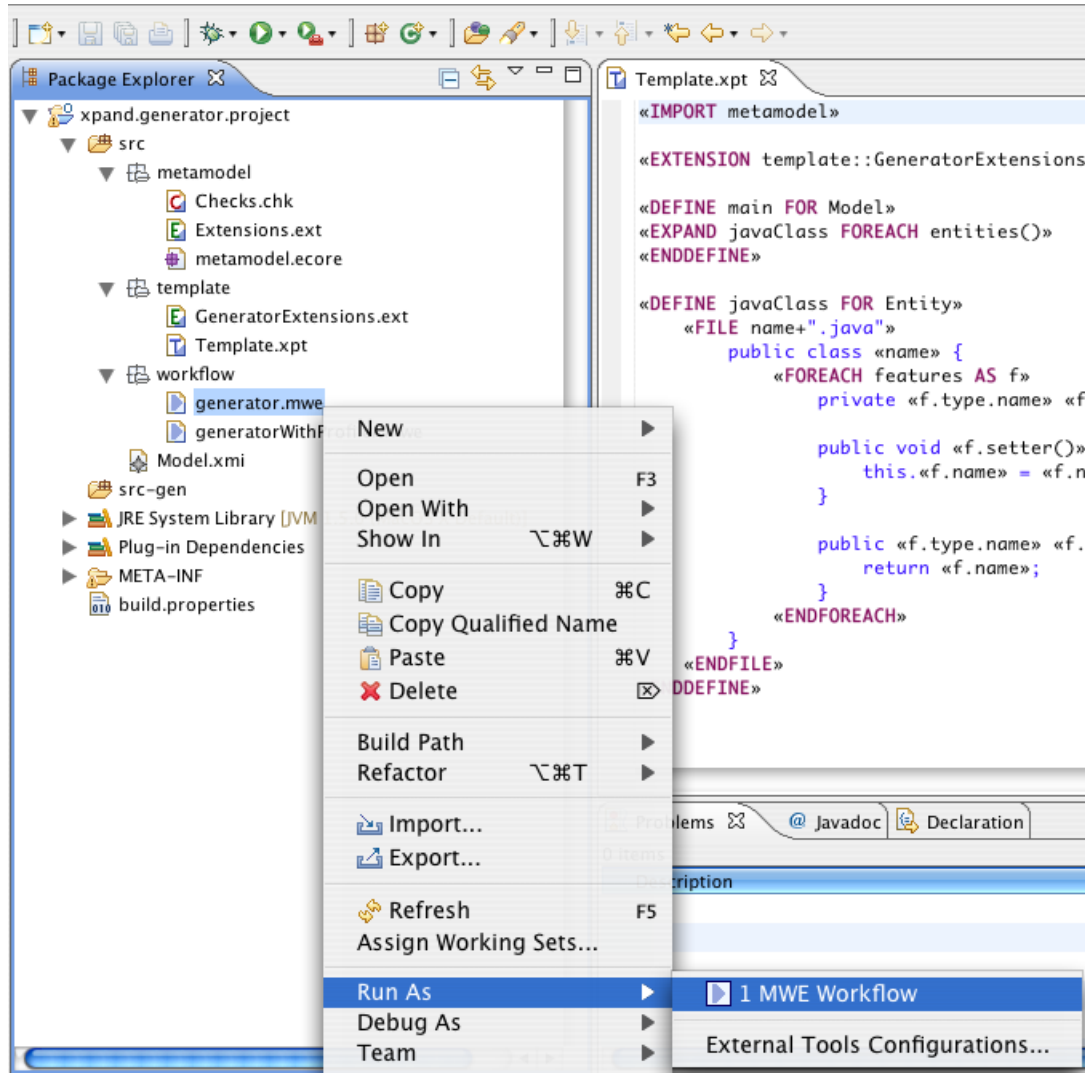


In the current implementation analyzes take place when *Eclipse* runs an incremental or full build. That means, problem markers are actualized when a file is saved, for instance. If you are in doubt about the actuality of problem markers, you should clean your project and let it rebuild again.

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

8. Running a workflow

You can start a workflow by right clicking on a workflow file (*.mwe) and selecting Run As → MWE workflow.



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

Part III. Tutorials

Chapter 6. UML2 Adapter

1. Introduction

The UML2 adapter for Xpand is available since version of *openArchitectureWare* 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 Xpand 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 Xpand types and tagged values to Xpand properties. You do not have to implement additional Java classes anymore. Just export your models and applied profiles. That's all!

2. Installation

First you need to install the UML2 feature from eclipse.org:

<http://download.eclipse.org/modeling/mdt/updates/releases/>

The Xpand uml2adapter is available from the Xpand update site:

<http://download.eclipse.org/modeling/m2t/updates/releases/>

(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.

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 'Xtend/Xpand' tab, activate the checkboxes *Enable project specific settings* and add the *UML2 profiles* metamodel contributor. Use the *Up*-button to push the UML2 profiles contributor to the first position.

3.1. Profiles in Eclipse

If you want Eclipse to register 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)

4. Runtime Configuration

At runtime, you just need the `org.eclipse.xtend.typesystem.uml2-1.0.0.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.

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="org.eclipse.xtend.typesystem.uml2.Setup" standardUML2Setup="true" />
  <component class="org.eclipse.xtend.typesystem.emf.XmiReader">
    ...
  </component>
  <component class="org.eclipse.xpand2.Generator">
    <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel" />
    <metaModel class="org.eclipse.xtend.typesystem.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 Xpand components, you should use the id / idRef mechanism:

```
<workflow>
  <bean class="org.eclipse.xtend.typesystem.uml2.Setup" standardUML2Setup="true" />
  <component class="org.eclipse.xtend.typesystem.emf.XmiReader">
    ...
  </component>
  <component class="org.eclipse.xpand2.Generator">
    <metaModel id="uml"
      class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel" />
    <metaModel id="profile"
      class="org.eclipse.xtend.typesystem.uml2.profile.ProfileMetaModel">
      <profile value="myProfile.profile.uml2" />
    </metaModel>
    ...
  </component>
  <component class="oaw.xpand2.Generator">
    <metaModel idRef="uml"/>
    <metaModel idRef="profile"/>
    ...
  </component>
```

Chapter 7. UML2 Example

1. Setting up Eclipse

Before you can use Xpand with Eclipse UML2, you first have to install the UML2 plugins into your Eclipse installation. (You need at least the 'UML2 End-User' Features. Downloadable from the Eclipse-Update site. Work with: Helios update site, and then browse to *Modeling* and select at least the *UML2 Extender SDK*)

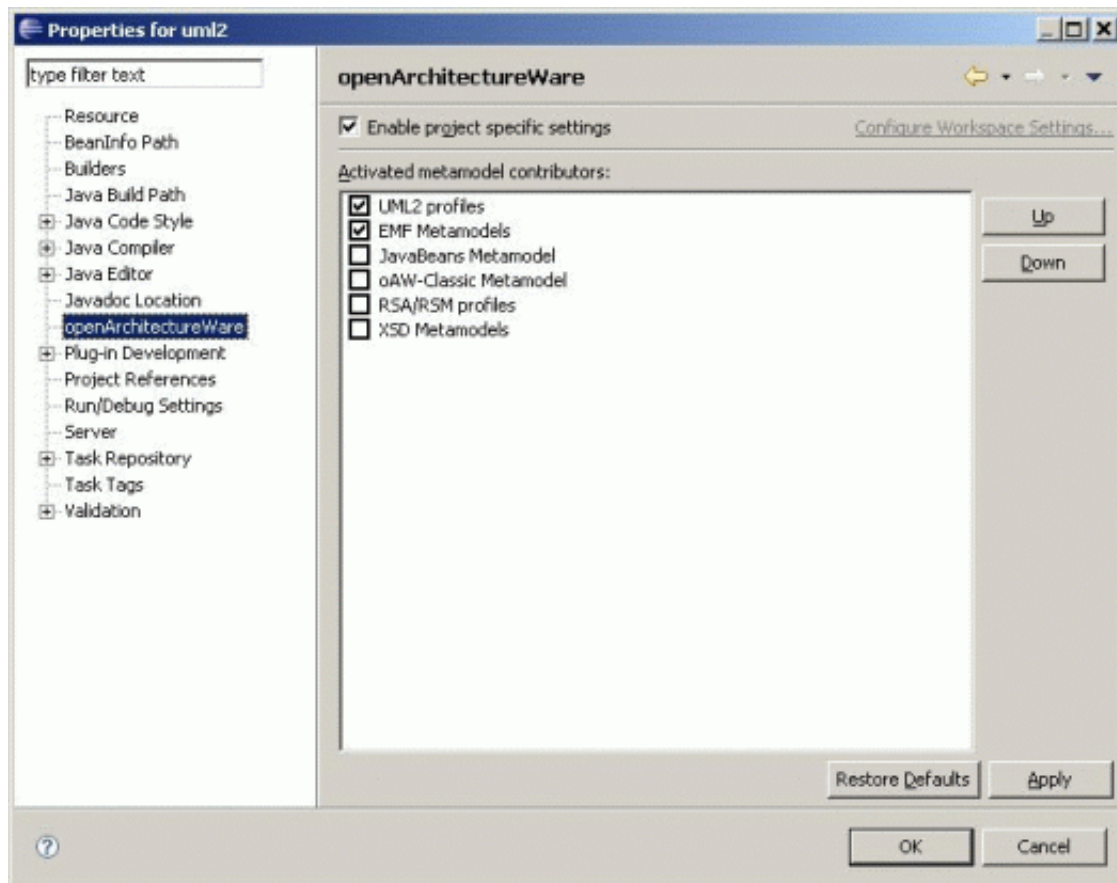
2. Setting up the project

Create a new Xpand plugin project named *xpand.uml2.generator*. You have to add the following dependencies to the manifest file:

- `org.eclipse.xtend.typesystem.uml2`

To tell the Xpand Eclipse plugins that this project is a UML2 specific one, you need to specify that in the Xpand preferences. Open the project properties, select the Xtend/Xpand tab and select the UML2 profiles metamodel.

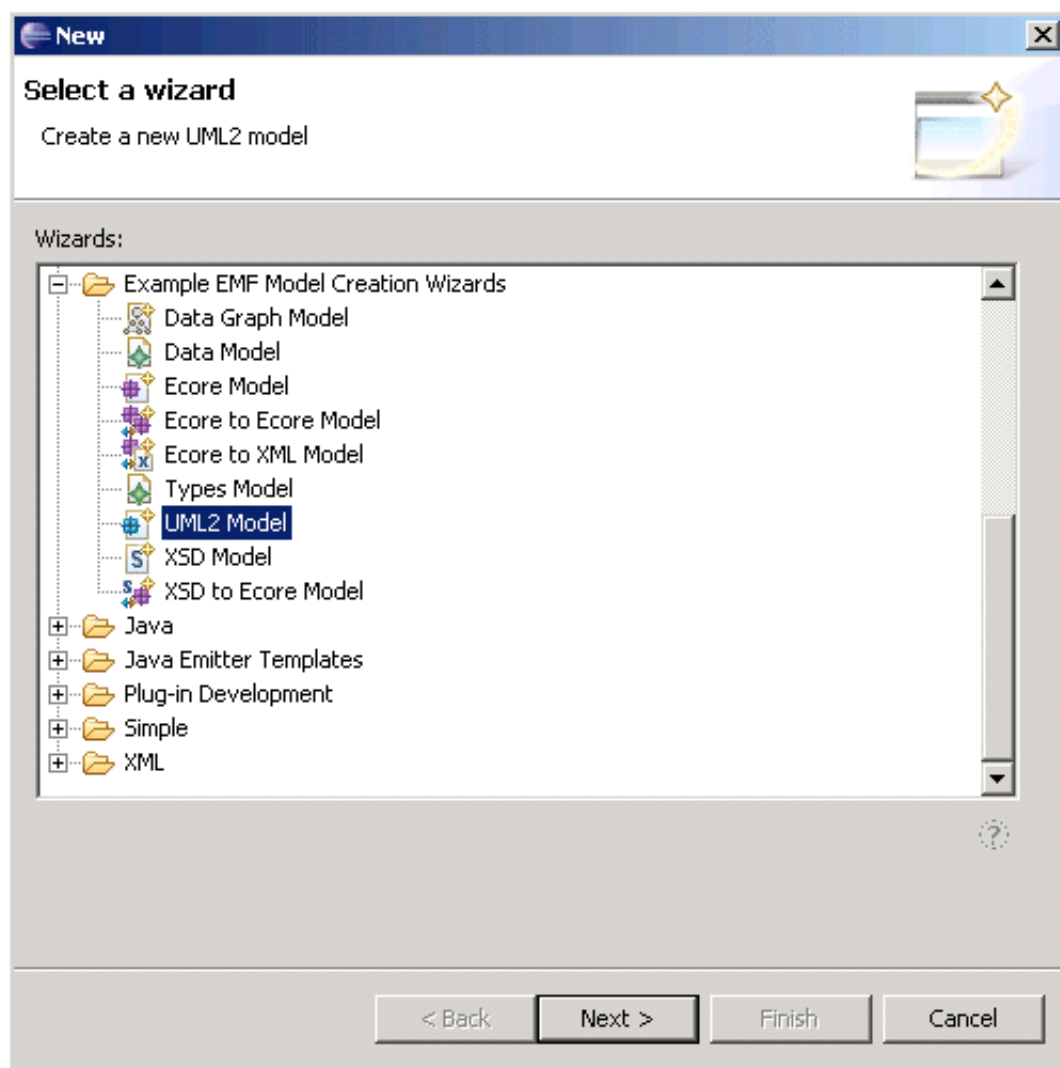
Figure 1. Configure UML2 profiles metamodel



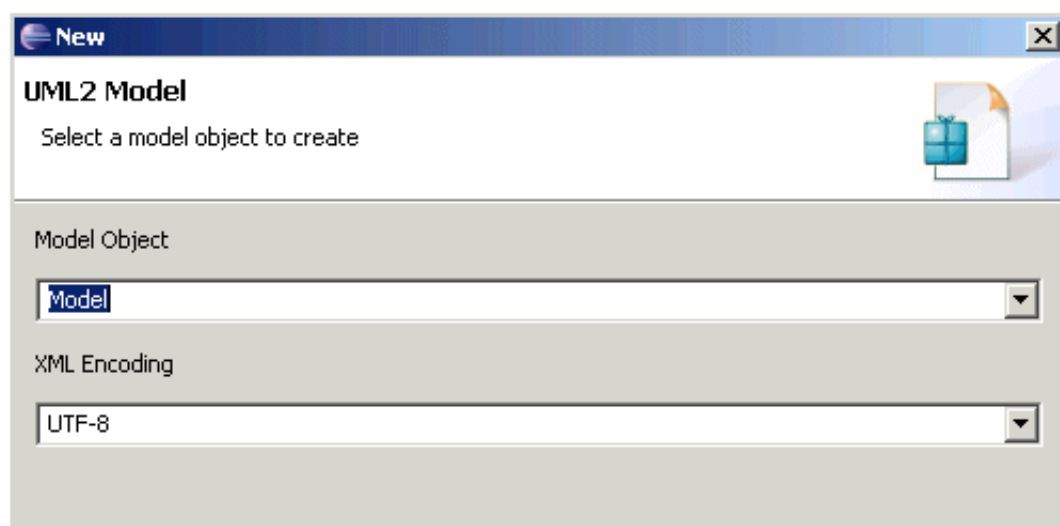
Note that if you want to transform an UML2 model into a normal EMF model, you need to add the *UML2 profiles* and the *EMF Metamodels* metamodel contributor. The order of profiles is important! The UML2 profiles entry must be first in the list.

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 2. Creating a new UML2 model

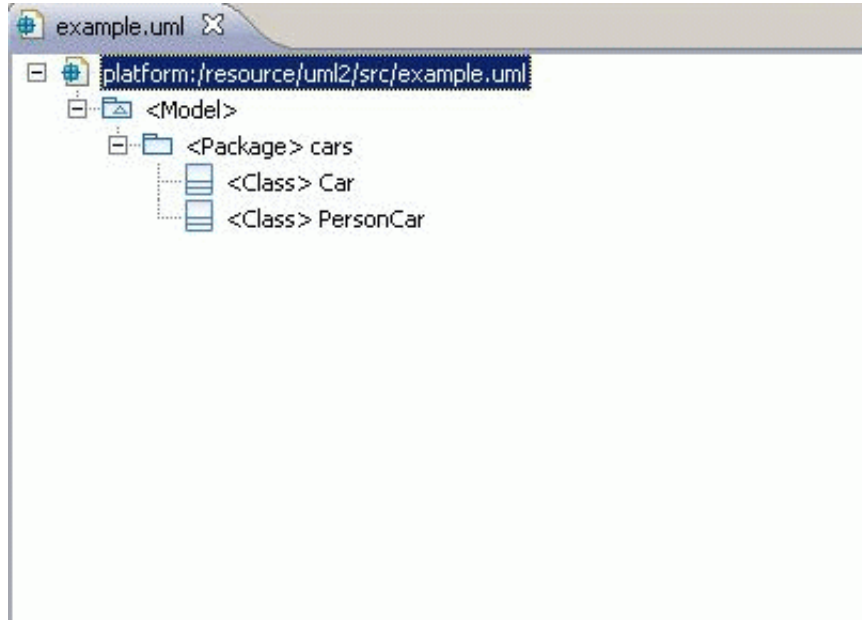
You then have to select the model object. Make sure its a *Model*, not a *Profile*.

Figure 3. Selecting the Model object

3.1. Modelling the content

You should then build a model that looks somewhat like this:

Figure 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!

4. Code generation

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 type `uml::Package` in the model and expand a template for the packages defined in it.

```

«DEFINE Root FOR uml::Model»
  «EXPAND PackageRoot FOREACH allOwnedElements().typeSelect(uml::Package)»
«ENDDEFINE»

```

In the package template, we again iterate over all owned elements and call a template that handles classes. Although we only have classes in that package we could not rely on this in general. The package may contain any other packageable element, so we need to filter classes using `typeSelect()`.

```

«DEFINE PackageRoot FOR uml::Package»
  «EXPAND ClassRoot FOREACH ownedType.typeSelect(uml::Class)»
«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»

```

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 Tutorial chapter.

```
<?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="org.eclipse.emf.mwe.utils.StandaloneSetup" >
  <platformUri value="." />
</bean>

<!-- load model and store it in slot 'model' -->
<component class="org.eclipse.emf.mwe.utils.Reader">
  <uri value="platform:/resource/xpand.uml2.generator/example.uml" />
  <modelSlot 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="org.eclipse.emf.mwe.utils.DirectoryCleaner"
  directory="src-gen"/>
```

and in the generator we also configure the UML2 metamodel.

```
<component id="generator" class="oaw.xpand2.Generator" skipOnErrors="true">
  <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel" />
  <expand value="templates::Root::Root FOR model" />
  <fileEncoding value="ISO-8859-1" />
  <outlet path="src-gen">
    <postprocessor class="org.eclipse.xpand2.output.JavaBeautifier" />
  </outlet>
</component>
</workflow>
```

If you run the workflow (by right clicking on the .mwe file and select Run As → MWE workflow) the two Java classes should be generated.

5. Profile Support

Xpand is shipped with a special *UML2 profiles* 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.

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.

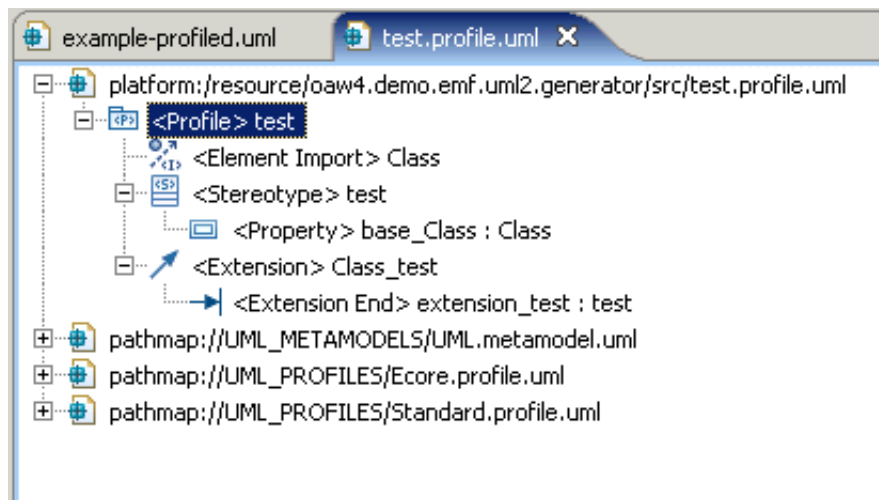
In this section, we explain the "manual way", which is good for explaining what happens, but completely useless for practical use. You do not want to build models of realistic sizes using the mechanisms explained below.

You start by 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.

The created *Profile* we call `test`. 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`. Then, add a stereotype to your profile (right mouse click on the profile -> *New Child* -> *Owned Stereotype* -> *Stereotype* Now you can define your stereotype: 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.

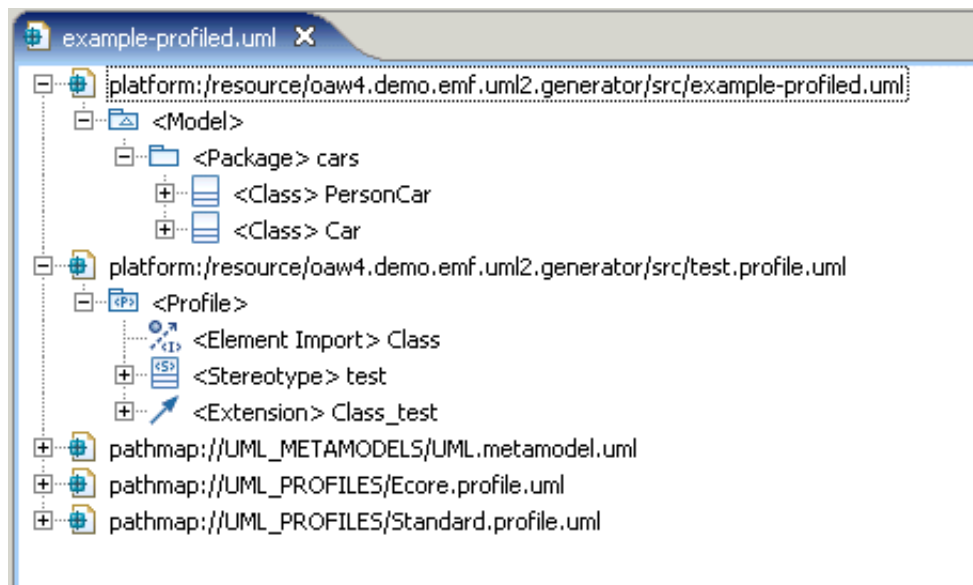
Figure 5. Modelling a Profile



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:

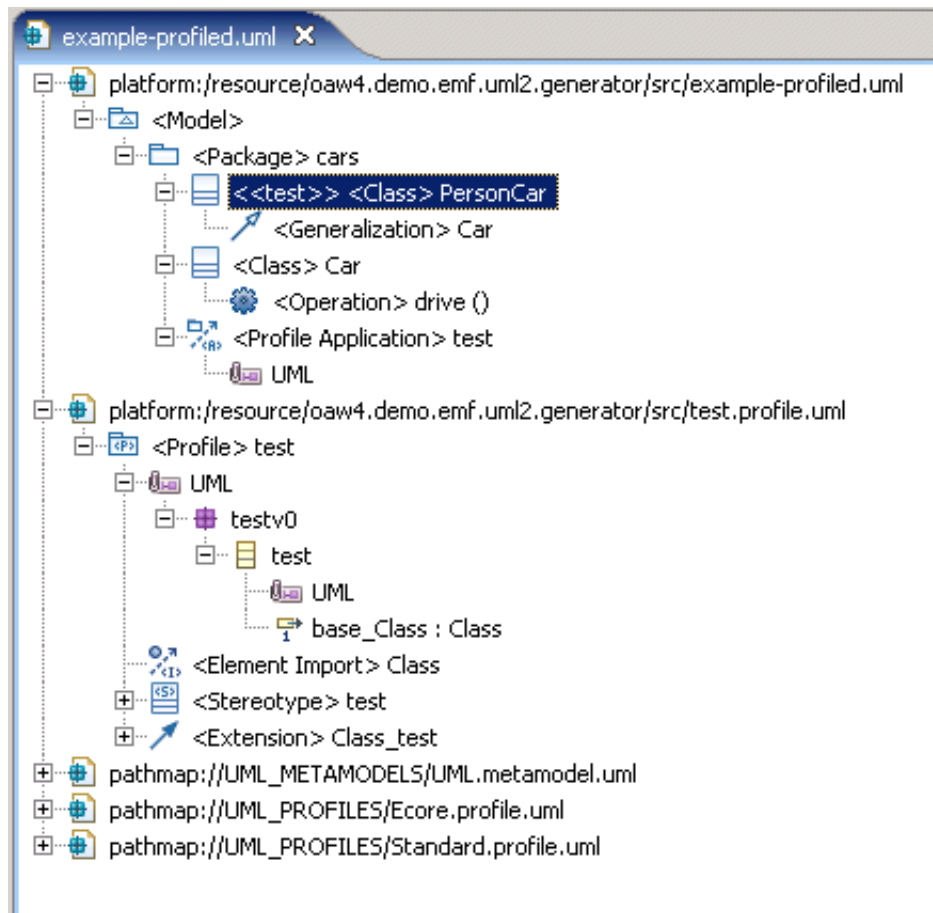
Figure 6. Loading the Profile



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 7. Defining the Profile

5.3. Generating Code

Note that all the stuff above was not in any way related to Xpand, it was just the "bare bones" means of creating and applying a profile to a UML2 model. Having an UML2 tool capable of storing models as EMF UML2 XMI would make the creation of the model far more easier. Since we cannot assume which UML2 tool you are using this tutorial shows you this way, which would always work without further tooling installed.

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 allOwnedElements().typeSelect(uml::Package)»
«ENDDEFINE»

«DEFINE PackageRoot FOR uml::Package»
  «EXPAND ClassRoot FOREACH ownedType.typeSelect(uml::Class)»
«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 model component with the new model `example-profiled.uml`:

```
<!-- load model and store it in slot 'model' -->
<component class="org.eclipse.emf.mwe.utils.Reader">
  <uri value="platform:/resource/xpand2.uml2.generator/example-profiled.uml" />
  <modelSlot value="model" />
</component>
```

And here is the modified generator component:

```
<component id="generator" class="oaw.xpand2.Generator" skipOnErrors="true">
  <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>

<component id="generator" class="oaw.xpand2.Generator" skipOnErrors="true">
  <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel"/>
  <metaModel id="profile"
    class="org.eclipse.xtend.typesystem.uml2.profile.ProfileMetaModel">
    <profile value="test.profile.uml"/>
  </metaModel>
  <expand value="templates::Root::Root FOR model"/>
  <fileEncoding value="ISO-8859-1"/>
  <outlet path="src-gen">
    <postprocessor class="org.eclipse.xpand2.output.JavaBeautifier"/>
  </outlet>
</component>
</workflow>
```

The only thing, we have to do is to add a new metamodel that references the profile we just created.

Chapter 8. XSD Tutorial

This tutorial shows how XML and XML Schemas Definitions (XSD) can be used to generate software. It illustrates how XML files are treated as models, XSDs as meta models and how this integrates with oAW. This tutorial is an introduction, for in-depth details see Chapter 9, *XSD Adapter*.

1. Setup

XSD support for oAW comes with oAW 4.3.1 or later. Make sure the following plugins are installed as well:

- XSD - XML Schema Definition Runtime (<http://www.eclipse.org/xsd/>, available via Ganymede Update Site)
- Web Tools Platform (WTP) (WTP is not required to use oAW XSD support, but helpful, as it provides a nice XML Schema editor and a schema-aware XML editor. (<http://www.eclipse.org/webtools/> , available via Ganymede Update Site)

2. Overview

This tutorial explains how you can do code generation with Xtend and Xpand, using XML Schema Definitions as meta models and XML files as models. To keep things easy, the introduced example is a minimalistic one. A text file is generated from contents specified in XML. The general concept of models, meta models and why and when code generation is useful, is not explained. At the end, a deeper view under the hood is taken to understand how XML Schemas are transformed to EMF Ecore models, and which flexibilities/restrictions this approach provides.

All source files listed within this tutorial are also available as an example project which can be imported into the Eclipse workspace by running *"File" / "New" / "Example..." / "Xpand/Xtend Examples using an XSD Meta Model" / "M2T custom XML to Text via Xpand (minimal Example)"*. This will create the project `org.eclipse.xpand.examples.xsd.m2t.minimal` in your workspace. This minimal example is based on *"M2T custom XML to Java via Xpand"* (`org.eclipse.xpand.examples.xsd.m2t.xml2javawizard`) which is more comprehensive and recommended for further reading.

To generate code from XML files with oAW, at least files of the following four types are needed:

- Meta Model (`metamodel.xsd`)
- Model (`model.xml`)
- oAW Xpand Template (`template.xpt`)
- oAW Workflow (`workflow.oaw`)

Figure 1. Minimalistic oAW XSD Project

3. Step 1: Create a Project

To create a Project, create an ordinary Xtend/Xpand-Project. This is done in Eclipse by changing to the Xtend/Xpand perspective and clicking on *"File" / "New" / "Xtend/Xpand Project"*. After entering a name for the project it is created.

After the project is created, support for XSD meta models needs to be activated. Click with your right mouse button on the project and open the properties window. Then go to the "Xpand/Xtend" page, *"enable project specific settings"* and activate the *"XSD Metamodels"* checkbox. There is no need to leave support for any other meta models activated, except you are sure that you want to use one of them, too. Figure 2, "Activate XSD Meta Model Support for Project" shows how the configuration is supposed to look like.

Figure 2. Activate XSD Meta Model Support for Project

Then, `org.eclipse.xtend.typesystem.xsd` needs to be added to the project's dependencies. To do so open the file `META-INF/MANIFEST.MF` from your project and navigate to the "Dependencies"-tab.

`org.eclipse.xtend.typesystem.xsd` needs to be added to the list of "Required Plug-ins", as it is shown in Figure 3, "Required Dependencies for Project".

Figure 3. Required Dependencies for Project

4. Step 2: Define a Meta Model using XML Schema

In case you are not going to use an existing XML Schema Definition, you can create a new one like described below. These steps make use of the Eclipse Web Tools Platform (WTP) to have fancy editors.

In Eclipse, click on "File", "New", "Other..." and choose "XML Schema" from category "XML". Select the project's "src" folder and specify a filename. Clicking on "finish" creates an empty XSD file. It is important that the XSD file is located somewhere within the project's classpath.

This XML Schema consists of two complex data types, which contain some elements and attributes. "complex" in the XSD terminology means that as opposed to simple data types that they can actually have sub-elements and/or attributes. This example is too minimalistic to do anything useful.

The complex Type Wizard contains the elements `startpage`, `name`, `welcometext`, and `choicepage`. Except for `choicepage` all elements have to contain strings, whereas the string of `startpage` must be a valid id of any `ChoicePage`. The complex type `ChoicePage` just contains an `id` and a `name`. For oAW it does not make any difference if something is modeled as an XML-attribute or XML-element. Just the datafield's type defines how oAW treats the value.

To get an overview how schemas can be used by the oAW XSD Adapter, see Section 5, "How to declare XML Schemas"

Internally, the oAW XSD Adapter transforms the XSD model to an Ecore model which oAW can use like any other Ecore model. For more information about that, see Section 4, "Behind the scenes: Transforming XSD to Ecore"

Figure 4. WTP Schema Editor

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/wizard"
  xmlns:tns="http://www.example.org/wizard"
  elementFormDefault="qualified">

  <complexType name="Wizard">
    <sequence>
      <element name="startpage" type="IDREF" />
      <element name="name" type="string" />
      <element name="welcometext" type="string" />
      <element name="choicepage" type="tns:ChoicePage" />
    </sequence>
  </complexType>

  <complexType name="ChoicePage">
    <sequence>
      <element name="title" type="string" />
    </sequence>
    <attribute name="id" type="ID" />
  </complexType>

  <element name="wizard" type="tns:Wizard" />
</schema>
```

5. Step 3: Create a Model using XML

As the title says, data in XML-Format will be the model. And as a model has to be valid according to a meta model, the XML files must be valid according to the XSD.

In case you are not going to use an existing XML file, you can create a new one like described below. These steps require the Eclipse Web Tools Platform (WTP) to be installed.

In Eclipse, click on *"File"*, *"New"*, *"Other..."* and choose *"XML"* from category *"XML"*. After specifying a filename within folder *"src"* choose *"create XML file from an XML Schema"* and select you XML Schema Definition file. Telling Eclipse which schema to use has three advantages: Eclipse validates XML files, there is meta model aware code completion while editing and Eclipse creates a `xsi:schemaLocation`-attribute which tells anyone who reads the XML file where the schema file is located. This tutorial does not use the `xsi:schemaLocation`-attribute and introduces the schema file in the oAW workflow instead. For all possible ways see Section 5, "How to declare XML Schemas". It is important that the XML file is located somewhere within the project's classpath.

```
<?xml version="1.0" encoding="UTF-8"?>
<wizard xmlns="http://www.example.org/wizard">
  <startpage>start</startpage>
  <name>My Example Setup</name>
  <welcometext>Welcome to this little demo application.</welcometext>
  <choicepage id="start">
    <title>Wizard Page One</title>
  </choicepage>
</wizard>
```

6. Step 4: Create a Template using Xpand

Create an ordinary oAW Xpand file: Being in the Xpand/Xtend perspective, go to *"File"*, *"New"*, *"xPand template"*. The Xpand language itself is explained by several other oAW documents. Having XSD meta model support activated like described in Section 3, "Step 1: Create a Project", oAW scans and watches all it's projects for suitable meta models. Based on what is found, the Xpand editor provides meta model aware code completion.

This example imports *"metamodel"* at the beginning, which refers to a file called `metamodel.xsd` that you have created within the project's classpath in Section 4, "Step 2: Define a Meta Model using XML Schema". The `define-block` can be understood as a function named *"Root"* which takes one object of type `metamodel::Wizard` as a parameter. This is the meta model's type for the XML's root object. The `file-block` creates a file named `wizard.txt` and writes the text that is surrounded by the `file-block` into the file. `name`, `welcometext` and `choicepage.title` are elements or attributes defined in the XSD meta model. Their values are stored within the XML file and this templates inserts them into the generated (`wizard.txt`) file.

```
«IMPORT metamodel»

«DEFINE Root FOR metamodel::Wizard»
«FILE "wizard.txt"»
Name: «name»
Welcometext: «welcometext»
First Page Title: «choicepage.title»
«ENDFILE»
«ENDDEFINE»
```

7. Step 5: Create a Workflow

The workflow ties together model, meta model and templates and defines the process of how to generate code.

To create a new workflow file, switch to the Xpand/Xtend perspective, click on *"File"*, *"New"* and *"Workflow file"*. After specifying a folder and a filename an empty workflow is created.

The minimalistic approach consists of two steps:

1. Read the Model: This is done by `org.eclipse.xtend.typesystem.xsd.XMLReader`. It needs exactly one `uri` element which defines the XML file. A further nested element of type `org.eclipse.xtend.typesystem.xsd.XSDMetaModel` tells the `XMLReader` which `metamodel` to use. `XSDMetaModel` can contain multiple `schemaFile` elements. How the schemas are used for the XML file is determined based on the declared namespaces. `modelSlot` defines a location where the model is stored internally, this is like a variable name which becomes important if you want to handle multiple models within the same workflow.
2. Generate Code: This part just does the regular code generation using Xpand and is not specific to the oAW XSD Adapter at all. The generator `org.eclipse.xpand2.Generator` needs to know which meta model to use. This example references the previously declared one. The `expand` element tells the generator to call the definition named `Root` within file `template.xpt` using the contents of slot `model` as parameter. Element `outlet` defines where to store the generated files.

```
<workflow>
  <component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
    <modelSlot value="model" />
    <uri value="model.xml" />
    <metaModel id="mm"
      class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
      <schemaFile value="metamodel.xsd" />
    </metaModel>
  </component>
  <component class="org.eclipse.xpand2.Generator">
    <metaModel idRef="mm" />
    <expand value="template::Root FOR model" />
    <outlet path="src-gen" />
  </component>
</workflow>
```

8. Step 6: Execute Workflow aka Generate Code

Before you actually execute the workflow, or in case of errors, you can use Figure 5, “Files of this Tutorial” to double check your files.

Figure 5. Files of this Tutorial

To execute the workflow, click with your right mouse button on the workflow file and choose *“Run As”, “oAW Workflow”*, as it is shown in Section 8, “Step 6: Execute Workflow aka Generate Code”.

Figure 6. Execute Workflow

When executing the workflow, this output is supposed to appear in Eclipse's Console View. If that View does not pop up automatically, you can reach it via *“Window”, “Show View”, “Console”*.

```
May 25, 2009 3:09:35 PM org.eclipse.emf.mwe.core.WorkflowRunner prepare
INFO: running workflow: /Users/meyscholdt/Eclipse/workspace-3.5-M7/org.eclipse.xpand.examples.xsd.m2t.minima
May 25, 2009 3:09:35 PM org.eclipse.emf.mwe.core.WorkflowRunner prepare
INFO:
May 25, 2009 3:09:36 PM org.eclipse.xtend.typesystem.xsd.XSDMetaModel addSchemaFile
INFO: Loading XSDSchema from 'xsd/m2t/minimal/metamodel.xsd'
May 25, 2009 3:09:37 PM org.eclipse.xtend.typesystem.xsd.builder.OawXSDEcoreBuilder initEPackage
INFO: Creating EPackage 'metamodel' from XSDSchema 'file:///bin/xsd/m2t/minimal/metamodel.xsd' (http://ww
May 25, 2009 3:09:37 PM org.eclipse.emf.mwe.core.container.CompositeComponent internalInvoke
INFO: XMLReader: Loading XML file xsd/m2t/minimal/model.xml
May 25, 2009 3:09:37 PM org.eclipse.emf.mwe.core.container.CompositeComponent internalInvoke
INFO: Generator: generating 'xsd:m2t::minimal::template::Root FOR model' => src-gen
May 25, 2009 3:09:38 PM org.eclipse.xpand2.Generator invokeInternal2
INFO: Written 1 files to outlet [default](src-gen)
May 25, 2009 3:09:38 PM org.eclipse.emf.mwe.core.WorkflowRunner executeWorkflow
INFO: workflow completed in 657ms!
```

After code generation, there is a file called `wizard.txt` within the `src-gen` folder. Its contents is supposed to look like shown below. You should be able to recognize the structure you've defined within the template file and the contents from your XML model.

```
Name: My Example Setup
Welcometext: Welcome to this little demo application.
First Page Title: Wizard Page One
```

Chapter 9. XSD Adapter

The XSD Adapter allows oAW to read/write XML files as models and to use XML Schemas (XSDs) as meta models. This reference provides in-depth details, for a quick and pragmatic introduction see Chapter 8, *XSD Tutorial*.

1. Prerequisites

Please take a look at Section 1, “Setup”.

2. Overview

The XSD Adapter performs two major tasks:

1. It converts XML Schemas (XSDs) to Ecore models in a transparent manner, so that the Ecore models are hidden from the user. This is done in the workflow as well as in the IDE (to allow XSD-aware code completion for Xtend/Xpand/Check). For details about the mapping see Section 4, “Behind the scenes: Transforming XSD to Ecore”. For details about the workflow integration see Section 3, “Workflow Components”
2. It extends the `EmfMetaModel` with concepts that are needed for XSDs. These are, for example, support for feature maps (needed to handle comments, nested text, CDATA and processing instructions), QNames, EMaps and composed Simpletypes.

3. Workflow Components

The XSD Adapter provides the following workflow components:

3.1. XSDMetaModel

The `XSDMetaModel` loads the specified XSD, transforms them to Ecore models and makes them available for the other oAW components. If XSDs include/import other XSDs or if XML files reference XSDs via `schemaLocation`, these XSDs are also loaded (details: Section 5, “How to declare XML Schemas”). The most common scenario is to declare the `XSDMetaModel` within an `XMLReader`:

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="model.xml" />
  <metaModel id="mm" class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
    <schemaFile value="metamodel.xsd" />
    <registerPackagesGlobally value="true" />
  </metaModel>
</component>
```

Another option is to specify an `XSDMetaModel` independently of other components as a bean:

```
<bean id="mymetamodel" class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
  <schemaFile value="metamodel.xsd" />
</bean>
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="model.xml" />
  <metaModel idRef="mymetamodel" />
</component>
```

Attention: It can lead to errors when XSDs are loaded multiple times, which can only happen when using multiple `XSDMetaModels` within one workflow. The safe way to go is to declare just one `XSDMetaModel` per workflow and reference it from all components that need it.

Properties:

- `schemaFile`: optional, allowed multiple times: Specifies an XSD file which is being loaded. The path can be a complete URI, or relative to the project root or classpath.
- `registerPackagesGlobally`: optional, default "false": If true, generated EPackages are registered to `org.eclipse.emf.ecore.EPackage.Registry.INSTANCE`, EMF's global package registry. Warning: when running workflows from your own java code, make sure to remove the generated packages from the registry before the next run!

3.2. XMLReader

The `XMLReader` reads one XML file which is valid according to the XSDs loaded by the `XSDMetaModel`. The XML file is loaded as a model and stored in the specified slot. Example:

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="model.xml" />
  <metaModel idRef="mymetamodel" />
</component>
```

Properties:

- **slot:** required: The name of the slot which in which the loaded model is stored. Other workflow components can access the model via referring to this slot.
- **uri:** required: The file name of the XML file which should be read. Absolute URIs, and pathnames relative to the project root or to the classpath are valid.
- **metaModel:** optional: Specifies the `XSDMetaModel` (see Section 3.1, “`XSDMetaModel`”) for the `XMLReader`. In case no `XSDMetaModel` is specified, an `XSDMetaModel` with default configuration is instantiated implicitly. It is important to pay attention that all needed XSDs can be found while the loading process: Section 5, “How to declare XML Schemas”.
- **useDocumentRoot:** optional, default "false": Dealing with XML files as models, most people think of the XML's root element as the model's root object. This is the default used by the `XMLReader`. But the XML's root element actually has a parent, the so-called `DocumentRoot`. Additionally the `DocumentRoot` contains comments/processing instructions and CDATA section which appears before or after the XML's root element, and, most notably, the `DocumentRoot` contains information about the used namespaces. If `useDocumentRoot` is set to true, the `XMLReader` stores the `DocumentRoot`-Object instead the XML's root element's object to the specified slot.
- **option:** optional, can be specified multiple times: Option specifies a key-value-pair, which is handed on to the EMF's `XMLResource` in the loading process. Valid options are documented via JavaDoc in interface `org.eclipse.emf.ecore.xml.XMLResource`. Additionally, the `XMLReader` supports these options:

- **DEFAULT_NAMESPACE:** Specifies a default namespace, in case the XML file does not declare one:

```
<option key="DEFAULT_NAMESPACE" val="http://www.dlese.org/Metadata/opml" />
```

- **NAMESPACE_MAP:** Specifies a mapping for namespaces, which is applied when loading XML files.

```
<option key="NAMESPACE_MAP">
  <val class="org.eclipse.xtend.typesystem.xsd.lib.MapBean">
    <mapping from="http://www.eclipse.org/modeling/xpand/example/model/wrong"
      to="http://www.eclipse.org/modeling/xpand/example/model/loadcurve" />
  </val>
</option>
```

3.3. XMLWriter

The `XMLWriter` writes the model stored in a slot to an XML file. If the slot contains a collection of models, each one is written to a separate file. The model(s) must have been instantiated using an XSD-based meta model. Example:

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLWriter">
  <metaModel idRef="svgmm" />
  <modelSlot value="svgmodel" />
  <uri value="src-gen/mycurve.svg" />
</component>
```

Properties:

- **slot:** required: The name of the slot which holds the model or the collection of models which shall be serialized to XML.
- **metaModel:** required: The instance of `XSDMetaModel`, which holds the XSD that the supplied models are based on. Also see Section 3.1, “`XSDMetaModel`”

- `uri`: required if no `uriExpression` is specified: The file name of the XML file which should be written. Absolute URIs are valid. Use relative path names on your own risk.
- `uriExpression`: required if no `uri` is specified: In the scenario where multiple XML files are written, this provides a mechanism to determine the file name for each of them. The oAW-expression specified in `expression` is evaluated for each file and has to return a file name. The model that is going to be written is accessible in the expression via a variable that has the name specified in `varName`. Example:

```
<uriExpression varName="docroot" expression="'src-gen/' + ecore2xsd::getFileName(docroot)" />
```

- `option`: optional, can be specified multiple times: Option specifies a key-value-pair, which is handed on to the EMF's XMLResource in the writing process. Valid options are documented via JavaDoc in interface `org.eclipse.emf.ecore.xml.XMLResource`.

3.4. XMLBeautifier

The `XMLBeautifier` uses EMF to load the XML file, formats the mixed content (elements and text contained by the same element) and writes the file back to disk applying a nice indentation for the elements. The `XMLBeautifier` is not intended to be used in combination with the `XMLWriter`, since the `XMLWriter` cares about indentation by itself. Instead, use it for "manually" constructed XML files using `Xpand`. Since the frameworks for loading/storing XML always load the whole file into a complex data structure in memory, this approach does not scale well for huge XML files. Example:

```
<component class="org.eclipse.xpand2.Generator">
  <metaModel idRef="mm" />
  <expand value="${src-pkg}::${file}::Root FOR '${out}'" />
  <outlet path="${src-gen-dir}" />
  <beautifier class="org.eclipse.xtend.typesystem.xsd.XMLBeautifier">
    <maxLineWidth value="60" />
    <formatComments value="true" />
    <fileExtensions value=".xml, .html" />
  </beautifier>
</component>
```

Properties:

- `maxLineWidth`: optional: Specifies the number of character after which a linewrap should be performed.
- `formatComments`: optional, default true: Specifies if formatting should also be applied to comments.
- `fileExtensions`: optional, default ".xml, .xsl, .xsd, .wsdd, .wsdl": Specifies a filter for which files formatting should be applied. Only files that match one of the specified file extensions are processed.
- `loadOption`: optional, can be specified multiple times: Option specifies a key-value-pair, which is handed on to the EMF's XMLResource in the loading process. Valid options are documented via JavaDoc in interface `org.eclipse.emf.ecore.xml.XMLResource`.
- `saveOption`: optional, can be specified multiple times: Same as `loadOption`, except for the difference that these options are applied while the writing process. Example:

```
<saveOption key="XML_VERSION" val="1.1" />
<saveOption key="ENCODING" val="ASCII" />
```

4. Behind the scenes: Transforming XSD to Ecore

In the code generation process an XML Schema is transformed to an EMF Ecore model, which is then used as a meta model by EMF. XSD complex data types are mapped to EClasses, XSD simple data types are mapped to EMF data types defined in `org.eclipse.emf.ecore.xml.type.XMLTypePackage` and `org.eclipse.xtend.typesystem.xsd.XSDMetaModel` maps them to oAW data types. The document *XML Schema to Ecore Mapping* explains the mapping's details. <http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf>

5. How to declare XML Schemas

There are three different ways to declare your XSDs. It does not matter which way you choose, or how you combine them, as long as the XSD Adapter can find all needed schemas.

1. Within the Workflow: `org.eclipse.xtend.typesystem.xsd.XSDMetaModel` can have any amount of `schemaFile` elements.

```
<component class="org.eclipse.xtend.typesystem.xsd.XMLReader">
  <modelSlot value="model" />
  <uri value="{file}" />
  <metaModel id="mm" class="org.eclipse.xtend.typesystem.xsd.XSDMetaModel">
    <schemaFile value="model/loadcurve.xsd" />
    <schemaFile value="model/device.xsd" />
  </metaModel>
</component>
```

2. Within the XML file: XML files can contain `schemaLocation` attributes which associate the schema's namespace with the schema's filename. If the schema is created using WTP like described in Section 5, "Step 3: Create a Model using XML", the `schemaLocation` attribute is created automatically.

```
<?xml version="1.0" encoding="UTF-8"?>
<device:Device
  xmlns:device="http://www.eclipse.org/modeling/xpand/example/model/device"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eclipse.org/modeling/xpand/example/model/device device.xsd">
  <device:Name>MyLaptop</device:Name>
</device:Device>
```

3. Within an XSD: If one schema imports another, the `import` element can have a `schemaLocation` attribute, too.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  targetNamespace="http://www.eclipse.org/modeling/xpand/example/model/device"
  elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.eclipse.org/modeling/xpand/example/model/device"
  xmlns:lc="http://www.eclipse.org/modeling/xpand/example/model/loadcurve"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore">

  <import
    namespace="http://www.eclipse.org/modeling/xpand/example/model/loadcurve"
    schemaLocation="loadcurve.xsd">
  </import>

  <complexType name="Device">
    <sequence>
      <element name="Name" type="string" />
      <element name="LoadCurve" type="lc:LoadCurve" />
    </sequence>
  </complexType>

  <element name="Device" type="tns:Device"></element>
</schema>
```

Chapter 10. Incremental Generation

As projects become larger, so typically do their models. However, the larger the models are, the longer the code generation process takes. In a mature project, a developer typically changes only a small part of a large model. Performing a full code generation process for the whole model slows down the development cycle considerably due to various factors:

- The whole model must be traversed, and each statement in the Xpand templates must be executed. The larger the model is and the more Xpand templates you have, the higher the negative impact is.
- All generated files are written to disk. The I/O operation itself is one major contributor to the overall elapsed time. What's more, files are typically post-processed by beautifiers, which is another time consuming operation. If you are working with protected regions, the impact is even more dramatic.
- Since every file has a new timestamp after code generation, typically a compiler will pick up these new files and start compilation, which adds more CPU and I/O cycles to the process.

Considering that for a small change in the model only a fraction of the generated files actually do change their contents, performing a full generation is obviously a waste of time.

Beginning with the Helios release train (Eclipse 3.6, Xpand 0.8), Xpand now ships with an incremental generation facility. This works very similar to the incremental Java compiler in Eclipse. It detects which parts of a model have changed since the last generation process. It then determines which files need to be generated due to that change and which are unaffected by it. Only the former are the regenerated, while the latter remain untouched.

The following sections explain how this incremental generation feature works and how you can use it.

1. Technical Background

The key to incremental generation lies in knowing which element in a model was used to generate which file. This information can easily be computed during generation, by tracking which parts of the model are accessed in the context of any given «FILE» statement. A callback for the Xpand generator does this job and builds up a so-called *trace model* on-the-fly.

The second important information is the actual change that has occurred in a model. There are basically two ways to compute this. One is to attach a change listener when editing the model and capture the change as it happens. The other way is to keep a backup copy of the model and compare the old version with the current version to compute the change. See Section 3.2, “Performance Considerations” for pros and cons of each of the two ways. In either case, the result is a so-called *diff model*.

When we know which parts of a model have changed, and we also know which files have been produced based upon these parts of the model, we can then skip the generation of all other files, thus performing incremental generation.

2. Using Incremental Generation

2.1. The Incremental Generation Facade

The easiest way to benefit from incremental generation is to use the *IncrementalGenerationFacade* workflow component:

```
<workflow>
  <component id="incremental"
    class="org.eclipse.xpand2.incremental.IncrementalGenerationFacade">
    <newModelFile value="path/to/your/model.file" />
    <oldModelFile value="path/to/backup/model.file" />
    <traceModelFile value="path/to/store/trace/model.trace" />
    <outlet path="path/to/your/outlet/" overwrite="true"/>
  </component>

  <component id="generator" class="org.eclipse.xpand2.Generator">
    <expand value="your::template::Root FOR model" />
    <outlet path="temp/" overwrite="true"/>
    <metaModel class="org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel" />
    <vetoableCallback idRef="incremental" />
  </component>
</workflow>
```

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

The *IncrementalGenerationFacade* takes four parameters:

- The *newModelFile* is the file path where the model to generate is stored. This file is stored in a model slot named *model*
- The *oldModelFile* is the file path where a copy of the previous state of the model is stored. The model is automatically copied to this location after the generation process and kept between generator invocations.
- The *traceModelFile* is the file path where the *trace model* of the generation process is stored between generator invocations.
- A regular *outlet* list that must match the one given for the regular generator invocation.

The *IncrementalGenerationFacade* component must then be passed as a *vetoableCallback* parameter to the invocation of the *Xpand Generator*.

With the simple workflow given above, you should be able to observe that for any given change in the model, only the files affected by that change are regenerated, while all others remain untouched. Even deleting elements will result in specific (previously generated) files being deleted from the hard disk.

Note that you have to use file paths for all models because they are physically copied on the hard disk. Passing locations that can only be resolved from the classpath is not possible.

2.2. The Incremental Generation Callback

While the *IncrementalGenerationFacade* is easy to use, it is rather restricted in its capabilities and fixed in the operations it performs. Using the *IncrementalGenerationCallback* gives you more control over the steps involved. A typical workflow for incremental generation needs to perform the following tasks:

1. Read the (current) model into a slot.
2. Read the previous state of the model into another slot. This may, of course, not exist, e.g. for the very first invocation. Full generation must be performed in this case.
3. Compute the changes between the two versions of the model (if possible) and put that *diff model* into a slot.
4. Read the *trace model* computed during the previous generator invocation and put it into a slot. As with the old state of the model, this may not exist, which also leads to full generation.
5. Initialize the *IncrementalGenerationCallback* with the *diff model* and the *trace model*.
6. Run the *Xpand Generator* component with the *IncrementalGenerationCallback*.
7. Clean obsolete files, i.e. files that need to be deleted because the corresponding elements in the model have been deleted.
8. Write the new *trace model* computed during code generation to the hard disk so that it is available for the next generation process.
9. Make a backup copy of the model so that it can be compared with the next version upon subsequent generator invocation.

This is a sample workflow that performs all these steps:

```
<workflow>
  <!-- read new model -->
  <component id="modelreader" class="org.eclipse.emf.mwe.utils.Reader"
    uri="model/my.model"
    firstElementOnly="true"
    modelSlot="model"
  />
  <!-- read old model, copied from previous run. may not exist, so ignore missing model -->
  <component id="oldmodelreader" class="org.eclipse.emf.mwe.utils.Reader"
    uri="temp/old.model"
    firstElementOnly="true"
    ignoreMissingModel="true"
    modelSlot="oldmodel"
  />

  <!-- compute diff. -->
```

```
<component id="compare" class="org.eclipse.xpand2.incremental.compare.EmfCompare"
  oldModelSlot="oldmodel"
  newModelSlot="model"
  diffModelSlot="diff"
/>

<!-- read trace model, produced by previous run. may not exist, so ignore missing model -->
<component id="tracemodelreader" class="org.eclipse.emf.mwe.utils.Reader"
  uri="temp/trace.trace"
  firstElementOnly="true"
  ignoreMissingModel="true"
  modelSlot="oldtrace"
/>

<!-- this is the actual incremental generation callback -->
<component id="incremental"
  class="org.eclipse.xpand2.incremental.IncrementalGenerationCallback"
  diffModelSlot="diff"
  oldTraceModelSlot="oldtrace"
  newTraceModelSlot="trace"
/>

<!-- generate code -->
<component id="generator" class="org.eclipse.xpand2.Generator">
  <expand value="resources::templates::Test::Test FOR model" />
  <outlet path="somewhere/" overwrite="true"/>
  <metaModel class="org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel" />
  <vetoableCallback idRef="incremental" />
</component>

<!-- clean obsolete files -->
<component id="cleaner" class="org.eclipse.xpand2.incremental.FileCleaner">
  <oldTraceModelSlot value="oldtrace" />
  <newTraceModelSlot value="trace" />
  <outlet path="somewhere/" overwrite="true"/>
</component>

<!-- write trace model -->
<component id="tracemodelwriter" class="org.eclipse.emf.mwe.utils.Writer"
  modelSlot="trace"
  uri="temp/trace.trace"
/>

<!-- make backup copy of model -->
<component id="copier" class="org.eclipse.emf.mwe.utils.FileCopy"
  sourceFile="model/my.model"
  targetFile="temp/old.model"
/>
</workflow>
```

3. Additional Notes

3.1. Limitations

The incremental generation process can only be used with EMF-based models. That's because all intermediate artifacts (*diff model* and *trace model*) which reference the original models are also stored as EMF models. It is therefore not possible to refer to other model formats. Moreover, you should make sure that your model has stable IDs for the individual model elements so that the model comparison doesn't run into any ambiguities.

Also note that at the moment, Xpand cannot track access to model elements from JAVA extensions. This can lead to cases where a change in a specific model element should trigger a specific (set of) file(s) to be regenerated, but it actually doesn't. That's because Xpand didn't know about the model element being accessed during the original file generation, so it has no indication that a regeneration is required. For that reason you should try to access your model as much as possible from Xpand or Xtend, and only resort to JAVA code when it is unavoidable.

3.2. Performance Considerations

The main performance benefits of incremental generation come from *not* doing things that are not necessary. Given the workflow from Section 2.2, “The Incremental Generation Callback”, it may seem counterproductive

to first perform a costly model comparison operation before it can even be determined whether a file has to be generated or not.

While it is true that model comparison is a very expensive operation, it turns out that it still outweighs the costs of unnecessarily generating files, even if no postprocessing or subsequent compiler invocation is involved.

That said, it is definitely preferable to do without a model comparison and rather capture the changes to the model on-the-fly. So whenever you are working in a controlled environment, you may want to consider a customized integration of the generator invocation with the model editors.