Chapter 1. Workflow Reference

1.1. Introduction

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

1.2. Workflow components

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

```
public interface WorkflowComponent {

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

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

}
```

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

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

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

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

1.2.1. Workflow

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

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

```
public void addComponent(WorkflowComponent comp)</para>
```

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

1.2.2. Workflow Components with IDs

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

1.2.3. More convenience

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

1.3. Workflow Configuration

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

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

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

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

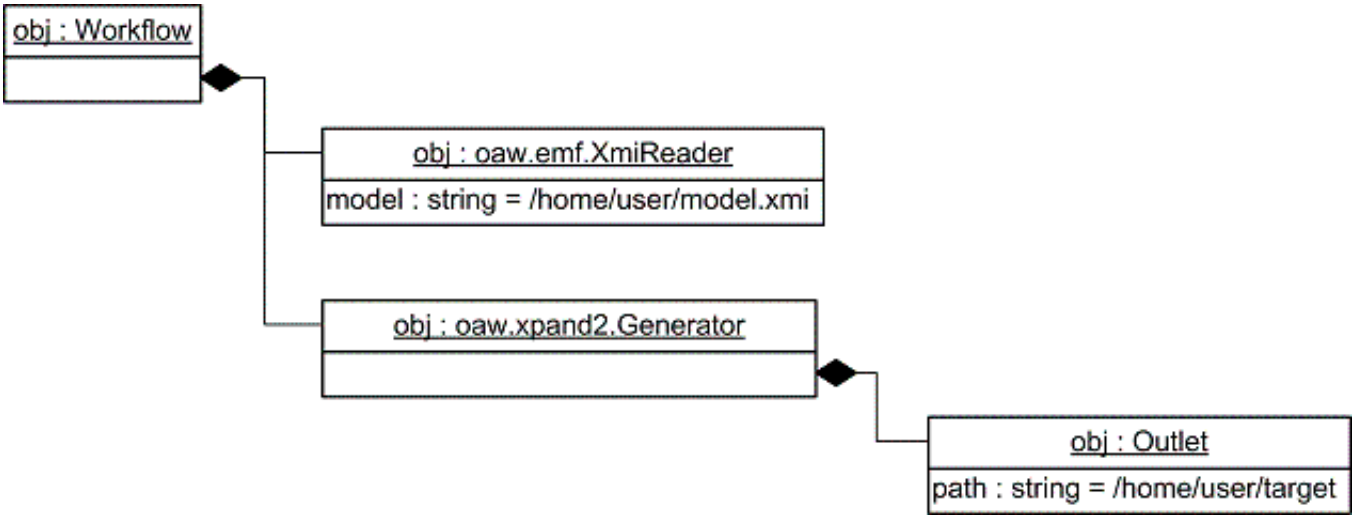

Figure 1.1. Java Object Graph

The configuration language expresses four different concepts:

1.3.1. Properties

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

We have two different kinds of properties

1. simple properties

2. property files

Here is an example:

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

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

Example:

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

1.3.1.1. Components

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

Example:

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

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

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

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

1.3.1.1.1. Accessor methods

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

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

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

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

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

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

1.3.1.2. Component creation

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

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

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

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

1.3.1.3. References

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

Example:

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

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

1.3.1.4. Simple Parameters

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

Example:

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

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

1. using an XML attribute

2. using a nested XML element with an attribute value

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

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

1.3.1.4.1. Converters

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

1. `Object`

2. `String`

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

4. `Boolean` (both primitive and wrapper)

5. `Integer` (both primitive and wrapper)

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

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

file 1: mybean.mwe

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

file 2: workflow.mwe

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

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

file 1: mybean.mwe

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

file 2: workflow.mwe

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

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

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

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

1.3.1.6. InheritAll Feature

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

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

1.3.2. Component Implementation and Workflow Execution

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

1.3.2.1. The Workflow Context

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

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

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

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

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

```
public class XmiReader implements WorkflowComponent {

 private String outputSlot = null;

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

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

}
```

The checker component reads the model from that slot:

```
public class Checker implements WorkflowComponent {

 private String modelSlot;

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

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

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

1.3.2.2. Issues

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

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

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

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

1.3.2.3. Controlling the Workflow

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

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

1.3.2.3.1. Using Aspect Orientation wih Workflows

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

```
public class GeneratorAdvice extends AbstractWorkflowAdvice {

 private String advices;

 public String getAdvices() {
  return advices;
 }

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

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

}
```

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

```
<workflow>

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

1.3.3. Invoking a workflow

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

1.3.3.1. Starting the WorkflowRunner

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

```
java org.eclipse.emf.mwe.core.WorkflowRunner path/workflow.mwe
```

You can override properties using the -p option:

```
java org.eclipse.emf.mwe.core.WorkflowRunner -pbasedir=/base/ path/workflow.mwe
```

1.3.3.2. Starting with Ant

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

```
 <target name='generate'>
  <taskdef name="workflow" classname="org.eclipse.emf.mwe.core.ant.WorkflowAntTask"/>
   <workflow file='path/workflow.mwe'>
    <param name='baseDir' value='/base/'/>
   </workflow>
   ...
  </target>
```

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

1.3.3.3. Starting from you own code

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

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

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

1.3.3.4. Starting from Eclipse

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