

openArchitectureWare 4.1.2

Standard Library Reference

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

Table of Contents

WHAT'S THIS?.....	3
CLONING (EMF ONLY).....	3
COUNTER.....	3
ELEMENT PROPERTIES.....	3
GLOBAL VARIABLES.....	4
IO.....	4
ISSUES.....	4
NAMING (EMF ONLY).....	5
MIXINS (EMF ONLY).....	6

What's this?

The standard library contains a number of useful extensions that you can use for building your own generators and transformations. This document explains the various utilities it provides.

Cloning (EMF Only)

```
extension org::openarchitectureware::util::stdlib::cloning;
```

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 object is not cloned* (a reference to the original is created in the new object)

Counter

```
extension org::openarchitectureware::util::stdlib::counter;
```

This will associate a counter with a given model element. *element.counterInc()* will increment the counter by one and return the current value. *element.counterReset()* will reset the counter to zero and return zero.

Element Properties

```
extension org::openar...tureware::util::stdlib::elementprops;
```

This allows you to temporarily associate name-value pairs with any model element. *element.setProperty(String name, Object value)* sets the property named *name* to the *value*. *Object element.getProperty(String name)* returns the value of the property named *name*. Note that, since the implementation is based on a static hash map, the values are kept through a complete oAW workflow run (you can set them in a transformation and retrieve them later during code generation). Also, because of the static hash map, this feature is not threadsafe!

Global Variables

```
extension org::openarchitectureware::util::stdlib::globalvar;
```

While Element Properties associate name-value pairs with a given model element, Global Variables provide name-value pairs that are not associated with any model element. `storeGlobalVar(String name, Object value)` creates global variable, and `getGlobalVar(String name)` returns the value of one.

IO

```
extension org::openarchitectureware::util::stdlib::io;
```

Supports logging/ println-debugging of values:

- `someExpression.debug()` logs `someExpression.toString()` with DEBUG priority
- `someExpression.info()` logs `someExpression.toString()` with INFO priority
- `someExpression.error()` logs `someExpression.toString()` with ERROR priority
- `someExpression.syserr()` prints `someExpression.toString()` to `System.err`
- `someExpression.syserr("prefix")` prints "[prefix]: "+`someExpression.toString()` to `System.err`

Note that each of these functions return 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 embedd one of these io functions anywhere such as in:

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

Issues

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

This allows you to report workflow issues from within a transformation or code generation templates. Note that this is not to encourage you not to use constraint checks and generally raise the errors directly from within the transformations. However, sometimes it is sensible and useful to be able to do that. There are four functions provided:

- `reportError(String message)` creates a workflow `ERROR` with the `message`
- `reportError(Object context, String message)` creates a workflow `ERROR` with the `message`, also outputting the context's qualified name (see Naming below)

- `reportWarning(String message)` creates a workflow `WARNING` with the `message`
- `reportWarning(Object context, String message)` creates a workflow `WARNING` with the `message`, also outputting the context's qualified name (see Naming below)

Note that in order for this feature to work, you have to run the `ExtIssueReporter` component somewhere in the workflow before the component that executes the transformation or template in which you use any of these four functions, as in

```
<workflow>

  <component id="XmiReader.uml2model"
            class="oaw.emf.XmiReader">
    ...
  </component>

  <component class="org.openarchitectureware.util.
                stdlib.ExtIssueReporter"/>

  <component id="XtendComponent.uml2ecore"
            class="oaw.xtend.XtendComponent">
    ... // report issues from within here
  </component>

</workflow>
```

Naming (EMF Only)

```
extension org::openarchitectureware::util::stdlib::naming;
```

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. In order for this to work, model elements are expected to have a `name` attribute of type `EString` (note: it's not an accident that the `uml2ecore` utility can add such a name attribute to every meta class automatically).

The utility contains the following functions:

- `element.qualifiedName()` returns the qualified name in the sense explained above
- `element.namespace()` returns the namespace, i.e. the qualified name minus the name of the element itself
- `element.loc()` tries to build a useful description of an element in the model; very useful for error reporting

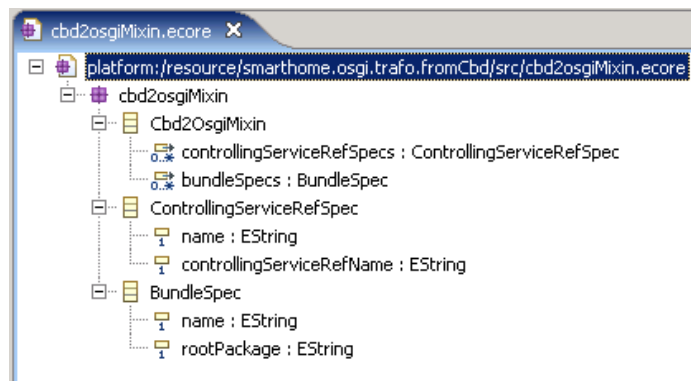
- `collection.findByName(String name)` searches the *collection* for an element named *name*
- `element.findChildByName(String name, oaw::Type t)` searches all the children of element for an element of type *t* named *name*

Mixins (EMF Only)

```
extension org::openarchitectureware::util::stdlib::mixin;
```

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:



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 following functions:

- `mixinModelRootElement.getMandatoryMixin(Object context, 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 *context*. If none is found, an workflow ERROR is raised and a null object is returned (so you can call additional operations on it without getting a null evaluation error).
- `mixinModelRootElement.getOptionalMixin(Object context, oaw::Type t)`: same as above, but does not raise an error in case nothing is found