ECOOP 2006 ETX Submission

openArchitectureWare

a flexible Open Source platform for
model-driven software development

(c) 2006 Markus Völter, Independent Consultant
Heidenheim, Germany, +49 171 86 01 869
voelter@acm.org, www.voelter.de

Abstract

openArchitectureWare (oAW) is a suite of tools and components assisting with model driven software development built upon a modular MDA/MDD generator framework implemented in Java supporting arbitrary import (model) formats, meta models, and output (code) formats. Supportive tools (such as editors and model browsers) are based on the Eclipse platform. openArchitectureWare is a "tool for building MDSD/MDA tools". At the core there is a workflow engine allowing the definition of transformation workflows as well as a number of prebuilt workflow components that can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code.

openArchitectureWare [oAW] is part of the Eclipse GMT [GMT] project.

Topic Area

Modeling environments and frameworks

Description

Here’s a list of the core features. The **workflow engine** precisely controls the generator's workflow, as specified in an XML workflow definition.

With a suitable instantiator, oAW can **read any model**. Currently, we provide out-of-the-box support for EMF, various UML tools (MagicDraw, Poseidon, Enterprise Architect, Rose, XDE, Innovator, …), the Eclipse UML2 project, textual models (using JavaCC or antlr parsers as frontends), XML, Visio as well as pure::variants variant configuration models.

Can generate any kind of output, using a **powerful template language called Xpand** that supports template polymorphism, template aspects and many other advanced features necessary for building non-trivial code generators such as optional typing of model elements and sorting and filtering of element sets.

The metamodel of the problem domain is represented as Java classes. Metamodels can be built using either **Eclipse EMF or oAW's own metamodel generator** that creates those Java classes from metamodels rendered in UML. The generator already comes with the most important UML metaclasses, which can be reused to build UML profiles.

Various ways of **checking model constraints** are available. A semi-declarative checking using Java APIs, a proprietary declarative constraints checking language as well as optional OCL integration based on the Kent OCL framework [OCL].

You can read **several models as part of one generator run**. These models can use different concrete syntaxes (one could be UML, another XML and a third one Visio). The generator then "weaves" these models together to form a comprehensive, all-encompassing model. Constraints can be checked over model boundaries; references among models can be established.

Model-to-model transformations can be implemented using the Wombat language, a textual, functional transformation language. There are two very important properties of this language: first, it has a very concise and powerful syntax. Second, it can transform between the various metametamodels, *e.g.,* you can transform a model defined with the oAW-classic metametamodel into an EMF model! Alternatively you can integrate third party transformation languages such as ATL [ATL]. An Adapter is provided.

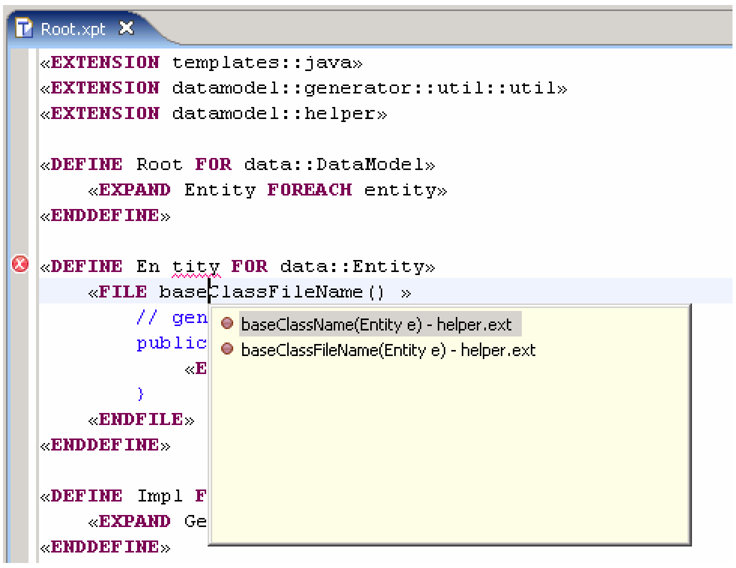
The Recipe Framework allows you to define validation rules for artefacts created outside of the generator (such as manually written subclasses). During code generation, these rules can be instantiated; later, the Eclipse IDE will read these checks and verify them. This will help to guide developers beyond the modelling/generation stage.

openArchitectureWare has a complete **modular design**: you can enhance or extend any of the framework components as you see fit.

It is also worth noting that openArchitectureWare comes with **several hundred pages of documentation**, allowing people to make use of the powerful features of openArchitectureWare.

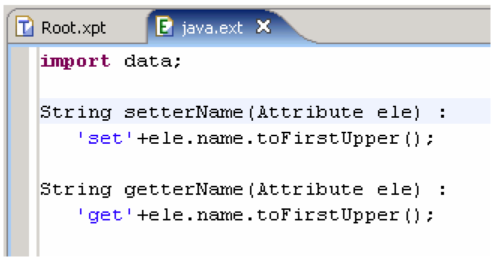
Eclipse IDE Integration

openArchitectureWare comes with a number of Eclipse Plugins that help make development more efficient. Here are a couple of screenshots. The following one shows the Xpand template editor. In addition to syntax highlighting, it provides metamodel-aware code completion facilities as well as static error checking.


Root.xpt
baseClassName(Entity e) - helper.ext
baseClassFileName(Entity e) - helper.ext

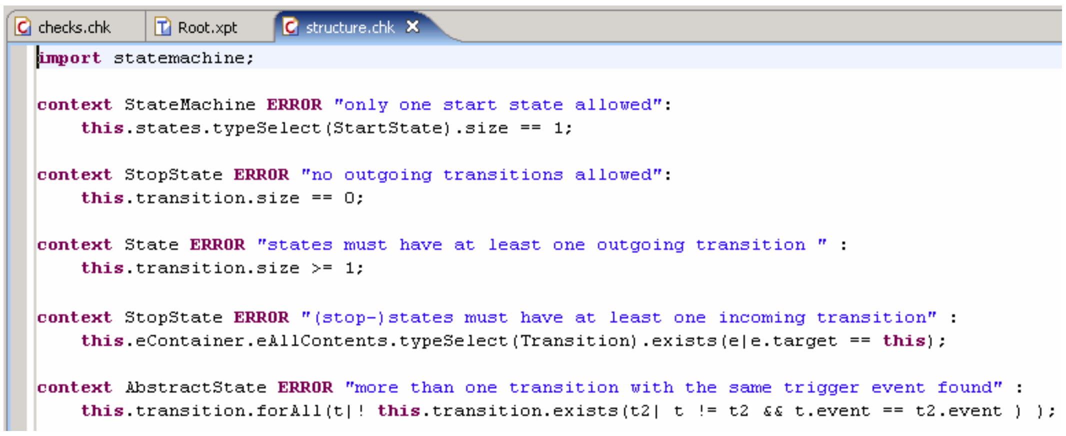
```
«EXTENSION templates::java»
«EXTENSION datamodel::generator::util::util»
«EXTENSION datamodel::helper»

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

«DEFINE En tity FOR data::Entity»
    «FILE baseClassFileName() »
        // gen
        public
            «E
        }
    «ENDFILE»
«ENDDEFINE»

«DEFINE Impl F
    «EXPAND Ge
«ENDDEFINE»
```

You can also define metamodel extensions, *i.e.*, properties on metaclasses, that are available in the template language. They are defined externally to the metamodel implementation classes. Again, the editors provide static type checking and code completion.



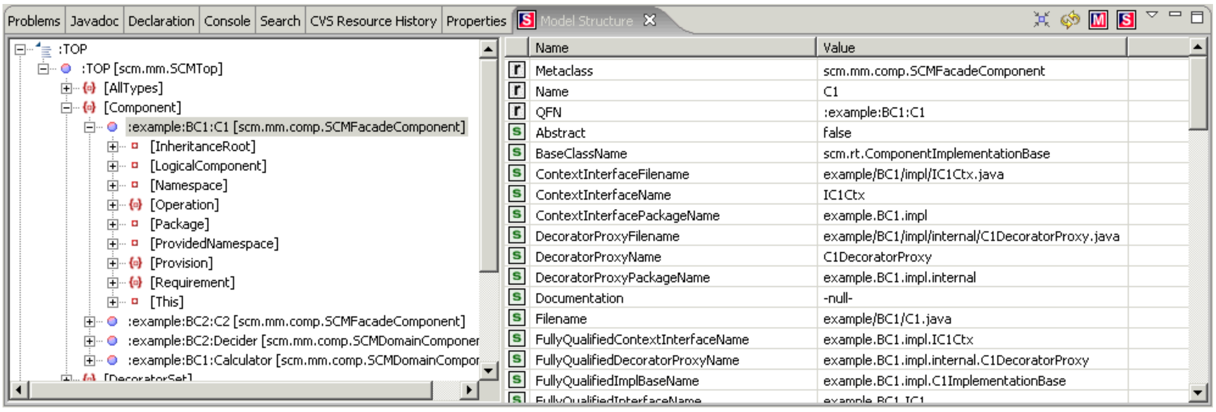
```
import data;

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

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

The following screenshot shows the editor for the OCL-like constraints check language. Here, too, error checking and code completion are supported.



```
import statemachine;

context StateMachine ERROR "only one start state allowed":
    this.states.typeSelect(StartState).size == 1;

context StopState ERROR "no outgoing transitions allowed":
    this.transition.size == 0;

context State ERROR "states must have at least one outgoing transition " :
    this.transition.size >= 1;

context StopState ERROR "(stop-)states must have at least one incoming transition" :
    this.eContainer.eAllContents.typeSelect(Transition).exists(e|e.target == this);

context AbstractState ERROR "more than one transition with the same trigger event found" :
    this.transition.forAll(t|! this.transition.exists(t2| t != t2 && t.event == t2.event ) );
```

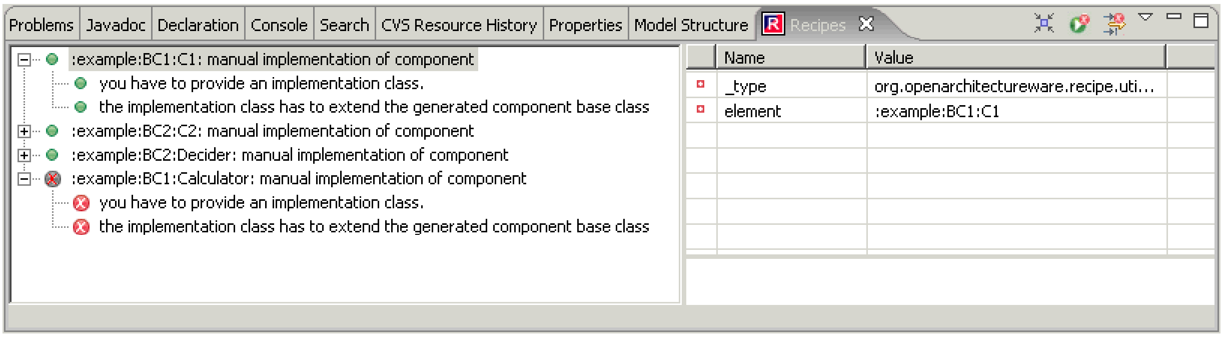
A model structure view helps you view and understand models that have been parsed from UML or other sources. For EMF based models this is not necessary, since the EMF-supplied editors provide exactly the same functionality.



```
:TOP
  :TOP [scm.mm.SCMTop]
    [AllTypes]
    [Component]
      :example:BC1:C1 [scm.mm.comp.SCMFacadeComponent]
        [InheritanceRoot]
        [LogicalComponent]
        [Namespace]
        [Operation]
        [Package]
        [ProvidedNamespace]
        [Provision]
        [Requirement]
        [This]
      :example:BC2:C2 [scm.mm.comp.SCMFacadeComponent]
      :example:BC2:Decider [scm.mm.comp.SCMDomainComponer
      :example:BC1:Calculator [scm.mm.comp.SCMDomainCompor
    [DecoratorSet]
```

	Name	Value
r	Metaclass	scm.mm.comp.SCMFacadeComponent
r	Name	C1
r	QFN	:example:BC1:C1
S	Abstract	false
S	BaseClassName	scm.rt.ComponentImplementationBase
S	ContextInterfaceFilename	example/BC1/impl/IC1Ctx.java
S	ContextInterfaceName	IC1Ctx
S	ContextInterfacePackageName	example.BC1.impl
S	DecoratorProxyFilename	example/BC1/impl/internal/C1DecoratorProxy.java
S	DecoratorProxyName	C1DecoratorProxy
S	DecoratorProxyPackageName	example.BC1.impl.internal
S	Documentation	-null-
S	Filename	example/BC1/C1.java
S	FullyQualifiedContextInterfaceName	example.BC1.impl.IC1Ctx
S	FullyQualifiedDecoratorProxyName	example.BC1.impl.internal.C1DecoratorProxy
S	FullyQualifiedImplBaseName	example.BC1.impl.C1ImplementationBase
S	FullyQualifiedInterfaceName	example.BC1.IC1

The checks verified by the recipe framework can be rendered nicely in an Eclipse view. Changing something in the manually written source code will update the checks in real time.



```
● :example:BC1:C1: manual implementation of component
    ● you have to provide an implementation class.
    ● the implementation class has to extend the generated component base class
● :example:BC2:C2: manual implementation of component
● :example:BC2:Decider: manual implementation of component
⊗ :example:BC1:Calculator: manual implementation of component
    ⊗ you have to provide an implementation class.
    ⊗ the implementation class has to extend the generated component base class
```

	Name	Value
▫	_type	org.openarchitectureware.recipe.uti…
▫	element	:example:BC1:C1

Uses

oAW is used in all kinds of domains – specifically, outside the mainstream. Examples include J2EE-based banking and insurance web applications, Spring-based applications, Eclipse-based Rich client apps in various domains, C/C++ based realtime systems in the automotive domain, data management and conversion in C++/Java/Python-based radio astrononomy systems, .NET applications, Enterprise SOA architecture, etc.

References

ATL	Atlas Transformation Language, http://www.eclipse.org/gmt/atl
GMT	Generative Model Transformer Project, http://www.eclipse.org/gmt
oAW	openArchitectureWare, http://www.openarchitectureware.org
OCL	Univeristy of Kent, Kent OCL Project, http://www.cs.kent.ac.uk/projects/ocl/