Orientation in Objects

Tutorial oAW Cartridge Entwicklung

Eine kompakte Einführung in die Erstellung einer openArchitectureWare Cartridge

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Zoltan Horvath
Orientation in Objects GmbH

Veröffentlicht am: 31.8.2009

ERSTELLUNG EINER OPENARCHITECTUREWARE CARTRIDGE

Dieses Tutorial beschreibt die wichtigsten Schritte zur Erstellung einer Cartridge für openArchitectureWare (oAW). Nach einer kurzen Einführung in das Generatorframework wird anhand eines Beispieles Schritt für Schritt gezeigt, welche Bestandteile benötigt werden, um aus einem UML2-Modell Code zu generieren.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

EINLEITUNG

oAW [1] ist ein modulares, in Java implementiertes Generator Framework, mit dem sich MDSD Werkzeuge erstellen lassen, und ist unter der Open Source Lizenz Eclipse Public License frei verfügbar. Seine Werkzeuge wie Editoren oder Modellnavigatoren sind selbst Eclipse-Plugins und Teil des Eclipse Modeling Project[2]. oAW bietet die Möglichkeit beliebige Modelle zu parsen. Das Framework besitzt eine eigene Sprachfamilie, um diese eingelesenen Modelle zu überprüfen, sie in andere Modelle zu transformieren und aus ihnen Code zu generieren. Gesteuert wird der Generierungsprozess durch eine Workflow-Engine.

Um Code für spezielle Technologien zu generieren werden eigene Module benötigt, sogenannte Cartridges. oAW stellt hierzu das Generator-Framework zur Verfügung, besitzt jedoch out-of-the-box keine solcher Cartridges. Diese müssen selbst erstellt werden.

Daher hat dieses Tutorial zum Ziel, durch ein übersichtliches und praxisnahes Beispiel die wichtigsten Schritte zum Erstellen einer eigenen Cartridge strukturiert mittels Codebeispielen und Diagrammen aufzuzeigen. Die Beispielcartridge, die noch einige Erweiterungen enthält, kann als Eclipse-Projekt in Form einer Zip Datei auch heruntergeladen und als Vorlage für zukünftige Projekte verwendet werden.

OPENARCHITECTUREWARE

Vor der Erstellung einer Cartridge, werden zunächst der Ablauf der Codegenerierung mit oAW sowie ihre Bestandteile näher beleuchtet.

ABLAUF DER CODEGENERIERUNG

Gesteuert und kontrolliert wird der Generierungsprozess durch das Herzstück von oAW, die Workflow-Engine. Sie enthält fertige Komponenten für folgende, grundlegende Abläufe:

1. Einlesen und Instanziieren der Modelle
2. Modellvalidierung
3. Modell-zu-Modell Transformation
4. Modell-zu-Code Transformation
5. Überprüfung und Formatierung der generierten Dateien

Das Modell wird zunächst vom Workflow durch einen Modell-Instantiator eingelesen und geparst. Ein Metamodellinstantiator verwebt die Metamodelle auf denen das Modell basiert, um das eingeparste Eingabemodell zu einer einzigen Metamodellinstanz (= Modell) zu verweben.

Begriffe

Modell und Metamodell: In der Modellgetriebenen Softwareentwicklung stellt das Modell ein Abbild des zu entwickelnden Systems dar. Ein Metamodell legt für ein Modell fest, welche Elemente es haben kann und welche Beziehungen diese zueinander besitzen, d.h. es legt ihre abstrakte Syntax fest. Elemente eines Modells können als Instanzen der Metamodellelemente betrachtet werden.

Domain Specific Language (DSL): Eine DSL ist eine formale Sprache, die speziell für eine bestimmte Domäne entworfen ist und kann somit auch als Metamodell verstanden werden. Sie kann sowohl grafisch als auch textuell sein. In diesem Tutorial wird die DSL mit UML2 beschrieben, da die UML ein Standard ist und durch viele grafische Tools beim Erstellen der Modelle unterstützt wird.

Zur Generierung des Quellcodes benutzt oAW eine Template-Engine. In den Templates ist definiert, wie aus den Elementen des Modells Elemente und Strukturen des Zielmodells (Quellcode) generiert werden sollen. Mittels des Codegenerators werden die Templates auf die geschaffene Metamodellinstanz angewendet, um schliesslich den Quellcode zu erzeugen.

Abbildung1 verdeutlich den Ablauf der Codegenerierung mit oAW:

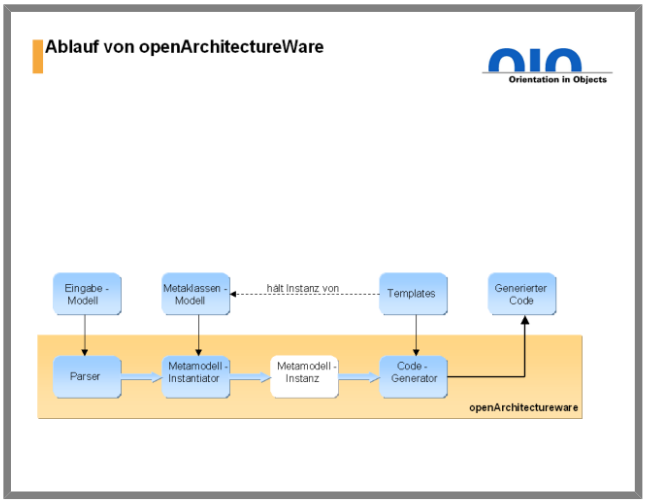

Abbildung 1: Ablauf der Codegenerierung

BESTANDTEILE VON OPENARCHITECTUREWARE

Im Folgenden werden die wichtigsten Bestandteile von oAW erklärt, welche benötigt werden, um erfolgreich eine eigene Cartridge zu erstellen.

DIE OAW EXPRESSION LANGUAGE

Alle folgenden oAW-eigenen Sprachen (XPand2, Xtend, Check) basieren auf einer gemeinsamen Expression-Language. Daher können sie auf gleichen Modellen und Metamodellen operieren. Sie besitzen alle die gleiche Syntax. Über Eclipse Plugins werden für diese Sprachen Editoren bereitgestellt, die Syntax-Erkennung, Codevervollständigung und Fehleranzeige bereitstellen.

TEMPLATES

Als Templatesprache verwendet oAW eine eigens entwickelte Sprache namens XPAND2. Die oAW XPAND2-Templates sind Schablonen, die den Text enthalten, welcher während der Modell-zu-Code Transformation in die Ausgabedateien geschrieben wird. Zudem unterstützt sie Konzepte der aspektorientierten Programmierung (AOP) sowie Polymorphismus, um auf den Modellen navigierend komplexe Code-Generatoren erzeugen zu können.

EXTENSIONS

Durch die oAW XTend Sprache lassen sich Templates und Metamodellelemente um Funktionalität erweitern. In XTend können Funktionen implementiert werden um gewisse Probleme zu verarbeiten, die in den Templates selbst nicht lösbar sind. Diese Extensions können um Java-Hilfsklassen erweitert werden, um dadurch zusätzliche Funktionalität zu erhalten.

CHECKS

Mittels der Check-Language bietet oAW die Möglichkeit nach dem Einlesen des Modells diesen auf bestimmte Bedingungen oder innere Zusicherungen zu überprüfen. Da bei einem Generierungsprozess ein Modell mehrmals durchlaufen und auch transformiert werden kann, hat oAW mit auf diese Art die Möglichkeit, diese Modelle mehrmals auf Bedingungen zu überprüfen. Als Alternative kann auch auf die Object Constraint Language (OCL) oder die Java-API verwendet werden.

WORKFLOW

Für den Ablauf der Generierung ist die Workflow-Engine zuständig. Sie beschreibt die Steuerung des Generierungsprozesses. Auch Konfigurationen und ggf. globale Variablen werden hier gesetzt. Die benötigten Komponenten des Generators werden hier verbunden bzw. aufgerufen.

MODELLIERUNG

Mit oAW soll der Prozess der domänenspezifischen Modellierung abgedeckt werden. Bei der Beschreibung einer Software durch ein Modell wird der Entwickler häufig auf das Problem stoßen, dass vorhandende Metamodelle oft nicht ausreichen die Anforderungen seiner Domäne ausführlich beschreiben zu können. Dazu sind individuelle Anpassungen des Metamodells nötig. Mittels verschiedener zur Verfügung stehender Modell-Instanziierer ist oAW nicht von einem bestimmten Modellformat abhängig. Eine besonders starke Unterstützung bietet oAW für EMF basierte Modelle, jedoch können auch andere Formate wie beispielsweise UML2, XML oder textuelle Modelle verarbeitet werden. Im Rahmen dieses Artikels wird das UML2 Metamodell verwendet, welches sich auf eine einfache Art durch UML-Profile erweitern lässt. Dadurch entsteht eine für das Beispiel passende DSL, mit der das Modell beschrieben werden kann. oAW hat zum Ziel das Modell auf eine konkrete Zielplattform zu überführen.

VORÜBERLEGUNGEN

Im folgenden Praxisteil dieses Artikels werden die wichtigsten Bestandteile und Abläufe einer Cartridge behandelt. Vorher müssen jedoch einige Überlegungen und Entscheidungen betreffend der Modellierung, dem Umfang der Funktionalität der Cartridge und der Zielarchitektur getroffen werden. Dazu müssen auch folgende Punkte geklärt werden:

1. Was soll generiert werden?

   Ziel der MDSD ist meist nicht die Erstellung von fertiger Software. Bestimmte Teile der Fachlogik können durch die unzureichenden Möglichkeiten der Beschreibung durch die Modellsprache nicht dargestellt werden. Diese muss also teilweise manuell implementiert werden.

   Was generiert werden soll, d.h. die Zielarchitektur, muss feststehen. In der MDSD empfiehlt sich oft die manuelle Erstellung einer Referenzimplementierung. Diese soll natürlich nicht schon die gesamte Software darstellen, sondern lediglich die wichtigsten Strukturen, beispielsweise ein Modul der Software. Dieses kann dann als Vorlage für die Templates dienen.

2. Was ist die DSL?

   Wie bereits erwähnt, kann oAW unterschiedliche Modellformate verarbeiten. Entwickler bevorzugen häufig als DSL die UML mit eigens definierten Profilen für die Modellbeschreibung. In UML-Profilen können individuelle Erweiterungen in Form von Stereotypen und Tagged Values (Eigenschaftswerten) realisiert werden. Auf die starke EMF Unterstützung muss jedoch durch die Wahl der UML als DSL nicht verzichtet werden. Gängige UML-Modellierungstools wie MagicDraw bieten die Möglichkeit, UML-Modelle nach EMF-UML2 (eine Implementation von UML auf Basis von EMF als Metamodell) zu exportieren, wodurch sich beide Vorteile kombinieren lassen. Da viele Projekte, wie beispielsweise die von Fornax[3] entwickelten Cartridges, auch auf diese Strategie zurückgreifen, wird sie auch in diesem Artikel verwendet.

3. Funktionsumfang der Cartridge

   Die Beispielcartridge soll alle elementaren Funktionen eines oAW Codegenerators erfüllen. Das bedeutet im einzelnen: Es soll ein auf EMF basierendes Modell eingelesen werden, in dem der Problemraum beschrieben ist. Durch XPand2 Templates, dessen Funktionsumfang wir durch Xtend und Java Extensions für bestimmte Problemfälle erweitern, soll schliesslich der gewünschte und passende Code generiert werden. Auf Modellvalidierungen durch die Check-Language wird in diesem Artikel verzichtet. Jedoch sind im angehängten, erweiterten Projekt Beispiele vorhanden.

SYSTEMVORAUSETZUNGEN

Es gelten folgende Systemvoraussetzungen und installierte Software für die neue Cartridge:

- Java ab Version 6
- MagicDraw (aktuellste Version empfohlen). Eine kostenlose Community-Edition befindet sich zum Download unter [4]. Als alternatives, kostenloses Modellierungswerkzeug kann auch der Open Source Topcased UML Editor[5] verwendet werden, der direkt die Infrastruktur der Eclipse IDE nutzt.
- Eclipse IDE (3.4.x) mit installiertem oAW Plugin (Version 4.3) inclusive der benötigten Dependencies. Zum Download und Installation siehe [1].

ZIELARCHITEKTUR

Bevor man nun endgültig mit der Entwicklung der Cartridge beginnt, muss die Frage "Was soll generiert werden?" beantwortet werden: Die neue Cartridge soll eine einfache JPA Entität generieren können:

```
@Entity @Table(name = "USER") public class Person { }
```

Beispiel 1: Zielcode

Für jede UML-Klasse die man modelliert, wird eine Java-Klasse mit dem entsprechenden Namen in ein Verzeichnis entsprechend dem UML-Package generiert. Zusätzlich lassen sich die UML-Klassen über einen Stereotypen als Entität kennzeichnen und der Tabellenname dieser Entität soll mittels eines Tagged Values selbst definiert werden können. Diese beiden Erweiterungen werden schliesslich als JPA Annotationen in die Klasse generiert.

ERSTELLEN DER CARTRIDGE

Jetzt kann die eigentliche Cartridgeerstellung beginnen. Die korrekte Installation der benötigten Software ist Voraussetzung.

PROJEKTE ANLEGEN

Für die neue Cartridge in der Eclipse IDE unter 'File->New->Project' 'openArchitectureWareProject' wählen. Dadurch wird ein neues Projekt erstellt, was automatisch alle benötigten Projektabhängigkeiten für oAW und den damit verbundenen Cartridgebau einbindet.

Nun werden alle benötigten Source-Verzeichnisse erstellt, in denen die einzelnen Komponenten der Cartridge erstellt werden. Diese Verzeichnisse sind:

- src/model

  Enthält das Modell und das Metamodel.

- src/templates

  Enthält die Templates.

- src/extensions

  Enthält die Xtend Erweiterungen.

- src/java

  Enthält die Java Erweiterungen.

- src/workflow

  Enthält den Workflow.

Das angelegte Projekt wird alle Resourcen für die Cartridge beinhalten. Für die generierten Dateien wird ein neues Java-Projekt mit dem Namen "de.oio.jpa.test" erstellt. Innerhalb dieses Projektes erstellt man noch ein Source-Verzeichnis namens "src-gen", in das die Dateien generiert werden.

Diese Projektstruktur kann auch aus dem Beispielprojekt im Anhang dieses Artikels entnommen werden.

DSL UND METAMODELL DEFINIEREN

Wie bereits in den Vorüberlegungen erwähnt, ist das Metamodell die UML2. Jedoch lässt sich der Problemraum nicht vollständig damit beschreiben, da noch individuelle Eigenschaften definiert werden sollen. Die Java-Klassen sollen mit zusätzlichen Informationen (den JPA-Annotationen) versehen werden können. Dazu benötigt man noch ein UML-Profil als Erweiterung. Das Profil enthält einen Stereotyp namens "Entity", welches das Modellelement "Class" des UML-Metamodells erweitert. Das bedeutet, dass dieser Stereotyp auf alle Elemente vom Typ Class angewendet werden kann. Zusätzlich enthält Entity einen Eigenschaftswert "tableName", welcher sich später als Tagged Value allen Modellelementen, die den Stereotyp "Entity" besitzen zuordnen lässt. Zum Erstellen wird MagicDraw verwendet (siehe Abbildung 3).

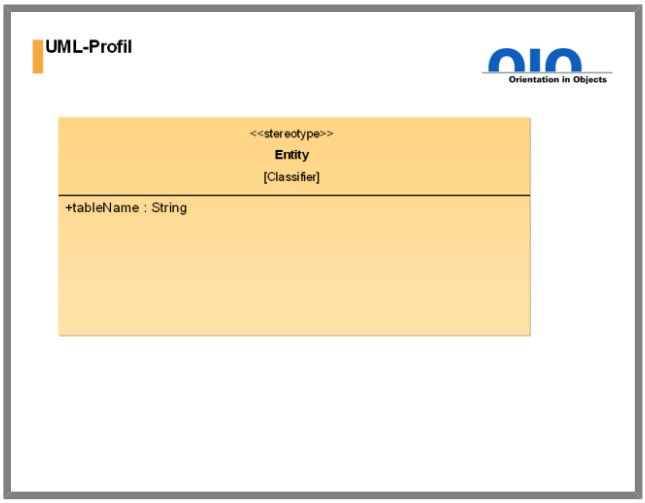

Abbildung 2: UML-Profil

Dieses Profil wird in das "src/model" Verzeichnis des Projektes exportiert. Mit MagicDraw kann man wählen, nach welchem Modellformat exportiert werden soll. Hier wird "emf-uml2" gewählt, um die starke EMF Unterstützung durch oAW zu nutzen.

XPAND2 TEMPLATE ERSTELLEN

Ein Template kümmert sich um die gefundenen Modellelemente sowie Stereotypen und verweist gegebenenfalls auf weitere Templates, um die gefundenen Modellklassen zu verarbeiten. Sie bestehen aus IMPORT Statements, gefolgt von den verwendeten Extension Statements, die auf benötigte Xtend Erweiterungen verweisen sowie ein oder mehrere DEFINE Blöcke (Definitionen). Innerhalb von DEFINE Blöcken können die Texte der Ausgabedateien für die jeweiligen Modellelemente definiert werden. Ein DEFINE Block ist immer einem bestimmten Modellelementtyp zugeordnet und wird durch das EXPAND Statement aufgerufen.

In dem Beispiel sollen Modellelemente vom Typ "Entity" verarbeitet werden, der in dem Profil als Stereotyp definiert ist. Folgender Codeabschnitt zeigt wie das Template aussieht. "Persistence" ist der Name des Profiles, welches als Import eingebunden werden kann. Die verwendete Extension enthält zusätzlich Funktionen, wie beispielsweise die Erzeugung des vollqualifizierten Pfades (getFQPackagePath()) oder Hilfsfunktionen für die Erstellung von getter und setter Methoden. Mehr dazu in den nächsten beiden Abschnitten.

In den ersten beiden DEFINE - Blöcken (Namen sind frei wählbar) werden zuerst für alle Elemente im Modell die Packages selektiert. Im dritten DEFINE-BLOCK wird für jedes gefunde Element aus einem Package, das mit dem Stereotypen "Entity" versehen ist, eine Java-Datei erstellt. Über eine FOREACH-Schleife werden zusätzlich alle Attribute einer modellierten Entität ausgelesen und mit dazugehörigen getter/setter Methoden genereriert. Der letzte DEFINE-Block erstellt für jede dieser Entitäten eine ID.

Auf einen Tagged Value eines Modelltypes lässt sich direkt innerhalb eines solchen Blockes (der ja einem bestimmten Typ zugeordnet ist) zugreifen. Im Beispiel ist "tableName" ein Tagged Value von "Entity". Nun wird das neue XPand2 Template "Template.xpt" im Verzeichnis "src/templates" erstellt.

```
«IMPORT persistence»
«EXTENSION extensions::Extensions»
/**
* The entry point for the generation
*/
«DEFINE main FOR uml::Model»
 «EXPAND package FOREACH
 ownedElement.typeSelect(uml::Package)»
«ENDDEFINE»
/**
* Creates all packages
*/
«DEFINE package FOR uml::Package»
 «EXPAND package FOREACH nestedPackage»
 «EXPAND entity FOREACH ownedElement.typeSelect(Entity)»
«ENDDEFINE»
/**
* Creates all classes that have stereotype Entity
*/
«DEFINE entity FOR Entity»
 «FILE getFQNPackagePath() + "/" + name + ".java"»
 package «getFQNPackageName()»;
  @javax.persistence.Entity
  «IF tableName != null && tableName.length>0»
   @javax.persistence.Table(name="«tableName»")
  «ENDIF»
  public class «name» {
  «FOREACH attribute AS a»
  private «a.type.name» «a.name»;
  «ENDFOREACH»
  «EXPAND id»
  «FOREACH attribute AS a»
  public void «a.setter()»(«a.type.name» «a.name») {
   this.«a.name» = «a.name»;
  }
  public «a.type.name» «a.getter()»() {
   return «a.name»;
  }
  «ENDFOREACH»
  }
 «ENDFILE»
«ENDDEFINE»
/**
* Creates an id for each Enitity.
*/
«DEFINE id FOR Entity»
 @javax.persistence.Id
 @javax.persistence.GeneratedValue
 private int id;
 public int getId() {
  return id;
 }
 public void setId(int id) {
  this.id = id;
 }
«ENDDEFINE»
```

Beispiel 2: Template.xpt

XTEND ERWEITERUNG

Anforderungen, die in Templates nicht gelöst werden, lassen sich in XTend-Erweiterungen behandeln. Ein XTEND Template kann wie eine XPAND2 Template IMPORT und EXTENSION Statements besitzen. In der Extension im Beispiel sind Operationen definiert, die den vollständigen Namen und Pfad einer generierten Datei ermitteln. Die neue Xtend Datei "Extenstions.ext" wird nun im Verzeichnis "src/extensions" erstellt.

```
getter(uml::Property this) :
 "get"+name.toFirstUpper();
setter(uml::Property this) :
 "set"+name.toFirstUpper();
String getFQNPackageName(uml::Type type):
 JAVA
 javaHelper.JavaHelper.getFQNPackageName(org.eclipse.uml2.uml.Type);
String getFQNPackagePath(uml::Type type):
 getFQNPackageName(type).replaceAll("\\.", "/");
```

Beispiel 3: Extension.ext

Die ersten beiden Funktionen sind Hilfsfunktionen, welche aus den Namen der übergeben UML-Property Elemente "getter/- und setter" Methodenköpfe erstellen.

In der dritten Funktion wird mit dem Ausdruck JAVA eine Methode namens getFQNPackageName() mit den übergebenen Parameter uml.Type in der Klasse JavaHelper aufgerufen, welche im Verzeichnis javaHelper liegt. Diese Datei stellt eine Java-Extension dar.

JAVA-ERWEITERUNG

Für komplexe Funktionen sind Java-Hilfsklassen notwendig. Wie im letzten Codeabschnitt zu sehen ist, können Methoden von Javaklassen innerhalb der XTEND-Dateien aufgerufen werden. Die Funktionserweiterung mit Java bietet alle Möglichkeiten, die Java selbst bietet. Die folgende zu erstellende Java-Klasse enthält die Methode, die aus der XTend Datei aufrufen wird:

```
package javaHelper;
import org.eclipse.uml2.uml.Model;
import org.eclipse.uml2.uml.Package;
import org.eclipse.uml2.uml.Type;
public class JavaHelper {
  public static String getFQNPackageName(Type type) {
    String packageName = "";
    Package p = type.getPackage();
    while (p != null) {
      packageName = p.getName() + "." + packageName;
      p = (Package) p.getOwner();
      if (p instanceof Model) {
        p = null;
      }
    }
    if (packageName.endsWith(".")) {
      packageName =
        packageName.substring(0, packageName.length() - 1);
    }
    return packageName;
  }
}
```

Beispiel 4: JavaHelper.java

WORKFLOW

Abschliessend wird die Workflowdatei "generator.oaw" im Verzeichnis "src/workflow" erstellt. Die letzte Komponente der neuen Cartridge, die noch fehlt, ist die Workflow Datei zum Starten und Steuern des Generierungprozesses. Eine Workflowbeschreibung besteht aus einer Liste von Workflow-Komponenten, die sequentiell abgearbeitet werden. Daher ist die Reihenfolge von Bedeutung.

Die Workflow-Datei aus diesem Beispiel besteht im Wesentlichen aus zwei solchen Komponenten. Die erste liest das Modell mittels eines XMI-Readers ein und speichert es in einem "Slot". Der "Slot" bezeichnet den Speicherort unseres Modells.

In der zweiten Workflowkomponente wird der Generatorvorgang gestartet. Zunächst müssen verwendete Metamodelle registriert werden. Da das eingelesene Modell vom Typ EMF-UML2 ist, sind folgende (Meta-) Modelle notwendig:

- UML2-Metamodell
- EMF-Metamodell

- Metamodell des Modells - das Profil (persistence_profile)

Die registrierten Modelle werden benötigt, damit die XPAND2-Template-Engine Zugriff auf ihre Elemente hat und in der Lage ist, diese auszuwerten. Im Anschluss wird mit dem "expand" Statement das Template "template.xpt" als Einstiegspunkt für die Codegenerierung gesetzt. Die erzeugten Ausgabedateien werden in das Verzeichnis geschrieben, welches durch outletPath definiert ist. Nachdem die Dateien generiert worden sind, werden sie mittels eines JavaBeautifiers formatiert.

```
<?xml version="1.0"?>
<workflow>
 <!-- the model -->
 <property name="model" value="src/model/SampleModel.uml"
 />
 <!-- project folder for generated files -->
 <property name="src-gen" value="../de.oio.jpa.test/src-gen"
 />
  <!-- UML2 Setup for intializing resources and uris -->
  <bean class="oaw.uml2.Setup" standardUML2Setup="true"/>
  <!-- Definitions of needed metamodels and models -->
  <bean
    id="mm_ecore"
    class="oaw.type.emf.EmfMetaModel">
    <metaModelPackage
    value="org.eclipse.emf.ecore.EcorePackage" />
  </bean>
  <bean
    id="mm_uml2"
    class="oaw.uml2.UML2MetaModel" />
  <bean
    id="persistence_profile"
    class="oaw.uml2.profile.ProfileMetaModel">
    <profile value="src/model/persistence.profile.uml" />
  </bean>
  <!-- load model and store it in slot 'model' -->
  <component class="oaw.emf.XmiReader">
    <modelFile value="${model}" />
    <outputSlot value="model" />
  </component>
  <!-- generate code -->
  <component
  class="org.openarchitectureware.xpand2.Generator">
    <metaModel idRef="mm_ecore" />
    <metaModel idRef="mm_uml2" />
    <metaModel idRef="persistence_profile" />
    <expand
      value="template::Template::main FOR model" />
    <outlet path="${src-gen}">
      <postprocessor
           class="org.openarchitectureware.
                  xpand2.output.JavaBeautifier" />
    </outlet>
  </component>
</workflow>
```

Beispiel 5: generator.oaw

In der folgenden Abbildung ist der Ablauf des Workflows nochmals verdeutlicht:

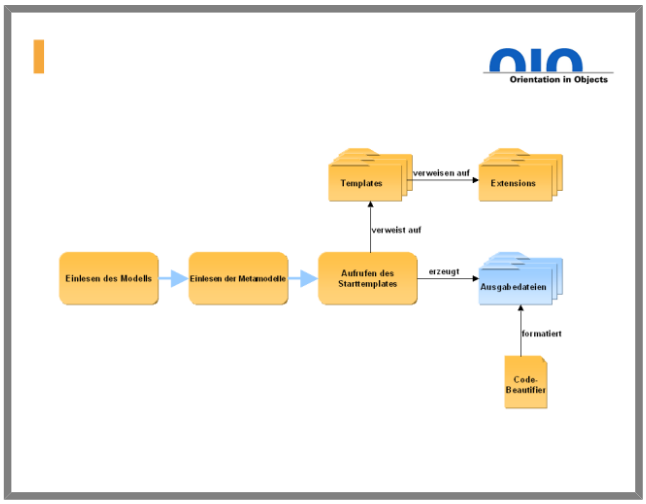

Abbildung 3: Ablauf des Workflows

Eine ausführliche Dokumentation zu allen oAW Komponenten befindet sich unter [1].

VERWENDEN DER CARTRIDGE

Nun kan die Cartridge getestet werden. Alles was dazu noch benötigt wird ist ein Modell.

MODELL ERSTELLEN

Da die DSL definiert ist, kann der Problemraum beschrieben werden, d.h das eigentliche Modell, aus dem der Quellcode entstehen soll, erstellt werden. Das Beispielmodell besteht aus einer UML-Klasse, die durch den im Profil definierten Stereotypen "Entity" gekennzeichnet ist. Somit lässt sich auch dem Tagged Value "tableName" ein Wert zuordnen. Zur Modellierung wird hier auch MagicDraw verwendet. Der Name des Modells ist "SampleModel" (entsprechend der obersten Property in der Workflowdatei, kann natürlich auch angepasst werden). Nicht vergessen darf man hierbei, das erstellte UML2-Profil vorher zu laden, damit die benötigten Stereotypen und TaggedValues angewendet werden können.

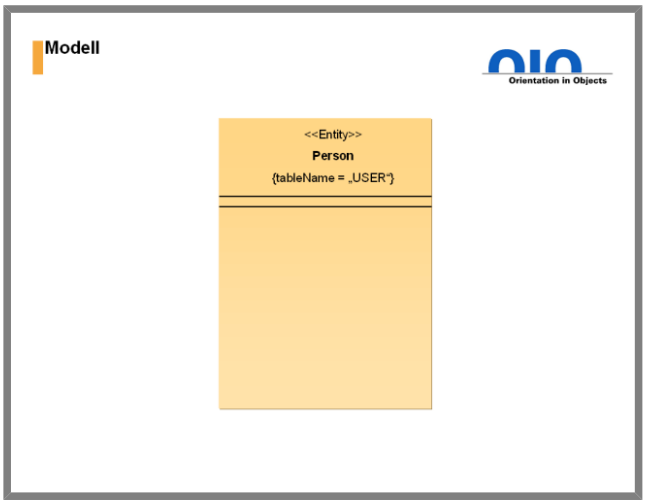

Abbildung 4: UML-Profil

Nun muss das Modell nach "emf-uml2" in das Verzeichnis "src/model" exportiert werden.

GENERATORVORGANG STARTEN

Um den Generatorvorgang zu starten, rechtsklickt man die Datei "generator.oaw" in "src/workflow" und wählt "Run as" -> "oAW Workflow". Die Ausgabedatei wird in das Verzeichnis "src-gen" des Projektes "de.oio.jpa.test" in das entsprechende Package (in der auch das Modell in MagicDraw erstellt wurde) generiert.

Wie zuvor bereits erwähnt, wird in der Regel nicht 100 % des benötigten Codes für ein System vom Generator erzeugt. Die von der Cartridge generierten JPA-annotierten Entitäten können jedoch ohne weitere Bearbeitung in eine Datenbank gespeichert werden. Dazu wird lediglich ein Persistenz-Kontext und ein EntityManager benötigt. Der EntityManager ist ein Element der JavaPersistence API, der persistenzbezogene Funktionalität wie Speichern, Laden oder Löschen von Objekten zur Verfügung stellt. Nähere Informationen zur JPA-API befinden sich unter [7].

```
public static void main(String[] args) throws Exception {
 EntityManagerFactory factory = Persistence
   .createEntityManagerFactory("test");
 EntityManager em = factory.createEntityManager();
 em.getTransaction().begin();
 Person p = new Person();
 em.getTransaction().commit();
 em.close();
}
```

Beispiel 6: Auschnittt aus der Klasse Main.java

Das vollständige Beispiel dazu befindet sich im angehängten Projekt.

ZUSAMMENFASSUNG

Dieses Tutorial soll das benötigte Know-How, sowie die wichtigsten Schritte zum Erstellen einer eigenen openArchitectureWare Cartridge kompakt vermitteln. Die oben entwickelte Cartridge kann Ihnen zusätzlich als Vorlage für weitere oAW Projekte dienen. Sie steht mit einigen zusätzlichen Funktionen im Anhang als kostenloser Download zur Verfügung.

Damit eine Cartridge sinnvoll in einem Projekt verwendet werden kann sind natürlich Erweiterungen notwendig. Beispielsweise kann der Generatorlauf durch initales Prüfen des Eingabemodells verbessert werden. Mittels der Check-Language lassen sich dafür Bedingungen und Regeln definieren. Verstößt das Modell gegen diese festgelegten Constraints, wird der Generatorlauf mit einer Fehlermeldung unterbrochen. Dadurch können Modellierungsfehler von Anfang an verrringert werden.

Ein nächster Schritt für die Erweiterung der Funktionalität der Cartridge könnte die Generierung eines entsprechenden DAOs für jede Entität sein. Dazu würde eine Änderung des beschriebenen Templates genügen.

ANHANG: DIE FERTIGE CARTRIDGE

Download oAW-Cartridge.zip

REFERENZEN

- [1] openArchitectureWare Homepage
  (http://www.openarchitectureware.org/)
- [2] Eclipse Modeling Project
  (http://www.eclipse.org/modeling/)
- [3] Fornax Platform
  (http://www.fornax-platform.org/)
- [4] MagicDraw Homepage
  (http://www.magicdraw.com/)
- [5] Topcased Homepage
  (http://www.topcased.org/)
- [6] Java DB
  (http://developers.sun.com/javadb/)
- [7] Java Persistence API
  (http://java.sun.com/javaee/technologies/persistence.jsp)