Orientation in Objects

Erstellen einer AndroMDA Cartridge

) Schulung)

) Beratung)

) Entwicklung)

AUTOR

Christoph Loewer
Orientation in Objects GmbH

Veröffentlicht am: 3.12.2007

ERSTELLEN EINER ANDROMDA CARTRIDGE

Die modellgetriebene Softwareentwicklung gewinnt in den letzten Jahren immer mehr an Schwung. Das Open Source Framework für Generatoren namens AndroMDA trägt zu diesem Trend maßgeblich bei. Die enthaltene Plug-in Architektur ermöglicht dem Entwickler eigene Plug-ins (Cartridges) zu entwerfen.

Ziel dieses Artikels ist es, dem Leser die benötigten Schritte zur Entwicklung einer eigenen Cartridge aufzuzeigen.

) Artikel)

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

Bei AndroMDA handelt es sich um ein durch UML-Modelle gespeistes Open-Source-Generatorframework, welcher den Ansatz eines Model Driven Architecture (MDA) getriebenen Generator verfolgt. AndroMDA erzeugt mit Hilfe von Modulen, den so genannten Cartridges, den Code zu verschiedenen Zielarchitekturen. Im Standardpaket von AndroMDA 3.x befindet sich bereits eine großzügige Sammlung verschiedener Out-of-the-box Cartridges [1].

Bevor also die Entwicklung neuer Cartridges in Erwägung gezogen wird, sollte ein Blick auf das großzügige Angebot der bereits angebotenen Cartridges geworfen werden [2]. Diese Cartridges gehören ebenfalls zum Open Source Angebot von AndroMDA. Sollten sie Ihre Anforderungen nicht abdecken können, so empfehle ich Ihnen die vorhandenen Cartridges als Vorlage zur Entwicklung Ihrer neuen Cartridges zu verwenden. Selbstverständlich können Sie auch die Cartridge, den praktischen Teil dieses Artikels, in Form einer Zip-Datei herunterladen und dieses Projekt als Vorlage verwenden. Den Link finden Sie am Ende des Artikels.

Ziel dieses Artikels ist es, dem Leser die benötigten Schritte zur Entwicklung einer Cartridge aufzuzeigen. Dies soll anhand einer strukturierten Vorgehensweise mit Zuhilfenahme übersichtlicher Diagramme und Programmausschnitte verständlich und praxisnah verdeutlicht werden.

ANDROMDA

Bevor wir in die Entwicklung einer neuen AndroMDA-Cartridge einsteigen, möchte ich noch die Grundlagen des Generatorframeworks erklären. Hierzu zählen der Aufbau des Frameworks, der Workflow (Ablauf) des Generatorprozesses und die Bestandteile einer Cartridge.

AUFBAU

Wichtigster Bestandteil dieses Codegenerators ist der AndroMDA–Kern. Er ist quasi die Engine des Frameworks und sorgt dafür, daß alle Komponenten von AndroMDA zusammenarbeiten. AndroMDA nutzt das Netbeans MDR Modul [3] um UML 1.4 Modelle, oder EMF um UML 2.0 Modelle einzulesen. Das EMF-Format wird von den im Moment gängigsten Editoren, wie beispielsweise den kostenpflichtigen UML Editoren MagicDraw oder Poseidon, auch von zahlreichen kostenlosen Eclipse Plugins wie Topcased oder dem UML2Tools unterstützt. Eine komplette Liste aller von AndroMDA unterstützten UML-Editoren finden Sie unter [4]. Als empfehlenswert gilt der UML-Editor Magicdraw [5].

Die Cartridges stellen die zentralen Komponenten zur Steuerung der Codegenerierung dar. Sie enthalten alle notwendigen Informationen, um aus einem plattformunabhängigen Modell den plattformspezifischen Code zu erzeugen. Die Codegenerierung erfolgt über eine Template-Engine. In unserem Fall wird das Apache Velocity [6] sein. Aber dazu im nächsten Kapitel mehr.

Gesteuert wird die Codegenerierung in AndroMDA durch das Buildmanagement Tool Maven2 [7] oder Apache Ant. AndroMDA enthält ein Plug-in für Maven, dem bevorzugten Buildtool, der den kompletten Generatorprozeß in AndroMDA steuert. Dies bietet dem Entwickler die Möglichkeit, mittels dieses Plugins ein neues AndroMDA Projekt zu erstellen, um dort zukünftige Anwendungen zu Generieren und zu Deployen. Dem entsprechend werden die Cartridges ebenfalls mit Maven2 erzeugt, um sie somit so einfach wie möglich in den späteren Generatorvorgang zu integrieren.

Folgende Abbildung verdeutlicht den angesprochenen Aufbau von AndroMDA:

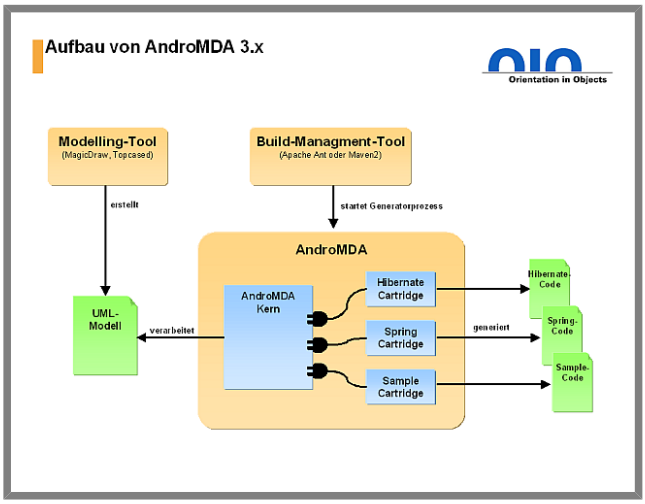

Abbildung 1: Aufbau von AndroMDA 3.x

BESTANDTEILE EINER ANDROMDA CARTRIDGE

Eine handelsübliche Cartridge besteht aus der Modellbeschreibung, den Templates, sowie Deskriptoren in Form von XML – Dokumenten. Die folgende Abbildung zeigt einen Überblick der wichtigsten Bausteine einer Cartridge, welche in den nachfolgenden Absätzen separat erklärt werden.

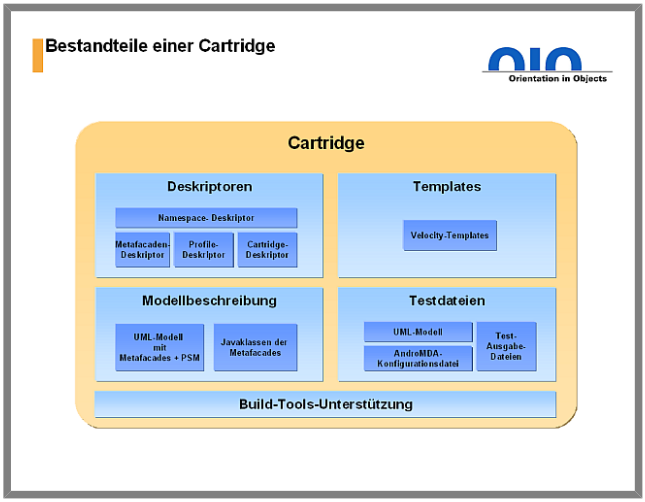

Abbildung 2: Bestandteile einer Cartridge

MODELLBESCHREIBUNG

Eine Metamodellerweiterung eines plattformunabhängigen Modells (PIM) wird in AndroMDA durch ein UML-Profil ermöglicht. Der Zugriff auf das Modell, bzw. auf die Instanzen der Metaobjekte innerhalb der Templates, geschieht über eine Schnittstelle, den sogenannten Metafacades. Sie sind der wesentliche Bestandteil der Modellbeschreibung. Zu jedem Metaobjekt des Modells wird eine passende Metafacade instanziiert. Die Metafacades besitzen dieselben Assoziationen wie die Metaobjekte eines UML-Modells und bieten dem Entwickler eine einheitliche Zugriffsfassade an, die von der konkreten UML Version (UML2.0, UML 1.4, UML, 1.3) abstrahiert (vgl. [8]).

Der Entwickler einer Cartridge kann die Fassaden erweitern, indem er eine neue Metafacade definiert, die von einer passenden Metafacade des AndroMDA Kerns abgeleitet ist. Zwischen den Metafacades kann der Entwickler beliebige Beziehungen erstellen und neue Attribute und Methoden definieren. Somit kann das benötigte Metamodell mit Hilfe dieser Metafacades nachgebildet werden.

Die Modellbeschreibung einer Cartridge besteht aus dem Plattform unabhängigen Modell (PIM), welches durch die Metafacades repräsentiert wird, und aus dem Plattform abhängigen Modell (PSM).

TEMPLATES

Die Templates (Schablonen) ermöglichen das Generieren der technologiespezifischen Ausgabedateien. Ab AndroMDA Version 3.x ist die Template Engine auswechselbar. Zur Auswahl stehen Freemaker10 und Apache Velocity [6], welches standardmäßig ab AndroMDA 3.2 eingesetzt wird. Velocity ist eine auf Java basierende Open-Source Template Engine der Jakarta-Projektgruppe von Apache. Die von der Engine verwendeten Templates sind in der Velocity Template Language (VTL)- einer relativ simplen und leicht verständlichen, aber dennoch umfassenden Skriptsprache - verfaßt.

DESKRIPTOREN

Die Konfiguration der einzelnen Komponenten, speziell der Metafacades und Templates, wird in den Deskriptoren definiert. Die Deskriptoren werden in 4 verschiedene Dateien unterteilt:

Name	Beschreibung
`namespace.xml`	Diese Datei repräsentiert die Schnittstelle zwischen der Cartridge und dem AndroMDA-Kern. Sie enthält Informationen zu den restlichen Deskriptoren und definiert gemeinsame Konfigurationsparameter zur Steuerung der Cartridge.
`metafacades.xml`	Im Metafacade-Deskriptor werden die verwendeten Metafacades und deren Mapping mit den Modellelementen definiert.
`profile.xml`	Diese Datei enthält eine genaue Beschreibung aller verwendeten Stereotypen und Eigenschaftswerten (TaggedValues).
`cartridge.xml`	Der Cartridge-Deskiptor enthält sämtliche Informationen über die zur Verfügung stehenden Templates und deren Input in Form von Metafacades. Mit Hilfe dieser Informationen wird die Model-Code-Transformation definiert.

TESTDATEIEN

In diesem Artikel werden die wichtigsten Bestandteile einer Cartridge behandelt. Auch wenn die Testdateien nicht dazu zählen, sollten sie in einer vollständigen Cartridge nicht fehlen.

Maven2 bietet dem Entwickler einer Cartridge die Unterstützung eines kompletten Buildzyklus, wobei in jedem Zyklus das komplette Cartridge erzeugt und anschließend getestet wird. Beim Ablauf der Tests wird ein einfaches aber effektives Konzept verfolgt. Das zuvor generierte Cartridge wird mit einem zusätzlichen Testmodell gestartet. Der daraus erzeugte Code wird anschließend mit dem Inhalt einer komprimierten Datei verglichen. Diese Datei enthält die erwarteten Ausgabedateien. Sollten die generierten Dateien und deren Verzeichnisstruktur den erwarteten Ausgabedateien entsprechen, so wird der Testvorgang erfolgreich abgeschlossen. Die Testdateien bestehen somit aus den erwarteten Ausgabedateien, dem Testmodell und den, für einen kompletten Generatorlauf notwendigen, Konfigurationsdateien.

EINE NEUE CARTRIDGE ERSTELLEN

Bevor wir in die Entwicklung einer neuen AndroMDA-Cartridge einsteigen, möchte ich noch die Grundlagen des Generatorframeworks erklären. Hierzu zählen der Aufbau des Frameworks, der Workflow (Ablauf) des Generatorprozesses und die Bestandteile einer Cartridge.

VORÜBERLEGUNGEN

Zunächst müssen Sie sich über die Zielarchitektur im Klaren sein. Das heißt, Sie wissen was Sie generieren wollen. Der übliche Ablauf einer modellgetriebenen Softwareentwicklung empfiehlt im ersten Schritt die Umsetzung einer Referenzimplementierung. Dabei muß nicht das komplette Produkt implementiert werden. Lediglich ein wesentlicher Bestandteil, ein Teil oder ein Modul Ihrer Applikation, sollte zuvor manuell von Ihren Systemarchitekten implementiert werden. Diese Referenzimplementierung dient später als Vorlage für Ihre Templates. Nähere Informationen hierzu finden Sie im Buch [18].

Des Weiteren haben Sie bereits eine Vorstellung wie Sie Ihre Anforderungen mit Hilfe der UML Diagramme modellieren können. Warum nur UML Modelle? Bei AndroMDA handelt es sich um ein MDA Werkzeug. Das Konzept der OMG [9] vorgestellten Spezifikation zur MDA basiert auf der Auswertung von MOF basierten Diagrammtypen, zu denen in erster Linie UML-Diagramme zählen. Ziel dieser Spezifikation und Verwendung der UML Sprache ist ein möglichst hoher Grad an Interoperabilität zwischen den einzelnen Tools. Zur Erweiterung der UML Sprache dienen die Stereotypen und deren Eigenschaftswerte, die so genannten TaggedValues.

Die Umsetzung des UML-Profils wird in „UML Profil“ besprochen.

VORAUSSETZUNGEN

Bevor wir mit der Implementierung einer neuen Cartridge starten, sollten Sie die folgenden Voraussetzungen erfüllen:

- Sie kennen sich bereits mit AndroMDA als Generator aus. Beispielsweise haben Sie das benötigte Know-how über das Tutorial [10] erlangt.
- Sie besitzen Grundlagenwissen in den Technologien: Java, UML, XML und Maven2.

Des Weiteren gelten die folgenden Werkzeuge als Systemvoraussetzungen:

- Java (aktuelle Version wird empfohlen) [11]
- Eclipse als IDE (aktuelle Version wird empfohlen) [12]
- Apache Maven2 (aktuelle Version wird empfohlen) [7]

- MagicDraw 9.5 (UML 1.4) für das Metamodell [5]
- MagicDraw 12.x (UML 2.0) für Testmodell [5]

ZIELARCHITEKTUR

Bevor wir den ersten Schritt zur Umsetzung unserer Beispielcartridge unternehmen, möchte ich noch die Zielarchitektur beschreiben. Ich antworte auf die Frage: „Was soll denn generiert werden?“ In unserem Beispiel besteht die Zielarchitektur aus mehreren Textdateien, verteilt in verschiedene Verzeichnisse. Folgender Inhalt soll jede Textdatei enthalten:

```
Hello World. Here is planet <Planetname>
```

Beispiel 1: Inhalt der Ausgabedatei

Als Input dient ein UML-Modell. Für jede UML-Klasse wird eine entsprechende Textdatei generiert, mit dem entsprechenden Namen als Dateiname und dem UML-Package der Klasse als Verzeichnispfad der Datei. Zusätzlich wird über ein Eigenschaftswert (TaggedValue) ein String-Wert, welcher der Name des Planeten ist, eingelesen und in den Inhalt der Datei geschrieben.

EIN NEUES ANDROMDA-CARTRIDGE PROJEKT

Im ersten Schritt erzeugen wir ein neues Java Projekt und alle benötigten „Source“-Verzeichnisse:

- `src/java`

  Enthält Javacode der Metafacades und Hilfsklassen
- `src/resources`

  Enthält die Konfigurationsdateien und das Metamodell
- `src/test`

  Enthält die Testdateien
- `src/META_INF`

  Enthält die Deskriptoren
- `src/templates`

  Enthält die Templates
- `target/src`

  Enthält den erzeugten Javacode der Metafacades und PSM-Objekte

Die Projektsstruktur können Sie auch aus dem Beispielprojekt im Anhang dieses Artikels entnehmen.

DIE BUILD-TOOL-UNTERSTÜTZUNG

Das Buildmanagement der Cartridge wird durch die Open Source Werkzeuge Apache Maven2 oder Ant gesteuert. In unserem Beispiel konzentrieren wir uns auf die Build-Unterstützung mit Maven2.

Maven2 ist ein auf Java basierendes Build-Management-Tool, welches mittels einer Plug-in Architektur der jeweiligen Applikation ermöglicht, gemeinsame Ressourcen zu benutzen. Maven2 bezeichnet diese Ressourcen als Artefakte. Üblicherweise handelt es sich dabei um Java Archive, welche an einem projektübergreifenden Ort, dem Repository, abgelegt werden. In unserem Fall ist die Applikation unser AndroMDA Generator mit all seinen Cartridges. Es existieren Artefakte für den AndroMDA-Kern, die UML-Profile und Cartridges. Nach Abschluß des Buildvorgangs wird die Cartridge in das lokale Repository abgelegt und somit den anderen Applikationen, in unserem Fall dem AndroMDA- Generator, zur Verfügung gestellt.

Sämtliche Maven-relevanten Informationen und Einstellungen eines Projekts werden in der Konfigurationsdatei „pom.xml“ gespeichert. Von einer detaillierten Beschreibung dieser Datei wird in diesem Artikel abgesehen. Das folgende Listing zeigt lediglich die ersten Zeilen dieser Datei, in der die ArtifactID definiert wird. Diese ID wird bei der Einbindung der Cartridge in das Generatorprojekt benötigt, siehe „In den Generator einbinden“.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>OiO AndroMDA Sample Cartridge</name>
  <artifactId>oio-andromda-sample-cartridge</artifactId>
  ...
```

Beispiel 2: pom.xml

Eine vollständige Vorlage für Ihre „pom.xml“-Datei finden Sie im Projekt des Anhangs.

NAMESPACE DEFINIEREN

Ein elementarer Baustein einer AndroMDA-Cartridge stellt der Namespace-Deskriptor dar. Er repräsentiert die Schnittstelle zwischen unserer Cartridge und des AndroMDA Kerns. Innerhalb dieses Deskriptors wird der Namensraum (Namespace) der Cartridge, in unserem Beispiel “OiOSample”, anhand dessen kann unsere Cartridge identifiziert und in das spätere Generatorprojekt eingebunden werden. Des weiteren enthält der Namespace-Deskriptor die Verweise auf die restlichen Deskriptoren, sowie eine Liste der angebotenen Konfigurationsparameter (Properties). Die Werte der Konfigurationsparameter werden im Generatorprojekt definiert und während des Generatorprozesses von den restlichen Deskriptoren und Templates ausgelesen. In unserem Fall benötigen wir ein Property „output”, welches das Ausgabeverzeichnis des generierten Codes darstellt.

Erstellen wir hierfür eine neue XML-Datei „namespace.xml“ im Verzeichnis „src/META_INF/andromda/“. In diesem Verzeichnis werden ebenso die restlichen Deskriptoren abgelegt.

```
<?xml version="1.0" encoding="UTF-8" ?>
<namespace name="OiOSample">
  <components>
    <component name="cartridge">
      <path>META-INF/andromda/cartridge.xml</path>
    </component>
    <component name="metafacades">
      <path>META-INF/andromda/metafacades.xml</path>
    </component>
    <component name="profile">
      <path>META-INF/andromda/profile.xml</path>
    </component>
  </components>
  <properties>
    <propertyGroup name="Outlets">
      <property name="output">
        <documentation>
          The location to which the files will be generated.
        </documentation>
      </property>
    </propertyGroup>
  </properties>
</namespace>
```

Beispiel 3: src/META_INF/andromda/namespace.xml

IN DEN GENERATOR EINBINDEN

Die projektspezifischen Informationen für den späteren Generatorvorgang befinden sich in der Konfigurationsdatei „andromda.xml“ Ihres Generator gestützten Projekts. Diese Datei enthält Informationen über das UML-Diagramm und Parametereinstellungen verwendeter Cartridges.

Damit unsere Cartridge in den Generatorprozeß integriert wird, müssen wir den Namespace und die Parameter innerhalb dieser Konfigurationsdatei definieren. Öffnen Sie hierzu die Konfigurationsdatei „andromda.xml“ Ihres AndroMDA-Projekts und fügen die folgenden Zeilen hinzu:

```
...
 <namespace name="OiOSample">
  <properties>
    <!-- Java specific settings -->
    <property name="languageMappingsUri">Java</property>
    <property
    name="wrapperMappingsUri">JavaWrapper</property>
     <!-- Outlets -->
    <property name="output">${test.output.dir}</property>
  </properties>
 </namespace>
...
```

Beispiel 4: <Projektxyz>/andromda.xml

Darüber hinaus muß die Cartridge in die Buildkonfiguration Ihres Generator gestützten Projekts integriert werden. In unserem Fall geschieht dies mittels eines neuen Eintrags in den „dependencies“ der Konfigurationsdatei „pom.xml“ von Maven unseres AndroMDA-Projekts.

```
<dependency>
   <groupId>org.andromda.cartridges</groupId>
   <artifactId>oio-andromda-sample-cartridge</artifactId>
   <version>3.2<version>
</dependency>
```

Beispiel 5: pom.xml (ihres Generator gestützten Projekts)

Es gilt zu Beachten, daß die Cartridge erst nach der ersten erfolgreichen Installation Ihrem Projekt zur Verfügung steht.

UML PROFIL

Wie wir bereits in den Vorüberlegungen definiert hatten, sind Sie sich bereits im Klaren über Ihre Zielarchitektur und die Spezifikation Ihres Metamodells. In unserem Beispiel besteht die Zielarchitektur aus mehreren Textdateien. Wobei jede Datei für ein Klassenelement mit dem Stereotyp „HelloWorld“ steht. Deren Inhalt besteht aus einem kleinen Text indem ein Eigenschaftswert (TaggedValue) des Stereotyps ausgelesen wird.

Wir müssen dementsprechend ein Profil erstellen, indem wir einen Stereotyp: „HelloWorld“ und ein TaggedValue: „helloworld.name“ definieren, siehe folgende Abbildung. Zur Hilfe steht uns eine aktuelle MagicDraw Version 12.5.

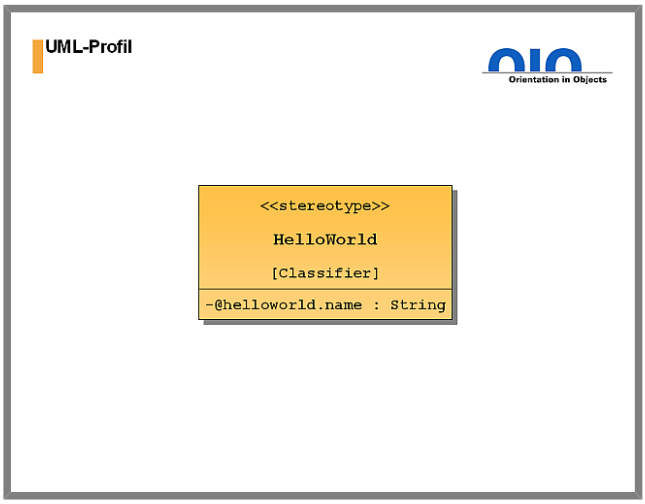

Abbildung 3: UML-Profil

PROFIL DESKRIPTOR

Entsprechend zu unserem UML-Profil müssen wir die Stereotypen und TaggedValues im Profile-Deskriptor der Cartridge beschreiben:

```
<?xml version="1.0" encoding="UTF-8" ?>
<profile>
 <elements>
   <!-- Stereotypes -->
   <elementGroup name="Stereotypes">
     <element name="ST_HELLOWORLD">
       <documentation>
         Says that an class is an Helloworld class.
       </documentation>
       <value>HelloWorld</value>
       <appliedOnElement>class</appliedOnElement>
     </element>
   </elementGroup>
   <!-- Tagged Values -->
   <elementGroup name="Tagged Values">
     <element name="TV_HELLOWORLD_NAME">
       <documentation>
         Defines the tagged value: name.
       </documentation>
       <value>@helloworld.name</value>
       <appliedOnElement>class</appliedOnElement>
     </element>
   </elementGroup>
 </elements>
</profile>
```

Beispiel 6: src/META_INF/andromda/profile.xml

Grundsätzlich gilt es zu Beachten, daß alle Stereotypen und TaggedValues im UML-Profil sowie im Profile-Deskriptor definiert werden.

METAFACADES

Der Entwickler einer Cartridge wird in der Regel die angebotenen Basis-Metafacades des AndroMDA-Kerns erweitern, indem er eine neue Metafacade definiert, welche von einer passenden Basis-Metafacade mittels Vererbung abgeleitet ist. Dadurch bietet AndroMDA dem Entwickler die Möglichkeit, sein Metamodell mit Hilfe von Metafacades auszudrücken. Dieses Konzept soll anhand der folgenden Abbildung verdeutlicht werden.

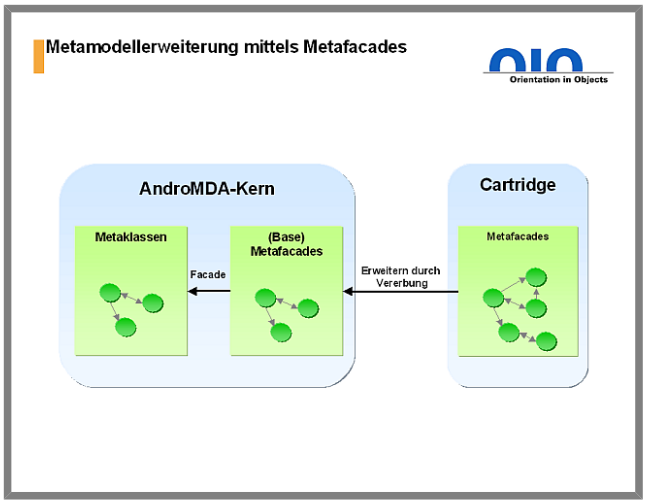

Abbildung 4: Metamodellerweiterung mittels Metafacades

In unserem Beispiel erzeugen wir die Metafacades mit Hilfe der angebotenen Meta-Cartridge von AndroMDA. In diesem Fall erfolgt die Definition der Fassaden über ein UML-Modell, wobei die Java-Repräsentation der Metafacades während des Buildvorgangs der Cartridge generiert wird.

1. METAFACADES IN MAGICDRAW 9.5 MODELLIEREN

Beginnen wir mit der Modellierung der Metafacades. In unserem Beispiel wird ein Metamodellelement namens „HelloWorld“ vom Typ „Class“ benötigt. Diesen Weg haben wir bereits durch die Definition des UML-Profils eingeschlagen. Dem entsprechend erstellen wir nun eine Klasse „HelloWorldFacade“ mit dem Stereotyp „Metafacade“ und einer Vererbungsbeziehung zur Klasse „ClassifierFacade“. Diese Beziehung legt den Typ der HelloWorld-Metafacade fest, und zwar den eines UML-Klassenelements. Ein HelloWorld Element wird dem entsprechend ein Klassenelement mit dem Stereotyp „HelloWorld“ sein.

Für das Auslesen des Eigenschaftswerts „HelloWorldName“ wird zusätzlich eine neue Operation namens „getHelloWorldName“ mit dem Rückgabetyp: „String“ modelliert.

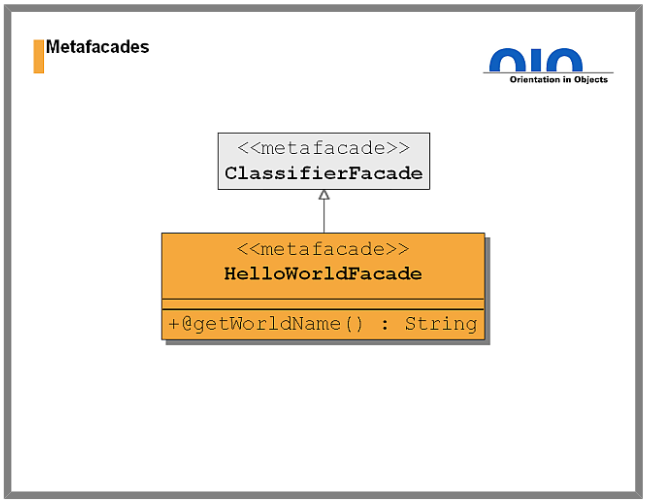

Abbildung 5: Metafacades

Hinweis: Das Modell mit den Metafacades wird in der Sprache UML 1.4 beschrieben. Die Bearbeitung ist nur mit MagicDraw 9.5 möglich. Als Vorlage können Sie die Cartridge des Anhangs verwenden.

2. GENERIEREN

Wie bereits angedeutet werden die Metafacades von AndroMDA generiert. Hierfür wird, wie bei allen AndroMDA-Projekten, eine entsprechende Konfigurationsdatei namens „andromda.xml“ benötigt. Sie enthält die Informationen zum Metamodell und den verwendeten Cartridges. In diesem Fall die Java- und Meta-Cartridge.

Von einem Listing und Beschreibung dieser Datei wird diesmal abgesehen. Der Inhalt entspricht den Konfigurationsdateien Ihrer AndroMDA-Projekte, was Ihnen bereits bekannt sein sollte. Sie finden eine Vorlage für Ihre Konfigurationsdatei im Projektverzeichnis: „src/resources/conf“ des Anhang.

Anschließend können Sie die Cartridge das erste mal installieren. Führen Sie den Befehl „mvn install“ im Verzeichnis Ihrer Cartridge aus. Bei diesem Vorgang werden die Javaklassen der modellierten Metafacades generiert, welche Sie nun in einem weiteren Schritt noch mit Leben füllen müssen.

3. FEHLENDE LOGIK IMPLEMENTIEREN

Die benötigte Logik der Operationen (in unserem Beispiel: „getHelloWorldName“) innerhalb der generierten Javaklassen fehlt noch. Der Generator erzeugt entsprechend der modellierten Klassenmethoden leere Methodenrümpfe, welche wir nun in einem weiteren Arbeitsschritt manuell implementieren werden. Öffnen Sie die Datei „HelloWorldFacadeLogicImpl.java“ in Ihrem Quellverzeichnis und implementieren die folgende Methode:

```
protected java.lang.String handleGetWorldName()
{
  String helloWorldName = null;
  if (this.hasStereotype(SampleProfile.STEREOTYPE_HELLOWORLD))
  {
    helloWorldName = (String) this.findTaggedValue(
       SampleProfile.TAGGEDVALUE_HELLOWORLD_NAME
    );
  }
  if (StringUtils.isEmpty(helloWorldName)) {
    helloWorldName = "Unidentifiable world";
  }
  return helloWorldName;
}
```

Beispiel 7: HelloWorldFacadeLogicImpl.java

Sie brauchen nicht befürchten, daß Ihre Änderungen überschrieben werden! Die generierten Metafacades bestehen immer aus 2 Klassen, welche über Vererbung miteinander verbunden sind (siehe folgende Abbildung). Die „Impl“-Metafacades werden lediglich generiert, sollten sie noch nicht im System bestehen.

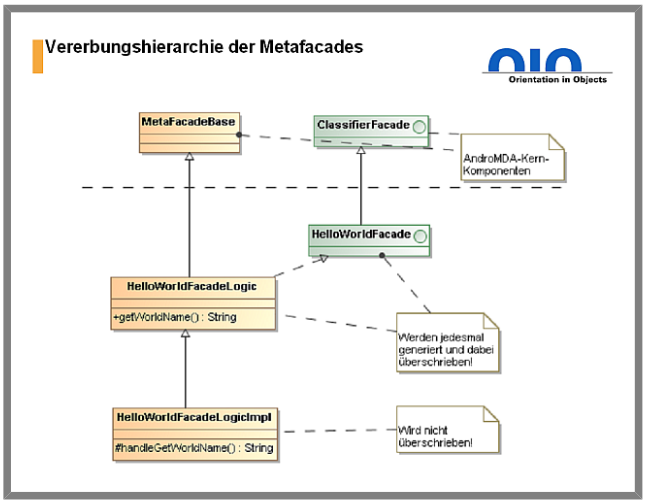

Abbildung 6: Verbungshierarchie der Metafacades

METAFACADE-DESKRIPTOR

Als nächstes beschreibe ich die Verbindung der Metafacades mit den Elementen des UML-Modells. Ohne diese Definition fehlt dem Generator die Information, welche Metafacades zu welchen UML-Elementen zugeordnet werden. Wir benötigen also einen weiteren Deskriptor „src/META_INF/andromda/metafacades.xml“ :

```
<?xml version="1.0" encoding="UTF-8"?>
<metafacades>
  <!-- Mapping all classes with stereotype HelloWorld
    to HelloWorldFacade -->
  <!-- Notice! Replace placeholder <Package> -->
  <metafacade
    contextRoot="true"
    class="<package>.HelloWorldFacadeLogicImpl" >
    <mapping
    class="org.omg.uml.foundation.core.Classifier$Impl">
      <stereotype>ST_HELLOWORLD</stereotype>
    </mapping>
  </metafacade>
</metafacades>
```

Beispiel 8: src/META_INF/andromda/metafacades.xml

Hierbei wird immer nach dem gleichen Muster vorgegangen. Ein neuer Eintrag enthält die Information um welches Metaobjekt es sich handelt und durch welche Metafacade das Metaobjekt abgebildet wird. Das Attribut „class” bestimmt die Klasse der Metafacade in Form des voll qualifizierten Klassennamens. Das Element „stereotype” enthält den Namen der Stereotyp-Variable, wodurch die betreffenden UML-Elemente definiert werden.

Das gezeigte Listing hat zur Folge, daß bei jedem Aufkommen eines Klassenelements mit dem Stereotyp „HelloWorld“ eine Objektinstanz der Metafacade „de.oio.cartridges.rcp.metafacades.HelloWorldFacadeLogicImpl” erzeugt wird. Die Instanzen der Metafacade stehen anschließend den Templates zur Auswertung zur Verfügung.

TEMPLATES

Die Model-Code-Transformation in AndroMDA wird mit Hilfe der Templates realisiert. Die Metafacades dienen den Templates als Input, welche zur Generierung des Codes verwendet werden (siehe folgende Abbildung).

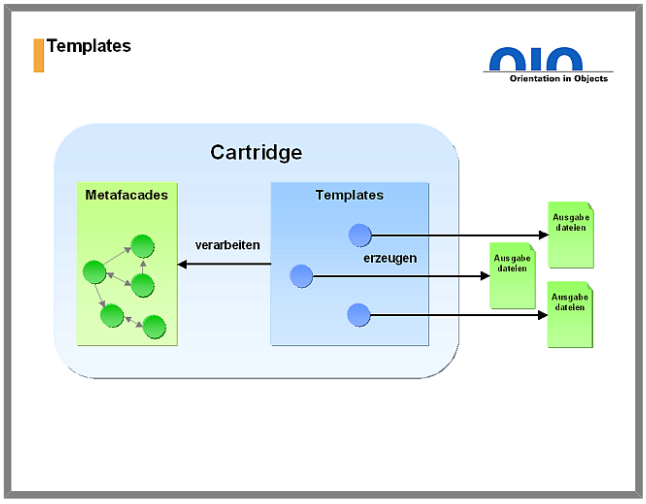

Abbildung 7: Templates

AndroMDA verwendet eine auf Java basierende Open-Source Template Engine namens Apache Velocity. Der Sprachumfang beschränkt sich im Wesentlichen auf eine geringe Menge verschiedener Anweisungen und Verweise, mit denen man Variablenoperationen, Schleifen, Verzweigungen und Zeichenkettenoperationen definieren kann. Eine ausführliche Beschreibung der Sprache befindet sich auf der Velocity Seite unter [13]. Zusätzlich besteht die Möglichkeit, Java-Methoden aus den Templates heraus aufzurufen, was uns erlaubt die Metafacades auszulesen.

Unsere Zielarchitektur sieht die Erzeugung einer Textdatei pro modellierten HelloWorld-Element vor. Die Implementierung eines einzelnen Templates wird exemplarisch anhand des folgenden Templates verdeutlicht. Erstellen Sie eine neue Velocity Datei „helloworld.vsl“ im Verzeichnis „src/templates/sample“:

```
Hello World. Here is planet "${helloWorldFacade.worldName}".
```

Beispiel 9: helloworld.vsl

In diesem Fall steht die Variable „$helloWorldFacade“ vom Typ „HelloWorldFacadeLogicImpl“ dem Template zur Verfügung. Über die Anweisung „$helloWorldFacade.helloworldname“ wird die Operation „getHelloworldName()“ aufgerufen. Der Rückgabewert in Form eines Strings wird an Stelle der Anweisung ausgegeben.

Anmerkend gilt es noch zu erwähnen, daß Velocity dem Entwickler eine Kurzschreibweise bei Methodenaufrufen ermöglicht. Der Methodenaufruf „${helloWorldFacade.getWorldName()}“ kann beispielsweise durch „${helloWorldFacade.worldName}“ verkürzt werden.

CARTRIDGE-DESKRIPTOR

Im letzten Schritt müssen wir die Metafacades mit den Templates verknüpfen. Die Beschreibung dieser Verknüpfung erfolgt innerhalb des Cartridge-Descriptors.

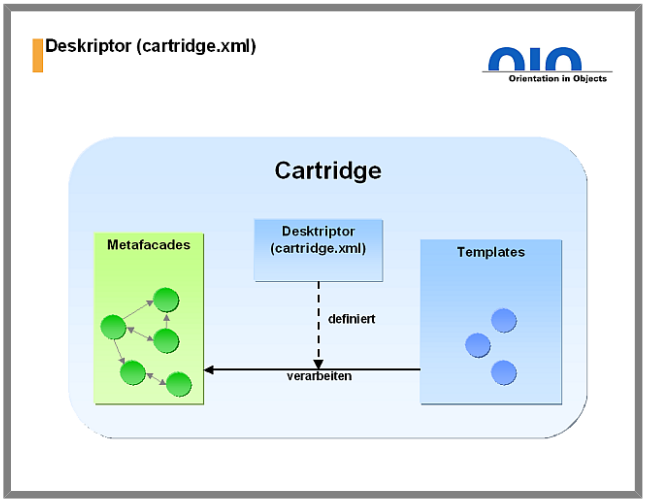

Abbildung 8: Deskriptor (cartridge.xml)

Wir erzeugen einen neuen Deskriptor „cartridge.xml“ mit folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<cartridge>
  <!-- Creates a textfile for each hello world element -->
  <!-- Notice! Replace placeholder <Package> -->
  <template
    path="templates/sample/HelloWorldElement.vsl"
    outputPattern="{0}/{1}.txt"
    outlet="output"
    overwrite="true"
    outputToSingleFile="false"
    outputOnEmptyElements="false">
    <modelElements variable="helloWorldFacade">
      <modelElement>
        <type
          name="<package>.HelloWorldFacade"
        />
      </modelElement>
    </modelElements>
  </template>
</cartridge>
```

Beispiel 10: src/META_INF/andromda/cartridge.xml

Auf Grund des Umfangs dieses Artikel werden lediglich die wichtigsten Attribute kurz erläutert.

Jedes „template" Element definiert die Ausgabe eines Templates. Das Attribut „path" gibt den Pfad dieses Templates an. Der Dateiname und das relative Verzeichnis der Ausgabedatei werden durch das Attribut „ouputPattern" definiert. Dabei wird die Pfadangabe mittels java.text.MessageFormat-Syntax [14] verarbeitet. Die beiden Platzhalter {0} und {1} entsprechen dem Paket- und Klassennamen des Modellelements und stehen zusätzlich dem Attribut zur Verfügung. Der absolute Ausgabepfad wird mit Hilfe des Attributs „outlet" definiert. In diesem Fall findet das Property „output", welches wir bereits im obigen Abschnitt des Namespace-Deskriptors definiert hatten, eine Verwendungsmöglichkeit.

Das Element „modelElement" beschreibt die Modellelemente, die dem Template in Form von Objektinstanzen der Metafacades zur Verfügung gestellt werden.

Besonders zu erwähnen gilt das Attribut „outputToSingleFile". Wird diesem Boolean-Wert der Wert „false" zugewiesen, so wird für jede Metafacade eine Ausgabedatei erzeugt. Dies entspricht unserer Erwartung. Wird jedoch dem Wert „outputToSingleFile = true" zugeordnet, so wird nur eine Datei erstellt und eine Liste aller passenden Modellelemente übergeben. In diesem Fall gilt zu beachten, daß die outputPattern-Muster: {0}, {1} nicht mehr verarbeitet werden können. Sofern für jedes UML-Element eine Ausgabedatei erzeugt werden soll, wird die erste Variante angewandt. Dieser Fall tritt in unserem Beispiel ein. Für jede modellierte UML-Klasse vom Typ „HelloWorld" wird eine entsprechende Datei generiert.

Mit der Implementierung dieses Deskriptors endet die Fertigstellung unserer Cartridge. Falls Sie es nicht bereits getan haben, installieren Sie nun die Cartridge, erstellen ein neues AndroMDA-Projekt, integrieren die Cartridge in das Projekt und laden das UML-Profil in Ihr Modell. Anschließend können Sie die ersten HelloWorld-Klassenelemente modellieren und generieren.

FAZIT

Abschließend kann gesagt werden, daß sich die Entwicklung einer neuen Cartridge ohne umfassende Kenntnisse im Umgang mit AndroMDA und den damit verbundenen Werkzeugen kompliziert und zeitaufwendig gestaltet. Demgegenüber fällt die Anpassung vorhandener Cartridges an projektspezifische Anforderungen einfacher aus. Dies ist jedoch nur möglich, sofern eine passende Cartridge zur Verfügung steht.

Die Grundidee der modellgetriebenen Softwareentwicklung sieht die Realisierung eigener Cartridges auf Basis einer Referenzimplementierung vor. Entsprechend hierzu empfehle ich bei größeren Softwareprojekten eigene Cartridges zu entwickeln, welche speziell auf die Bedürfnisse Ihres Projekts zugeschnittenen sind.

Dieser Artikel vermittelt Ihnen benötigtes Know-how und erleichtert den Einstieg in die Entwicklung einer AndroMDA-Cartridge. Als zusätzliches Bonbon gilt die in dieser Arbeit entwickelte Cartridge. Sie kann als Vorlage für Ihre eigene Cartridge dienen und steht selbstredend zum kostenlosen Download bereit, siehe Anhang.

ANHANG A: DIE FERTIGE CARTRIDGE

Download AndroMDA Cartridge.zip

REFERENZEN

- [1] AndroMDA Projekthomepage
  (http://www.andromda.org)
- [2] Beschreibung der AndroMDA Cartridges
  (http://galaxy.andromda.org/docs-3.1/andromda-cartridges/index.html)
- [3] Netbeans Metadata Repository (MDR) Projekthomepage
  (http://mdr.netbeans.org/)
- [4] Übersicht aller AndroMDA-UML-Tools
  (http://galaxy.andromda.org/docs/case-tools.html)
- [5] MagicDraw Projekthomepage
  (http://www.magicdraw.com/)
- [6] Apache Velocity Projekthomepage
  (http://jakarta.apache.org/velocity/)
- [7] Apache Maven Projekthomepage
  (http://maven.apache.org/)
- [8] AndroMDA Metafacades v3.2: Projektdokumentation
  (http://team.andromda.org/docs/andromda-metafacades/andromda-metafacades.pdf)
- [9] Object Management Group Homepage
  (http://www.omg.org/)
- [10] AndroMDA Anleitung zum Erstellen eines neuen AndroMDA Projekts
  (http://galaxy.andromda.org/index.php?option=com_content&task=category&sectionid=11&id=42&Itemid=89)
- [11] Java Sun
  (http://java.sun.com/)
- [12] Eclipse Projekthomepage
  (http://www.eclipse.org/)
- [13] Apache Velocity - User Guide
  (http://jakarta.apache.org/velocity/docs/user-guide.html)
- [14] MessageFormat API
  (http://java.sun.com/j2se/1.4.2/docs/api/java/text/MessageFormat.html)
- [15] AndroMDA 10-Schritte-Anleitung
  (http://galaxy.andromda.org/index.php?option=com_content&task=blogcategory&id=35&Itemid=77)
- [16] AndroMDA Developing your own cartridge
  (http://galaxy.andromda.org/docs/andromda-cartridges/developing.html)
- [17] A Bird's Eye view of AndroMDA
  (http://galaxy.andromda.org/docs-3.1/contrib/birds-eye-view.html)
- [18] Modellgetriebene Softwareentwicklung, dpunkt, ISBN: 3-89864-310-7
  Stahl, Thomas ; Völter, Markus
  2005