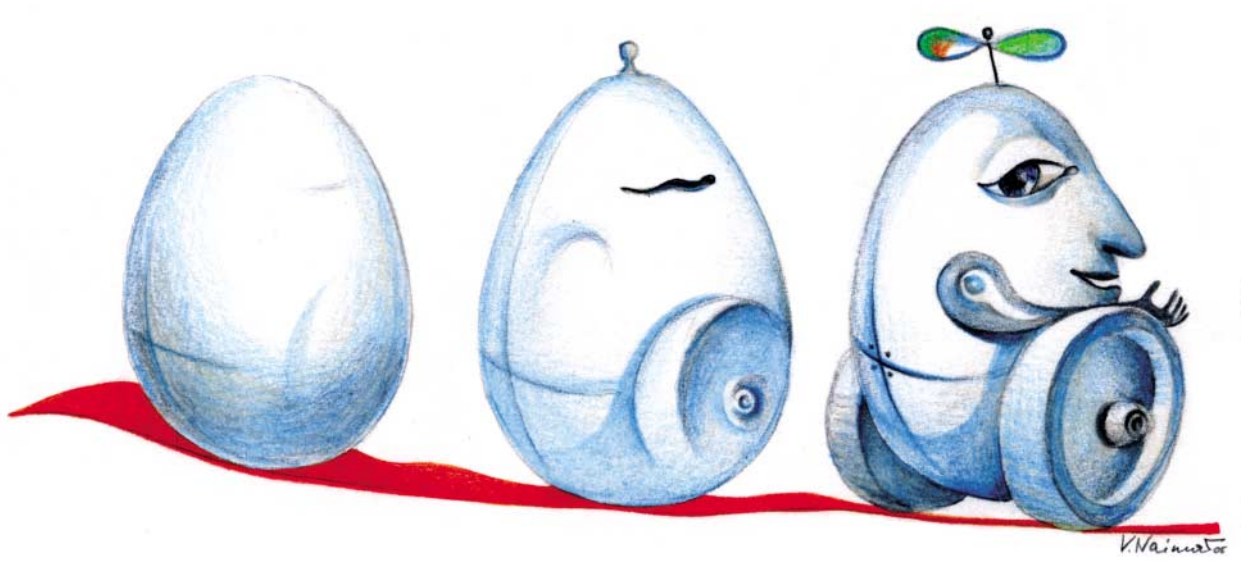
Codegenerierung mit dem openArchitectureWare Generator 3.0

The next Generation

VON KARSTEN THOMS UND BORIS HOLZER

Im Februar traf sich die Entwicklergemeinde des openArchitectureWare-Projekts [1], um die Entwicklungsarbeiten zu koordinieren und die nächsten Ziele abzustimmen. Als Ergebnis wurden nun der Generatorkern in der Version 3.0 freigegeben und andere Teilprojekte konsolidiert. In einer dreiteiligen Artikelserie stellen wir den aktuellen Stand vor und geben einen Ausblick auf die nahe Zukunft des Projekts.



openArchitectureWare (oAW) ist eine flexible Tool-Suite für modellgetriebene Softwareentwicklung (MDSD) und ist als Open-Source-Produkt frei verfügbar. Kern der Tool-Suite ist das Open Generator Framework (opengenfw). Um diesen Kern herum gibt es eine ganze Reihe von Teilprojekten, die nützliche Features für einen vereinfachten Umgang mit dem Generator bieten oder domänenspezifische Lösungen darstellen.

In den letzten Monaten sind viele Erweiterungen hinzugekommen, die für fortgeschrittene Anwender sehr nützlich sind. Es gibt aber auch viele Entwickler, die erst einmal den richtigen Einstieg suchen. Sven Efftinge hat mit seinem Beitrag [2] gezeigt, wie man mithilfe des Generators einen einfachen Webshop generieren kann. Das Projekt enthielt bereits Templates, um eine auf dem Struts- und Hibernate-Framework basierende Architektur abzubilden.

openArchitectureWare

Das Projekt openArchitectureWare ist hervorgegangen aus dem kommerziellen Produkt b+m Generator FrameWork der b+m Informatik AG (www.bmiag.de) und wurde 2003 unter der LGPL-Lizenz bei Sourceforge [8] als Open Source zur Verfügung gestellt. Kern des openArchitectureWare-Projekts ist das open Generator FrameWork (opengenfw). Zu den Kern-Features zählen:

- Unterstützung zahlreicher UML-Tools
- Jede Form von Modellinformationen ist auswertbar
- beliebige Ausgabeformate generierbar (Java, XML, JSP, PHP etc.)
- explizites Domain-Metamodell
- einfache Template-Sprache
- Ant-Integration
- Eclipse-Integration
- Unterstützung von aspektorientierten Ansätzen im Metamodell und in den Templates
- umfangreiche Dokumentation

Zahlreiche weitere Teilprojekte ergänzen die Plattform um Utilities, IDE-Integration, Tutorials, Beispielprojekte etc. Die wachsende Entwickler-Community sorgt für die ständige Weiterentwicklung und Verbesserung der openArchitectureWare-Plattform und bietet über das Forum auf der Sourceforge-Seite aktiven Support für Anwender. openArchitectureWare wird international in Unternehmen unterschiedlicher Branchen und Größen eingesetzt.

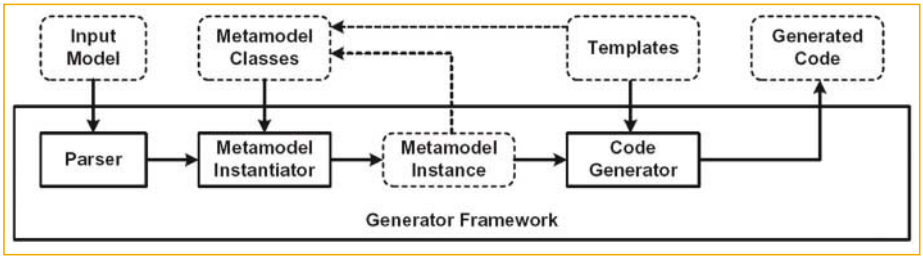


Abb. 1: Funktionsweise des Generator-Frameworks

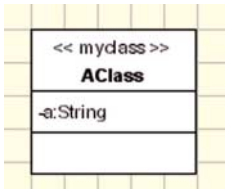


Abb. 2: Das Modell des Projekts

Wir wollen hier keine komplette Applikation generieren und konzentrieren uns auf die notwendigen Grundlagen, um aus einem einfachen UML-Diagramm Sourcecode zu erzeugen. In den zwei folgenden Artikeln werden wir auf fortgeschrittene Konzepte und Features von oAW eingehen.

Das GeneratorHelloWorld-Projekt

Als Beispiel verwenden wir das GeneratorHelloWorld-Projekt, welches bei oAW [1] als Beispielprojekt vorhanden ist. Das Projekt enthält:

- */model/model.zuml:* ein einfaches UML-Modell mit nur einer Klasse
- */templates/Root.tpl*: eine Generator-Template-Datei
- */src/mm/MyClass.java*: eine Metaklasse
- */doc/index.html*: Dokumentation
- */lib*: alle benötigten Bibliotheken

Als Erstes wird das Ant Build Script *build.xml* (ggf. muss die *project.properties* angepasst werden) mit dem Target *init* gestartet, um den Aufbau der Projektumgebung mit dem Anlegen von benötigten Verzeichnissen abzuschließen. Nun lassen wir das Build Script mit dem Default Target (*generate*) laufen. Der Generator wird dabei durch einen eigenen Ant-Task integriert und ausgeführt. Als Ergebnis wird im Ordner *src-gen* eine Java-Datei *AClass.java* generiert. Der Inhalt ist noch nicht spektakulär, lediglich der Rumpf der Klasse ist vorhanden:

```
public class AClass // MyClass
 {}
```

Listing 1

```
<<DEFINE Root FOR Model>>
 <<EXPAND Root FOREACH OwnedElement>>
<<ENDDEFINE>>

<<DEFINE Root FOR Class>>
 <<FILE Name(".java")>>
 public class <<Name>> {}
 <<ENDFILE>>
<<ENDDEFINE>>

<<DEFINE Root FOR MyClass>>
<<FILE Name(".java")>>
 public class <<Name>> // MyClass
 {
 }
<<ENDFILE>>
<<ENDDEFINE>>

<<DEFINE Root FOR ModelElement>>
<<ENDDEFINE>>
```

Funktionsweise des Generators

Wie kam nun das Ergebnis zustande? Während der Generierung (Abb. 1) parst das Generator Framework das UML-Modell (d.h. die Datei */model/model.zuml*) und erstellt mithilfe eines Instantiators eine Repräsentation davon im Speicher, die aus Instanzen von Metaklassen besteht. Für oAW gibt es bereits Instantiatoren für die meisten UML-Tools (siehe Kasten) und andere Modellrepräsentationen wie XML und Visio.

Der Code Generator nimmt die *Root.tpl*-Template-Datei (Listing 1) als Einstieg und expandiert für die im Modell enthaltene Model-Instanz die entsprechende Definition. Offenbar wurde für die modellierte Klasse (Abb. 2) nicht die im Template festgelegte Definition für *Class*, sondern für *MyClass* (*Class* hat nicht den Kommentar *// MyClass*) zur Codeerzeugung verwendet. Wir werden gleich sehen warum.

Das Metamodell

Das Generator-Framework enthält als Basis Metaklassen, die das UML-Metamodell abbilden. So werden z.B. im Modell enthaltene Klassen zu Instanzen der Metaklasse *Class*, Attribute zu *Attribute*, Assoziationen zu *Association* und *AssociationEnd* etc. Dies ist genau die Aufgabe, die ein Instantiator übernimmt.

Das vorhandene Metamodell wird in der Regel erweitert, um eine domänenspezifische Sprache (DSL) abzubilden. In unserem Beispiel wurde die Klasse mit dem Stereotyp *<<myclass>>* ausgezeichnet, *myclass* ist also Bestandteil unserer DSL. Dieser Stereotyp korrespondiert mit der Metaklasse *mm.MyClass* (im Verzeichnis *src/*), die einfach von der Metaklasse *Class* ableitet:

```
public class MyClass extends de.bmiag.genfw.meta.
                                    classifier.Class {
}
```

Durch eine Mapping-Datei (*metamappings.xml*) wird der Stereotyp *myclass* auf die Metaklasse *mm.MyClass* abgebildet:

```
<MetaMap>
 <Mapping>
  <Map>myclass</Map>
  <To>mm.MyClass</To>
 </Mapping>
</MetaMap>
```

Bei der Instanziierung des Modells werden nun alle mit *<<myclass>>* ausgezeichneten Klassen in *mm.MyClass* Objekte instanziiert, während „normale" Klassen auf *Class* abgebildet werden.

Unterstützte UML-Tools

- ARIS UML-Designer
- Artisan
- Gentleware Poseidon for UML
- Enterprise Architect
- Magic Draw
- Metamill
- MID Innovator Object
- Rational Rose + UniSys XML Tools
- Rational XDE
- Together

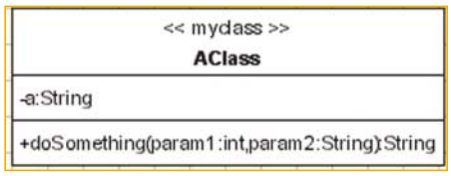


Abb. 3: Erweitertes Modell

Die Template-Sprache Xpand

Die Codegenerierungsvorschriften werden durch Templates beschrieben, welche Zugriff auf das instanziierte Modell haben. Die Templates werden in der oAW-eigenen Sprache Xpand beschrieben. Diese Sprache ist sehr einfach und enthält nur wenige Sprachelemente, die durch die Sonderzeichen << und >> ausgezeichnet werden. Die Templates bestehen aus Definitionsblöcken (*<<DEFINE>>*), die für bestimmte Elementtypen gelten (*Model*, *Class*, *MyClass*) und durch *<<EXPAND>>* auch einander aufrufen können. In den *DEFINE*-Blöcken können neben dem statisch auszugebenden Code Xpand-Steuerungsanweisungen wie Bedingungen (*<<IF>>*) und Schleifen (*<<FOREACH>>*) angegeben werden.

In Tabelle 1 sind die wichtigsten Sprachkonstrukte beschrieben. Die vollständige Syntax wird hier natürlich nicht aufgeführt, die wichtigsten Elemente sind aber in einer der möglichen Formen vorhanden. Die vollständige Xpand-Syntax kann in der Referenzdokumentation [3] nachgeschlagen werden.

Ihre Mächtigkeit erlangt Xpand vor allem in Kombination mit der Metamodellprogrammierung, auf die wir im nächsten Artikel näher eingehen. In der Tabelle sind Aufrufe auf dem Metamodell durch *<Invocation>* gekennzeichnet.

Der nächste Schritt

Die Modellklasse *AClass* besitzt noch ein Attribut mit dem Namen *a*. Es wird noch eine Methode *doSomething()* hinzumodelliert (Abb. 3). Daraus soll in der Klasse ein Property mit Getter und Setter sowie Signatur und Rumpf der Methode werden. Das Template *Root.tpl* wird dazu erweitert (Listing 2). Hier wird nun Gebrauch von der *<<FOREACH>>*-Schleife gemacht und die Definition einzelner Bereiche durch weitere *<<DEFINE>>*-Blöcke ausgelagert. Für die Generierung von Code für jedes Property wird die Definition *PropertyDef* expandiert und für jede Methode *MethodDef*. Wird der Generator mit dem neuen Modell und dem erweiterten Template aufgerufen, dann erzeugt dieser jetzt den in Listing 3 abgebildeten Code für die Klasse.

Namenskonventionen werden durch Aufruf von Methoden auf dem Metamodell realisiert, wie z.B. die Ausgabe eines Attributnamens als Parameter mit dem Präfix *p*. Aus dem Attribut *a* wird so durch die Angabe *<<Name.asPARAM>>* zu *pA*. Die Eigenschaft *Name* eines Modellelements ist im oAW-Metamodell vom Typ *Identifier*, die in diesem Beispiel spezialisiert wurde und eine Methode *String asPARAM()* definiert (Listing 4). Auf die gleiche Art erhalten wir die richtige Notation für die Namen der Getter- und Setter-Methoden.

Geschützte Bereiche

Die entstandene Klasse lässt sich allerdings noch nicht kompilieren. Die Methode hat den Rückgabewert *String*, der Rumpf der Methode ist mit leerem Inhalt generiert worden. Ein sinnvoller Inhalt ist aber häufig nicht (vollständig) generierbar. Was benötigt wird, ist eine Möglichkeit, in den generierten Code eigenen Code einzubringen, der bei einem erneuten Generatorlauf erhalten bleibt. Zu diesem Zweck gibt es so genannte geschützte Bereiche („Protected Region"), die durch einen *PROTECT*-Block gekennzeichnet werden. Wir ergänzen die Definition *MethodDef* wie folgt:

Sprachelement	Beschreibung
<<DEFINE <DefName> FOR <Identifier>>>... <<ENDDEFINE>>	Deklariert einen Definitionsblock mit der Kennung *<DefName>* für Modellelemente des Typs *<Identifier>*.
<<EXPAND <DefName> FOREACH <Invocation>>>	Expandiert die Definition mit der Kennung *<DefName>* für jedes Element, welches Ergebnis des Aufrufs *<Invocation>* ist.
<<FILE '<Filename>'>>...<<ENDFILE>>	Der Inhalt des Blocks wird in eine Datei mit Namen *<Filename>* geschrieben.
<<PROTECT CSTART '<CommentOn>' CEND '<CommentOff>' ID '<Id>' >>...<<ENDPROTECT>>	Zeichnet einen geschützten Bereich mit der eindeutigen Kennung *<Id>* aus. Der geschützte Bereich wird durch Codekommentare ausgezeichnet, die durch *<CommentOn>* und *<CommentOff>* umklammert werden.
<<IF <Cond>>>...<<ELSE IF <Cond>>>... <<ELSE>>...<<ENDIF>>	Bedingungen
<<FOREACH <Invocation> AS <VarName> EXPAND>>... <<ENDFOREACH>>	Expandiert für alle Elemente, die Ergebnis des Aufrufs *<Invocation>* sind, den enthaltenen Code. Das aktuelle Element der Iteration erhält den Namen *<VarName>*.
<<REM>>...<<ENDREM>>	Xpand-Kommentarblock

Tabelle 1: Kurzreferenz Xpand

Listing 2

```
...
<<DEFINE Root FOR MyClass>>
<<FILE Name".java">>
public class <<Name>>     // MyClass
{
  <<EXPAND PropertyDef FOREACH Attribute>>
  <<EXPAND MethodDef FOREACH Operation>>
}
<<ENDFILE>>
<<ENDDEFINE>>

<<DEFINE PropertyDef FOR Attribute>>
  private <<Type>> <<Name>>;
  public void <<Name.asSETTER>>
                          (<<Type>> <<Name.asPARAM>>) {
    <<Name>> = <<Name.asPARAM>>;
  }
  public <<Type>> <<Name.asGETTER>> () {
    return <<Name>>;
  }
<<ENDDEFINE>>

<<DEFINE MethodDef FOR Operation>>
  <<Visibility>> <<ReturnType>> <<Name>> <<NONL>>
  ("FOREACH Parameter AS curParam EXPAND USING
                                        SEPARATOR ", ""
    <<curParam.Type>> <<curParam.asPARAM>>
                              <<ENDFOREACH>>) { <<NL>>
  }
<<ENDDEFINE>>
...
```

```
<<DEFINE MethodDef FOR Operation>>
 ... { <<NL>>
 <<PROTECT CSTART "//" CEND " ID Id>>
 // place code here
 <<ENDPROTECT>>
 }
<<ENDDEFINE>>
```

Der Generator erzeugt nun innerhalb des Methodenrumpfes einen durch Kommentare gekennzeichneten Block:

```
public String doSomething (
 int pParam1,
 String pParam2) {
 //PROTECTED REGION
ID(I197eb84m10365753ca2mm4784) START
 // place code here
 //PROTECTED REGION END
}
```

An der mit *// place code here* gekennzeichneten Stelle können wir nun beliebigen Code einfügen, der bei jeder Generierung erhalten bleibt. Selbst eine Umbenennung der Methode im Modell ist möglich, da die Methode nicht durch ihren Namen gekennzeichnet ist, sondern durch einen vom UML-Tool vergebenen eindeutigen Identifier, der als ID der Protected Region im Code erscheint.

Schlussbemerkung

In diesem Artikel haben wir oAW so weit kennen gelernt, dass man mit dem oAW Generator loslegen kann. Das verwendete Beispiel ist trivial, ebenso der generierte Code. Reale Modelle und Templates sind natürlich deutlich komplexer. Die Grundprinzipien sollten aber durch das Beispiel klar geworden sein. Wir hoffen, dass wir mit diesem Artikel einen möglichst einfachen Zugang ermöglicht und Interesse für mehr geweckt haben.

Wer den Generator weiter ausprobieren möchte, kann das Modell verändern, indem z.B. Klassen, Attribute und Methoden ergänzt, umbenannt oder gelöscht werden und der Generator anschließend erneut ausgeführt wird. Auch mit den Templates sollte man etwas experimentieren und das Ergebnis untersuchen.

Es bietet sich zur Vertiefung unbedingt an, die umfangreich vorhandene Dokumentation zu nutzen, welche man über die oAW Homepage [1] herunterladen kann. Es befinden sich hier Dokumente von der Generator-Referenzdokumentation bis hin zu Artikeln, die sich mit dem Thema MDA/MDSD/MDD im Allgemeinen auseinander setzen [4] [5].

Im nächsten Artikel werden wir auf die Metamodellierung eingehen und demonstrieren, wie man Modelle, die einer spezifischen DSL beschrieben sind, mittels selbst entwickelter Metaklassen auswerten kann. Auf diese Weise entsteht auf Basis von oAW ein auf individuelle Anforderungen abgestimmter Generator. Neben der manuellen Erstellung der Metaklassen zeigen wir, wie der openArchitectureWare Generator genutzt werden kann, um diese Klassen ebenfalls zu generieren. Ein weiterer Schwerpunkt wird die Integration des Generators in Eclipse sein.

Die openArchitectureWare-Entwickler-Community möchte sich ausdrücklich beim Software & Support Verlag bedanken, deren Räumlichkeiten wir für das erste oAW Developer Community Come Together nutzen durften.

Karsten Thoms beschäftigt sich bei itemis schwerpunktmäßig mit modellgetriebenen Entwicklungsverfahren, der J2EE-Plattform und im Detail mit bekannten Java Open Source Frameworks. Er ist aktiv an der Entwicklung des openArchitectureWare-Projekts beteiligt.

Boris Holzers Schwerpunkte sind bei itemis die Konzeption J2EE-basierter Systeme, modellgetriebene Softwareentwicklung und das Coaching von Entwicklungsteams.

Listing 3

```
public class AClass          // MyClass
{
  private String a;
  public void setA (String pA) {
    a = pA;
  }
  public String getA () {
    return a;
  }
  public String doSomething (
    int pParam1,
    String pParam2) {
  }
}
```

Listing 4

```
public class Identifier extends
de.bmiag.genfw.meta.core.Identifier {
  ...
  public String asPARAM () {
    return "p"+capitalizeFirstLetter();
  }
  ...
  public String capitalizeFirstLetter () {
    if (Name==null || Name.length()==0) {
      return "";
    }
    String result = Name.substring(0,1).toUpperCase();
    if (Name.length()>1) {
      result += Name.substring(1);
    }
    return result;
  }
  ...
}
```

Links & Literatur

[1] openArchitectureWare Homepage: www.openarchitectureware.org

[2] Sven Efftinge: Shoppen in Kürze, in *Java Magazin* 1.2005

[3] b+m Informatik AG: Open Generator Framework, Referenz: www.openarchitectureware.org/data/openGeneratorFrameWorkReference_de.pdf

[4] Markus Völter: Metamodellbasierte Codegenerierung in Java: www.openarchitectureware.org/data/MetaModelBasedCodeGen.pdf

[5] Markus Völter, Thomas Stahl: Architektur und Generierung: www.openarchitectureware.org/data/ArchitekturUndGenerierung.pdf

[6] Dr. Marc Thomas, Karsten Thoms: Die MDA-Bank, in *Java Magazin* 11.2004

[7] Martin Grund, MDA-Tools: www.grundprinzip.de/files/MDA_Tools-Martin_Grund.pdf

[8] oAW-Projektseite: sourceforge.net/projects/architectureware