Model-to-Code-Transformation mit der Model to Text Transformation Language

Roland Petrasch

Beuth Hochschule für Technik, Fachbereich VI
Luxemburger Str. 10, 13353 Berlin
petrasch@beuth-hochschule.de

Abstract: Die Model-to-Text Transformation Language ist eine Sprache der Object Management Group zur Transformation von formalen Modellen zu Text bzw. Code. Dieser Beitrag stellt beispielhaft die Code-Generierung auf der Basis eines UML-Klassenmodells vor, wobei einige der Sprach-Features in Verbindung mit OCL zum Einsatz kommen. Auch einige Hinweise zur praktischen Anwendung seien gegeben.

1 Model-to-Text-Transformation im Rahmen der MDA

Mit einer Model-to-Text Transformation Language lassen sich Modelle auf der Basis von Templates zu textuellen Artefakten transformieren, wobei der häufigste Fall wohl die Code-Generierung darstellt. Gemäß der MDA muss für das Ausgangsmodell ein MOF-konformes [MOF06] Metamodell vorliegen, z.B. die UML Superstructure [UML10], während dies für die Zielsprache bzw. die Zielarchitektur nicht der Fall sein muss. Die Metamodellelemente des Ausgangsmodells finden im Rahmen der Transformations-Templates im Sinne von Platzhaltern Verwendung, wobei die Zielsprachenkonstrukte direkt in die Templates „programmiert“ werden.

Während für die Model-to-Model-Transformation (M2M) Metamodelle für das Ausgangsmodell (Platform Independant Model, PIM) und das Zielmodell (Platform Specific Model, PSM) vorliegen, ist nur ein Ausgangsmetamodell für die Model-to-Text-Transformation (M2T) notwendig. Das PSM aus der M2M-Transformation wird wieder als PIM für die M2T-Transformation bezeichnet [PM06].

Die Object Management Group veröffentlichte 2008 die Version 1.0 der „MOF Model to Text Transformation Language (MOFM2T)“ [M2T08]. Insofern stehen mit der OVT [QVT08] und der MOFM2T nun für beide Transformationsarten (M2M, M2T) entsprechende Standards der MDA zur Verfügung. Im Folgenden sei aus praktischer Sicht eine kurze Einführung in die Code-Generierung mit der MOFM2T gegeben.

2 Model-to-Text-Transformation: Ein Beispiel

Für das Beispiel wird ein mit Topcased [TOP10] erstelltes UML-Modell zu Java-Code transformiert. Dafür wird die MOFM2T-Implementierung von Acceleo in der Version 3.0 genutzt [ACC10]. Die Basis ist Eclipse 3.6 (Helios) in Form der Modeling-Distribution [ECL10]. Es soll eine einfache Java-Applikation generiert werden, die im sog. Applikationsprojekt untergebracht wird. Im Generatorprojekt befinden sich hingegen die Templates zur M2T-Transformation, d.h. zur Code-Generierung.

Das Klassenmodell (PIM) für das Beispiel enthält vier Klassen (s. Bild 1): Die Klasse `Shoppping-Cart` besteht aus Artikeln, wobei die Aggregation zur Klasse `Article` mit einer Assoziationsklasse `ShoppingItem` für die Warenkorbpositionen ausgestattet ist. Weiterhin gehört der Warenkorb zu genau einem Kunden (Klasse `Customer`).

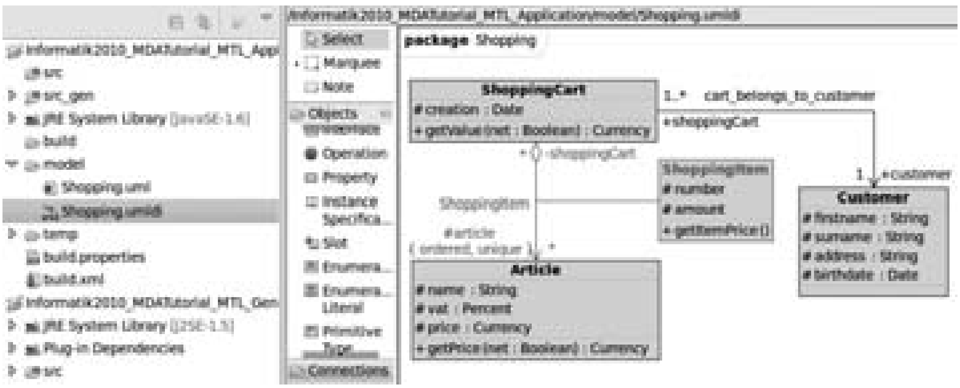

Bild 1: Applikationsprojekt mit UML-Klassenmodell als PIM

Im Generatorprojekt befinden sich die Acceleo Moduldateien mit den Templates. Bild 2 zeigt einen Teil des Templates für die Generierung von Java-Klassen, welches als Eingangsmodellelement eine UML-Klasse (Metaklasse `Class`) erwartet.

Bild 2: Generatorprojekt mit göffneter Template-Datei für Java-Klassen-Generierung

Ein Modul kann mehrere Templates enthalten. Tamplates in anderen Moduldateien lassen sich über Imports einbinden. Kommentare werden mit der comment-Direktive eingefügt, z.B. `[comment @main/]`, wobei die `@main`-Direktive den Einstiegspunkt beim Starten der Transformation markiert. Mit `[file …]` lässt sich eine Datei generieren, deren Inhalt durch nachfolgende Kommandos entsteht. Zunächst wird mit den folgenden Anweisungen die Klassendeklaration erzeugt:

```
[class.visibility/] class [class.name/] {
```

Gut erkennbar ist hier das Platzhalterprinzip des Templates: Überall dort, wo später bei der Code-Generierung Werte von Modellelementen des PIM eingesetzt werden sollen, stehen im Template die entsprechenden Metamodellelemente bzw. Ausdrücke, die diese Elemente verwenden, z.B. `[class.name/]`.Für die Behandlung der Properties sei zunächst auf das UML-Metamodell verwiesen [UML10, S. 48]: Dort sind die Metaklassen `Class` und `Property` durch eine Kompositionsbeziehung verbunden (s. Bild 3).

Bild 3: Ausschnitt aus dem Metamodell der UML 2.3: Metaklassen Class und Property (in Anlehnung an [UML10]

Im Template lassen sich die Attribute nun in Form einer Schleife behandeln. Das folgende Konstrukt zeigt den Schleifenkopf mit der Schleifenvariablen `prop` vom Typ `Property` (Metaklasse), die die Attribute (`ownedAttribute`) der Klasse `class` annimmt.

```
[for (prop:Property | class.ownedAttribute)]
```

Während einzelne Ausdrücke wie z.B. `[class.name/]` in eckigen Klammern stehen und durch das Zeichen „/" terminiert werden, bilden Konstrukte wie die Schleife einen Block, der mit einer Anweisung abgeschlossen werden muss, z.B. `[/for]`. Template-Aufrufe wie z.B. `[generateUMLProperty2Java(prop)/]`, können Parameter enthalten (`prop`). Die Definition des aufgerufenen Templates sieht wie folgt aus:

```
[template public generateUMLProperty2Java(prop : Property)]
  [if (not prop.visibility.oclIsUndefined())]
   [prop.visibility.toString()/]
  [else]
   protected [comment default visibility/]
```

```
  [/if]
  [getAsJavaType(prop.type)/] [prop.name/];
[/template]
```

Die If-Anweisung behandelt den Fall, dass die Visibility des Properties undefiniert ist: In diesem Fall wird einfach `protected` eingesetzt, was den Generator etwas toleranter in Bezug auf unvollständige Eingangsmodelle werden lässt. Die Bedingung mit der Funktion `oclIsUndefined()` zeigt die Verwendbarkeit von OCL [OCL10] im Rahmen der Templates.

Eine Query hilft dabei, wiederkehrende Aufgaben auszulagern. Im Folgenden seien zwei Queries mit identischem Namen vorgestellt, die für einen Klassennamen einen entsprechenden Java-Dateinamen liefern:

```
[query public getJavaFileName(string : String) : String =
  string.concat('.java')/]


[query public getJavaFileName(class : Class) : String =
  getJavaFileName(class.name)/]
```

Die Syntax einer Query ist leicht zu verstehen: Die Signatur enthält die typischen Elemente: Sichtbarkeit, Bezeichner, Parameterliste und Rückgabetyp. Der Body wird durch das Zeichen „=“ eingeleitet. Das „`/]`“ markiert das Ende des Query-Definitionsblockes. Die beiden Query-Signaturen unterscheiden sich nur durch den Parameter. Der Aufrufer profitiert vom polymorphen Verhalten: Durch den Parametertyp wird automatisch die richtige Implementierung ermittelt:

```
[file (getJavaFileName(class), false)]
[file (getJavaFileName('Customer'), false)]
```

Während beim ersten Aufruf `getJavaFileName (class : Class)` Verwendung findet, führt die zweite Zeile zum Aufruf von `getJavaFileName (string : String)`. Eine Variable lässt sich durch die Anweisung `[let …]` definieren. Im folgenden Beispiel ist dies die Sichtbarkeit einer Operation, wobei der Wert der Variablen `vis` davon abhängt, ob im Modell die Visibility überhaupt definiert wurde. Ist dies nicht der Fall, wird „`public`“ vereinbart.

```
[template public generateUMLOperation2JavaMethod(op :
Operation)]
[let vis : String = if (op.visibility.oclIsUndefined())
      then 'public' else op.visibility.toString() endif]
  [vis/] [getAsJavaType(op.type)/] [op.name/]
         ([ParameterList(op,',')/]) {
[/let]
// [protected ('operation body implementation')]
  // ToDo: implement operation
```

```
// [/protected]
}
[/template]
```

Das o.g. Beispiel zeigt auch die Definition einer protected Region, die durch einen Block mit `[protected …]` und `[/protected]` begrenzt wird. Im Code erscheint dann der entsprechende Hinweis und die Aufforderung, die Operation zu implementieren, als Kommentar.

Assoziationen zwischen Klassen führen dazu, dass bei der Umsetzung in Java in einer oder beiden beteiligten Klassen entsprechende Member entstehen, die vom Typ der jeweils anderen Klassen sind. Im Folgenden sei ein Ausschnitt aus dem Template `ge-nerateAssociation` gezeigt, das genau diese Aufgabe erfüllen soll: Es soll für eine Assoziation einer Klasse eine MemberVariable und eine Getter- sowie Setter-Methode generieren.

```
[template public generateAssociation(class : Class, assoc :
Association)]
[let assocEnd : Property = assoc.ownedEnd->select
     (type<>class)->asSequence()->first()]
  [assocEnd.type.name/] [assocEnd.name/];
  [GenerateGetterSetter(assocEnd, assocEnd.visibility)/]
[/let]
[/template]
```

Der Aufruf dieses Templates erfolgt dann im Rahmen einer Schleife, die über alle Assoziationen einer Klasse läuft. Dies könnte wie folgt aussehen:

```
[for (assoc:Association | class.getAssociations())]
  [generateAssociation(class, assoc)/]
[/for]
```

Um die Zuweisung für die Variable `assocEnd` zu verstehen ist erneut ein Blick auf das UML-Metamodell zu empfehlen: Die Assoziation ist über eine Meta-Assoziation zu einer oder mehreren Properties verbunden (`navigableOwnedEnd) (s. Bild 4).`

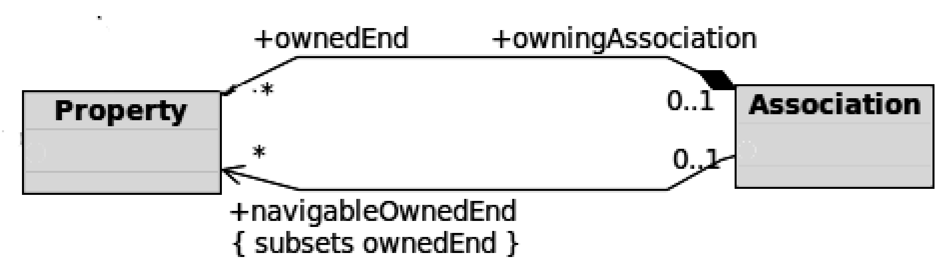

Bild 4: Ausschnitt aus dem Metamodell der UML 2.3: Metaklassen Property und Association (in Anlehnung an [UML10]

Kurz sei noch auf die Verwendung von OCL eingegangen. Wenn zu einer Klasse nur die konkreten Operationen ermittelt werden sollen, kann dies über eine Select-Funktion mit einer entsprechenden Bedingung erfolgen. Die Schleife für die Ausgabe der Namen aller konkreten Operationen einer Klasse `class` sieht wie folgt aus:

```
[for (op : Operation | class.ownedOperation->
      select(isAbstract=false))]
  [op.name/]
[/for]
```

3 Zusammenfassung

Die MOF Model-to-Text Transformation Language ist in Verbindung mit OCL ein mächtiges Werkzeug (nicht nur) zur Code-Generierung. Sie ist vom Ansatz her imperativ, typsicher und objektorientiert, daher unterstützt sie Techniken wie Polymorphie. Eine gute Erlernbarkeit ist auch durch die Syntax gegeben: Der überschaubare Befehlssatz umfasst die notwendigen Konstrukte und ist weitestgehend intuitiv verständlich.

Allerdings sollten die Templates im Sinne des Software-Engineerings mit den üblichen Qualitätsansprüchen konzipiert werden, um einen unwartbaren Wildwuchs zu verhindern. So gilt beispielsweise das Programming-by-Contract, d.h. bei jedem Template sollte klar sein, wie es zu verwenden ist, z.B. durch die Angabe der Pre-/Post-Conditions sowie der Invarianten. Auch sollte man auch hier die Entwicklungsprinzipien des S.O.L.I.D. Nicht vergessen, z.B. sollte ein Template oder eine Query im Sinne des Single Responsibility Principles nur genau für eine einzige Aufgabe verantwortlich sein.

Weiterhin besteht bei einigen Zielarchitekturen bzw. Zielsprachen die Gefahr, dass die Templates vom Umfang her unhandlich werden. Dem kann partiell durch die Beachtung der bereits genannten Techniken des Software-Engineerings entgegnet werden. Aber ab einer bestimmten Komplexität der Transformationen und Abstraktion der Ausgangsmodelle ist eine vorherige M2M-Transformation kaum zu vermeiden. Besonders wenn Designaspekte zu berücksichtigen sind, die manuelle Eingriffe erfordern. Derartige Entwurfsentscheidungen lassen sich praktisch kaum auf der Code-Ebene fällen – zumal im generierten Code bereits viele architekturorientierte Festlegungen enthalten sind. Frühzeitig sollte daher über M2M-Transformationen nachgedacht werden.

Die Diskussion um DSL (Domain Sepcific Languages) würde den Rahmen sprengen, es sei jedoch festgestellt, dass die UML als GPL (General Purpose Language) trotz ihrer Komplexität auf der Metamodellebene für viele Anwendungsgebiete geeignet erscheint. Was die Model-to-Text-Transformation mit MOFM2T angeht, ist die Verwendung einer eigenen MOF-konformen DSL jedoch genauso gut möglich.

Die hier verwendetet Acceleo-Implementierung in der Version 3.0 ist ausgereift und kommt als Eclipse-Plugin mit speziellem Editor mit Code Completion, Debug-Fähigkeit und Traceability daher. Für den Praxiseinsatz ist diese in Verbindung mit den anderen Eclipse-Projekten (EMF, UML etc.) gut geeignet. Der professionellen Anwendung von

MDA im Bereich Model-to-Text steht also nichts im Wege – vorausgesetzt das Personal ist ausreichend qualifiziert und eine Einführungsstrategie sowie ein passender Entwicklungsprozess für die modellgetriebene Software-Entwicklung sind vorhanden und akzeptiert.

Literaturverzeichnis

[ACC10] Obeo: Acceleo-Website, http://www.acceleo.org/pages/planet-acceleo/, Juli 2010

[ECL10] Eclipse Foundation: Eclipse Modeling Distribution (Helios Release), http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/heliosr, Juli 2010

[MDA03] Object Management Group: MDA Guide, Version 1.0.1, 2003

[MOF06] Object Management Group: MOF / Meta Object Facility Core, Version 2.0, 2006

[M2T08] Object Management Group: MOF Model to Text Transformation Language, Version 1.0, 2008

[OCL10] Object Management Group: Object Constraint Language, Version 2.2, 2010

[PM06] Petrasch R, Meimberg O: Model Driven Architecture. dpunkt Verlag, 2006

[QVT08] Object Management Group: MOF QVT - Version 1.0, 2008

[TOP10] Topcased Project, http://www.topcased.org/, Juli 2010

[UML10] Object Management Group: UML Superstructure and Infrastructure, Version 2.3, 2010