Kaskadierung von MDSD und Modelltransformationen

Markus Völter, voelter@acm.org, www.voelter.de

Die Grundlagen Modellgetriebene Softwareentwicklung (MDSD) sind ja zwischenzeitlich hinreichend oft beschrieben worden. Auch der Einsatz der grundlegenden MDSD Techniken ist heutzutage keine Seltenheit mehr. Zeit also, die etwas fortgeschrittenen Konzepte zu beleuchten. In diesem Artikel möchte ich auf zwei fortgeschrittene Themen eingehen: Zum einen die sogenannte kaskadierte MDSD und zum anderen Modelltransformationen. Beides ist in größeren MDSD Projekten essentiell, um die Komplexität des Ansatzes in Grenzen zu halten.

Aktueller Einsatz von MDSD

Modellgetriebene Softwareentwicklung wird heute in der überwiegenden Mehrzahl aller Projekte in einer Art „Minimalausbaustufe" eingesetzt. Dabei wird aus (mehr oder weniger) plattformunabhängigen Modellen direkt Code generiert. In aller Regel beschreiben die Modelle die grundlegenden Architekturkonzepte der Zielplattform mehr oder weniger direkt. Die Generierung von Code aus diesen Modellen ist daher auch relativ einfach – die semantische Lücke zwischen dem Modell und dem zu generierenden Code ist recht klein. Dieses Vorgehen nennen wir architekturzentrierte MDSD (siehe [SV05])

Dieser Ansatz ist mit den derzeitigen Tools sehr gut durchführbar. Trotz seiner relativen Einfachheit hat der Ansatz eine ganze Reihe von Vorteilen, darunter die Formalisierung (und damit die Präzisierung) der Architektur, die Automatisierung „nerviger" Detailimplementierung sowie als Folge davon die Steigerung der Softwarequalität, der Architekturkonformität und der Entwicklungsgeschwindigkeit.

Allerdings schöpft der Ansatz das Potential modellgetriebener Entwicklung nicht vollständig aus. Dies hauptsächlich deshalb, weil eben keine *fachlich domänenspezifischen* Abstraktionen bei der Modellierung verwendet werden, sondern „nur" die Architekturkonzepte. Damit kommt man zu einer breit nutzbaren, und damit gut wieder verwendbaren, aber wenig spezifischen MDSD-Infrasktruktur.

Kaskadierte MDSD

Um nun wirklich fachlich domänenspezifische Modelle verwenden zu können werden die Transformationen komplexer – die semantische Lücke zwischen Modell und zu generierendem Code wird größer. Um dieses Problem in den Griff zu bekommen schlägt die MDA mehrstufige Transformationen vor. Bei kaskadierter MDSD handelt es sich um eine pragmatische Umsetzung dieses Vorgehens.

Die Kernidee besteht zunächst mal darin, die architekturzentrierte MDSD wie sie heute praktiziert wird nicht als „Sparvariante“ zu verstehen, die man nur wegen unzureichender Tools verwendet. Statt dessen hat die Definition einer vernünftigen, technologienneutralen Architektur (siehe auch [MV05]), eines dafür passenden, formales Metamodells und die Implementierung einer generativen Infrastruktur für die Abbildung der Architektur auf eine Laufzeitplattform riesige Vorteile. Diese möchten wir beibehalten.

Nachdem also die Softwareentwicklung mittels dieser grundlegenden Infrastruktur im Projekt etabliert wurde, besteht der nächste Schritt nun darin, *darauf aufbauend* eine oder mehrere fachlich domänenspezifische Infrastrukturen aufzubauen. Es handelt sich also um eine Art Bottom-Up-Ansatz. Abbildung 1 zeigt das Prinzip.

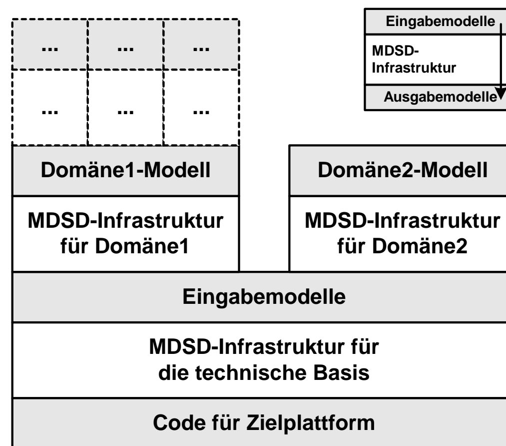

Abbildung 1: Prinzip kaskadierter MDSD

Bei kaskadierter MDSD erzeugen die Generatoren der nächst höheren Stufe als Ausgabe immer den Input für die zugrunde liegende – sowie oft Implementierungscode (ich zeige das im Beispiel unten). Komzeptionell handelt es sich bei dem Vorgehen also um Modell-zu-Modell-Transformationen. Der Tool-Support für Modell-zu-Modell-Transformationen ist derzeit bekanntermaßen noch nicht besonders gut (der OMG QVT Standard ist noch nicht verabschiedet). Natürlich kann man dieses Problem formal dadurch umgehen, dass man die konkrete Syntax des zugrunde liegenden Generators erzeugt (im Falle von UML wäre das XMI) also letztendlich doch wieder mit den üblichen Generatortool eine Modell-zu-Text Transformation durchführt.

Wir wollen hier allerdings einen eleganteren Weg zeigen, der auf der Transformation von Modellen innerhalb des Generators – also auf Objektgraph-Basis – beruht, und mit einigen der heute verfügbaren Werkzeugen gut umsetzbar ist.

Ein Beispiel

Um die abstrakten Erläuterungen oben etwas zu konkretisieren, hier nun ein real zum Einsatz gekommenes Beispiel. Es basiert es auf dem openArchitectureWare Generator [OAW].

Es handelt sich bei dem Beispiel um eine Enterprise-Anwendung, die laut initialer Planung auf Spring (serverseitig) und Eclipse (clientseitig) auszuführen war. Persistenz und Remote-Kommunikation sind zwei wichtige technische Aspekte. Zentrale architekturelle Abstraktionen (und damit Elemente des Metamodells) sind:

- Interfaces, die Verträge zwischen Dienstanbieter und Dienstkonsument definieren
- Komponenten die Interfaces implementieren bzw. andere Interfaces benötigen
- Datenstrukturen (Entitäten, Werttypen) sowie ihre Beziehungen
- Compositions, die Mengen von Komponentendeployments, sowie ihre Verdrahtung beschreiben
- Systeme und Systemknoten, die Compositions echter Netzwerkhardware zuordnen.

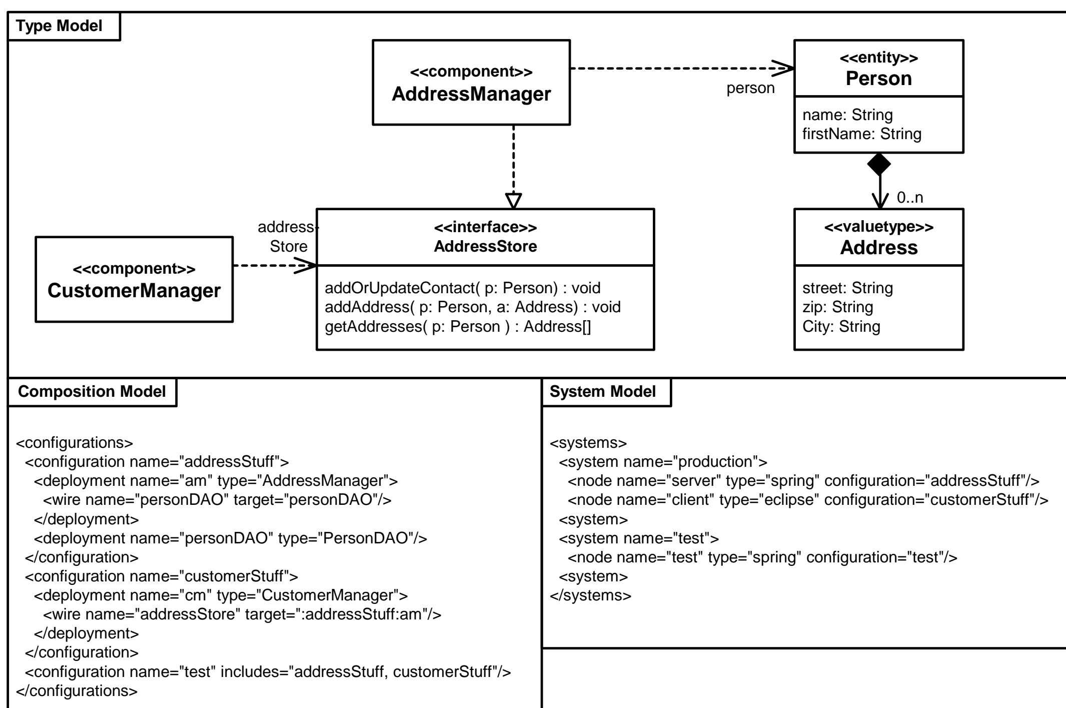

Abbildung 2: Beispielmodell

Abbildung 2 zeigt ein Beispielmodell. Im Rahmen des Typmodells wird ein Interface *AddressStore* definiert. Dieses wird von der Komponente *AddressManager* implementiert (also angeboten) und von der Komponente *CustomerManager* verwendet. Außerdem werden die benötigten Entitäten *Address* und *Person* definiert.

Im Rahmen des Composition-Models definieren wir nun zwei *Configurations* – eine Namens *addressStuff* und eine namens *customerStuff*. *addressStuff* enthält ein *Deployment* des *AddressManagers*, eines des *PersonDAOs* (weiter unten wird klar, wo diese Komponente herkommt – modelliert ist sie ja nicht!) sowie die betreffende Verdrahtung. *customerStuff* enthält ein Deployment von *CustomerManager* sowie die nötige Verdrahtung zur Adressverwaltung. Eine dritte *Configuration* ist inkludiert die beiden anderen zu Testzwecken.

Schlussendlich definieren wir nun zwei Systeme *production* und *test*. Im Falle von *test* lassen wir alle Deployments auf einem einzigen Spring-Knoten laufen. Im Falle des Produktionssystems verteilen wir die beiden Konfigurationen auf zwei verschiedene Knoten.

Was wird nun generiert

Basierend auf dem obigen Modell werden die folgenden Artefakte generiert:

- Java Interfaces
- JavaBeans für die Entitäten und Werttypen
- Komponenten-Basisklassen unter Verwendung von Springs *Dependency Injection* Idiom
- Die *beans.xml* für die Spring Konfiguration
- Remoting-Infrastruktur basierend auf Hessian
- Persistenz für die Datenstrukturen mittels Hibernate (also die *.hbm.xml* Files sowie die entspr. Einträge in der *beans.xml*)
- Ant Files, die ein JAR für jeden Knoten erstellen (bzw. ein eine Webapplikation (WAR), wenn man remote zugreifen will)
- Batch-Files um die Systeme zu starten (im Falle von Webapplikationen wir Tomcat gestartet)

Von Hand implementiert wird die reine Komponentenimplementierung (die von der generierten Basisklasse erbt) sowie das eine oder andere Property File (bspw. für die Datenbank-URLs, etc.). Abbildung 3 zeigt das Prinzip für Komponenten und Interfaces.

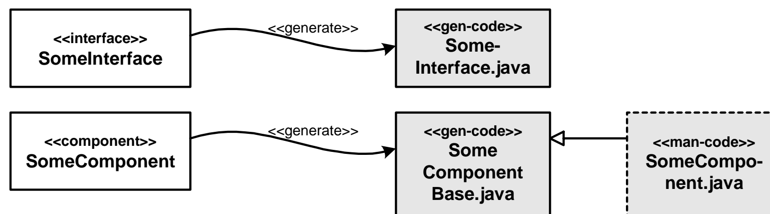

Abbildung 3: Code Generierung für Komponenten und Interfaces

Eine erste Modelltransformation

Im Falle der Entitäten ist etwas interessantes erkennbar: Obwohl wir keine Komponente *PersonDAO* im Typ-Modell definiert haben, haben wir sie trotzdem in der *Configuration addressStuff* deployt. Wie kann das sein?

Abbildung 4 zeigt das Prinzip. Der Generator verwendet eine einfache Modelltransformation, die die neuen Konzepte (hier: Entitäten) auf die vorhandenen architekturellen Bausteine (hier: Komponenten und Interfaces) abbildet. Es wird also für jede Entität im Modell Folgendes ausgeführt:

- Erstelle ein Interface namens <*Entity-Name*>*DAO* welches für die betreffende Entität die üblichen CRUD-Operationen (Create/Read/Update/Delete) definiert.
- Erstelle eine Komponente gleichen Namens, die dieses Interface anbietet und die Operationen sinnvoll implementiert.

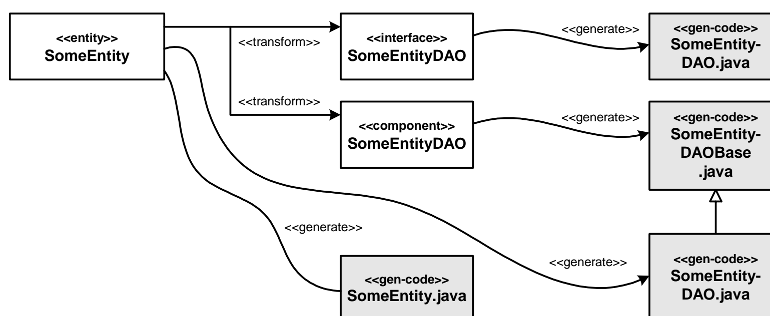

Abbildung 4: Die erste, einfache Modelltransformation

Man beachte, dass hier wirklich entsprechende Modellelemente erzeugt werden, nicht nur die Repräsentation in Code. Dies hat verschiedene Vorteile. Zum einen können die bereits vorhandenen Codegenerierungs-Schablonen für Komponenten und Interfaces unverändert weiter verwendet werden. Und zum anderen können auch weitere Mechanismen (wie eben obige Konfigurationen, die Systemdefinition sowie die dahinter liegenden

Korrektheits-Constraints) unverändert weiter verwendet werden. Der erstere Punkt spart Arbeit. Der zweite ist essentiell für eine funktionierende Systemdefinition.

Implementierung der Transformation

Modell-zu-Modell-Transformationssprachen sind derzeit immer noch ein Forschungs-/Experimental-Thema, die skalierbare, performante Anwendung im Team ist derzeit noch nicht möglich. Vollends problematisch wird die Sache, wenn man das Ergebnis der Transformation auch noch interaktiv weiterbearbeiten will[1]. Wenn man sich allerdings in zweierlei Hinsicht einschränkt, werden M2M-Transformationen praktikabel:

1. Die Transformation passiert „generator-intern", man kann das Ergebnis nicht interaktiv (im UML Tool oder sonst wo) weiterbearbeiten.
2. Als Transformationssprache verwendet man Java plus entsprechende Bibliotheken.

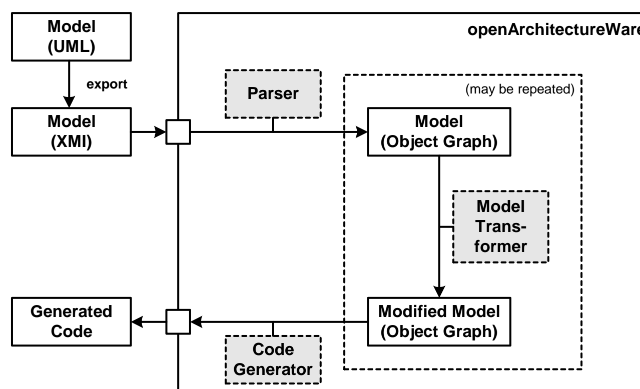

Abbildung 5: Modelltransformationen in openArchtiectureWare

Dieses Vorgehen kann mittels openArchitectureWare realisiert werden. Der Workflow inklusive der Transformationen ist in Abbildung 5 dargestellt. Im Rahmen eines Generatorlaufs können mehrere solche Transformationen durchgeführt werden.

Wie sieht nun die Transformation im Falle obigen Beispiels konkret aus? Wir müssen einen sogenannten *ModelModifier* implementieren (wer mehr über die Konzepte und die APIs von openArchitectureWare erfahren möchte, findet auf [OAW] ein entspr. Tutorial).

```
public class EntityMM implements ModelModifier {
```

[1] Mit den aktuell verfügbaren Tools ist eine derartige interaktive Synchronisation und Weiterbearbeitung nur schwer möglich, vor allem bei nicht-trivialen Modellen. Praktikabel wird dies derzeit vor allem dann, wenn man seinen eigenen, interaktiven Editor erstellt. Darin kann man dann bspw. Event-basiert die verschiedenen Modelle synchron halten. Siehe dazu [RV05]

Dessen *modifyModel()* Operation sucht sich alle Entitäten im Modell und ruft für jede die Operation *handleEntity()* auf.

```
public void modifyModel( MetaEnvironment me ) {
      ElementSet allEntities = EntityExtend.findAll( me );
      for ( … iterate over allEntities … ) {
            handleEntity( (Entity) it.next() );
      }
}
```

Dort legen wir zunächst ein neues Interface an, welches genauso heißt wie die Entität, nur mit dem Postfix „DAO“.

```
private void handleEntity( Entity e ) {
      Interface i =
            InterfaceExtent.newInstance( e, e.NameS()+"DAO" );
```

Dann fügen wir die Operationen zum speichern, löschen und updaten der betreffenden Entität ein.

```
      i.addOperation( createOperation( e, "save" ) );
      // add update and delete operation in the same way
```

Wir definieren dann eine neue Komponente, die das oben definierte Interface implementiert.

```
      Component c =
          ComponentExtent.newInstance( e, e.NameS()+"DAO" );
      c.addProvidedInterface( i );
}
```

Der Code zur Definition der Operationen sollten eigentlich selbsterklärend sein

```
private void createOperation( Entity e, String name ) {
      Operation o = OperationExtent.newInstance( e, name );
      Parameter p =
          ParameterExtent.newInstance( e, "entity" );
      p.setType( e );
      o.addParameter( p );
      return o;
}
}
```

Man sieht, durch die angenehme API der Metaklassen[2] (die *Extents*, die *add…()* Operationen, etc.) ist die Implementierung einer solchen Modelltransformation relativ effizient zu machen – dedizierte, deklarative Modelltransformationen wären zwar besser, bringen alles in allem aber nur noch einen marginalen Vorteil.

[2] die aus einem in UML beschriebenen, architekturspezifischen Metamodell generiert wird

Überprüfung der Transformation

Nachdem die Transformation „intern" im Generator passiert, das Ergebnis also während der Entwicklung nicht im UML Tool sichtbar wird, muss man einen anderen Weg finden das funktionieren der Transformation sicherzustellen. Dazu bietet openArchitectureWare in zweierlei Hinsicht Unterstützung. Zum einen kann man beliebige Modelle (wenn sie einmal zum Objektgraph geworden sind) mittels des in Eclipse integrierten *Model Structure* View anschauen. Sämtliche Modellelemente inkl. ihrer Beziehungen und Eigenschaften sind darin vorhanden - Abbildung 6 zeigt einen Screenshot.

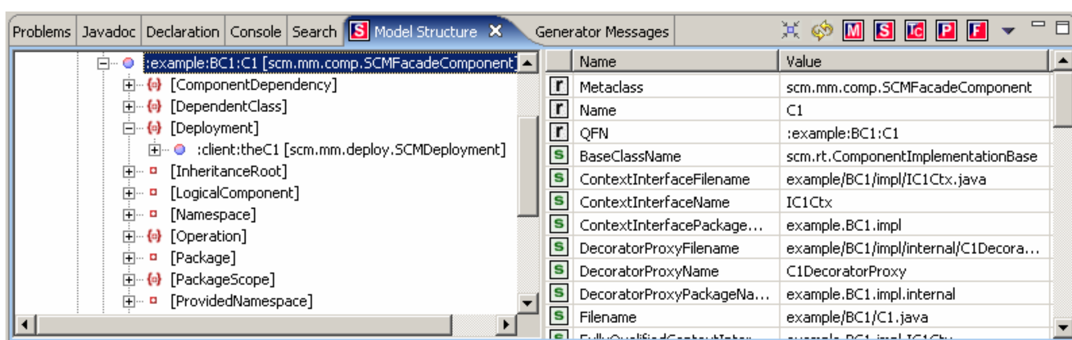

Abbildung 6: oAW Model Structure Browser

Diese interaktive Verifikation ist nützlich (gerade auch während der Entwicklung der Transformation) aber nicht ausreichend. Ein automatisierter Unit Test ist nötig. Auch hier kommt openArchitectureWare mit der nötigen Unterstützung in Form einer entsprechenden Erweiterung des JUnit Frameworks. Das folgende ist ein sog. *Generator Test Case* der die obige Transformation (bis zu einem bestimmten Grad) verifiziert.

Zunächst definieren wir im Testcase das Ant File und das Target welches die Generator-Konfiguration enthält und unter anderem angibt, welches UML Modell geladen werden soll.

```
public class EntityMMTest implements GeneratorTestCase {
  public String getAntFileName() {
     return "path/to/project/build.xml"; }
  public String getAntFileTarget() { return "validate"; }
```

Wir stellen dann sicher, dass die erwartete *Entity* tatsächlich vorhanden ist (wenn die schon fehlt, werden wir natürlich auch kein DAO etc. bekommen). Man beachte, dass wir hier die ganz normalen oAW-Utilties verwenden – Fehlermeldungen dieser Utility-Funktionen, die sonst einfach auf der Konsole landen werden hier automatisch zu einer JUnit *Failure* und lassen somit den Test fehlschlagen.

```
  public void testEntity() {
        MMUtil.findByName( getMetaEnv(), Entity.class, "Person",
              "cannot find Entity named Person" );
```

```
}
```

Im Folgenden testen wir nun das Vorhandensein (und die Details) des Interfaces und der Komponente.

```
public void testInterface() {
    Interface i = MMUtil.findByName( getMetaEnv(),
        Interface.class, "PersonDAO",
        "cannot find Interface named PersonDAO" );
    // verify operations of i
}

public void testComponent() {
    Interface i = MMUtil.findByName( getMetaEnv(),
        Component.class, "PersonDAO",
        "cannot find Component named PersonDAO" );
}

}
```

Spezifische Metamodellerweiterungen

Es ist oft nötig, im Falle solcher Modelltransformationen das zugrunde liegende Metamodell zu erweitern. Standardmäßig repräsentiert das Metamodell ja die Konzepte der Architektur – also deren zentrale Bausteine. Die Transformation legt Instanzen dieser Elemente an (siehe im Transformationsquellcode die *XYZExtent.newInstance()*-Aufrufe). Oft ist es notwendig, das Metamodell speziell zu erweitern. Ein gutes Beispiel wäre, dass sich die DAO Komponenten und das DAO Interface eine Referenz auf die Entität merken wollen, für die sie erstellt wurde, um das spätere Schreiben der Implementierungscode-Templates zu vereinfachen. Der folgende Code zeigt, wie man sich das im Rahmen der Transformation vorstellen würde (das Interessante sind die *setBaseEntity()* Aufrufe):

```
private void handleEntity( Entity e ) {
  DAOInterface i =
    DAOInterfaceExtent.newInstance( e, e.NameS()+"DAO" );
  i.setBaseEntity( e );
  // operations stuff as before
  DAOComponent c =
    DAOComponentExtent.newInstance( e, e.NameS()+"DAO" );
  c.setBaseEntity( e );
  c.addProvidedInterface( i );
}
```

Damit dies möglich wird, muss man das Metamodell entsprechend Abbildung 7 erweitern. Die neuen Untertypen *DAOComponent* und *DAOInterface* besitzen je eine Referenz auf die Entität für die sie „zuständig“ sind. Aufgrund des Polymorphismus sind *DAOComponents* ansonsten ganz normale Komponenten, *DAOInterfaces* normale *Interfaces*. Sie werden also insbesondere bei der Codegenerierung im Rahmen der Templates behandelt wie alle

anderen Komponenten und Interfaces – dazu muss an den Templates nichts modifiziert werden!

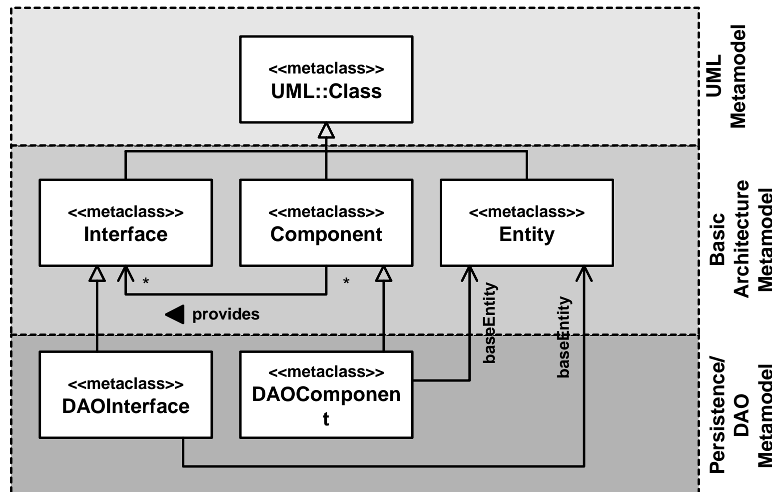

Abbildung 7: Metamodellerweiterung für DAOs

Möglicherweise sind auch noch weitere Constraints notwendig. Beispielsweise muss sichergestellt werden, dass eine *DAOComponent* nur genau ein Interface anbietet, und dies vom Typ *DAOInterface* ist:

```
public class DAOComponent extends Component {
  public String CheckConstraints() {
    Checks.assertElementCount( this, ProvidedInterface(), 1,
      "DAO components can only provide a single interface!" );
    Interface ii = (Interface)ProvidedInterface().get(0);
    Checks.assertType( this, ii, DAOInterface.class,
      "DAOComponents can only provide a DAO Interface" );
  }
  // more stuff
}
```

Aspektmodelle

Ein Kerngedanke des Ansatzes der kaskadierten MDSD ist auch, dass zwar die vorhandenen Templates weiter verwendet werden. Allerdings ist es notwendig, neue Templates zu schreiben, da ja die (sonst manuell implementierte) Anwendungslogik des Basisbausteins (hier: die Komponente) auch generiert werden soll. Im Beispiel bedeutet das, dass die Operationen in einer *DAOComponent* den entsprechenden Code enthalten sollen, der für die Implementierung der Persistenz notwendig ist. Da dieser Code auch generiert werden soll, müssen oft weitere Angaben im Modell gemacht werden (bspw. Query Statements, etc.). Nachdem man die *DAOComponents* aber während der

Entwicklung nicht „in die Finger bekommt“ stellt sich die Frage, wo man die betreffenden Informationen im Modell unterbringt.

Hier bietet sich eine Lösung an, die man (Buzzword-lastig ☺) als Aspektorientierte Modellierung bezeichnen könnte (siehe auch [MV04]). Für den Aspekt der Persistenz stellen wir ein eigenes Modell zur Verfügung, welches die für diesen Aspekt spezifischen Informationen enthält. Es ist Aufgabe des Generators, die beiden Modelle zu „verweben“ und insbesondere darauf zu achten, dass das Basismodell sowie das (oder die) Aspektmodell(e) konsistent zueinander sind. Das folgende ist ein einfaches Beispiel für ein Aspektmodell für unsere DAOs:

```
<queries>
  <entity name="Person">
   <query name="name" fields="name"/>
   <query name="nameAndFirstName" fields="name, firstName"/>
  </entity>
</queries>
```

Aufgabe des Generators wäre es nun zunächst, sicherzustellen, dass in dem Query-Aspektmodell nur Entities referenziert werden, die es im Basismodell tatsächlich auch gibt. Des weiteren muss der Generator bei der Definition der Operationen des DAO Interfaces dafür sorgen, dass es für jedes Query eine entsprechende Methode angelegt wird. Der Name der Methode entspr. dem Namen des Queries, die Parameter und deren Typen lassen sich aus dem *fields*-Attribut sowie den Attributtypen aus dem Entitätsmodell ableiten.

Natürlich muss im Rahmen der Implementierungscodegenerierung auch der betreffende Query Code generiert werden. Je besser das Persistenzframework, desto einfacher diese Aufgabe…

Ein „fachliches“ Beispiel

Bevor wir zum Fazit dieses Artikels kommen möchte ich noch kurz auf ein fachliches Beispiel eingehen… die Persistenz ist ja auch rein technischer Natur und sollte vor allem dazu dienen, das Prinzip zu erläutern. Im Folgenden wollen wir aufzeigen, wie Geschäftsprozesse sinnvoll integriert werden können. Modelliert werden Geschäftsprozesse folgendermaßen.

- Man definiert eine Prozesskomponente (Untermetaklasse von *Component*)
- Diese implementiert genau ein *TriggerInterface* (Untermetaklasse von *Interface)*
- Ein Zustandsdiagramm (mit Triggern, Guards und Actions) definiert den Prozess.

Abbildung 8 zeigt die Implementierung der Transformation. Aus der Zustandsmaschine werden zunächst für alle Trigger entsprechende Operationen in das Trigger-Interface eingebaut. Außerdem wird eine Entität angelegt, die – neben dem aktuellen Zustand – auch alle per Trigger-Event in die Maschine hereingereichten Parameter speichert (ggfs.

persistent – Entitäten können ja grundsätzlich persistent sein. Auch hier also die Wiederverwendung vorhandener Basisbausteine!).

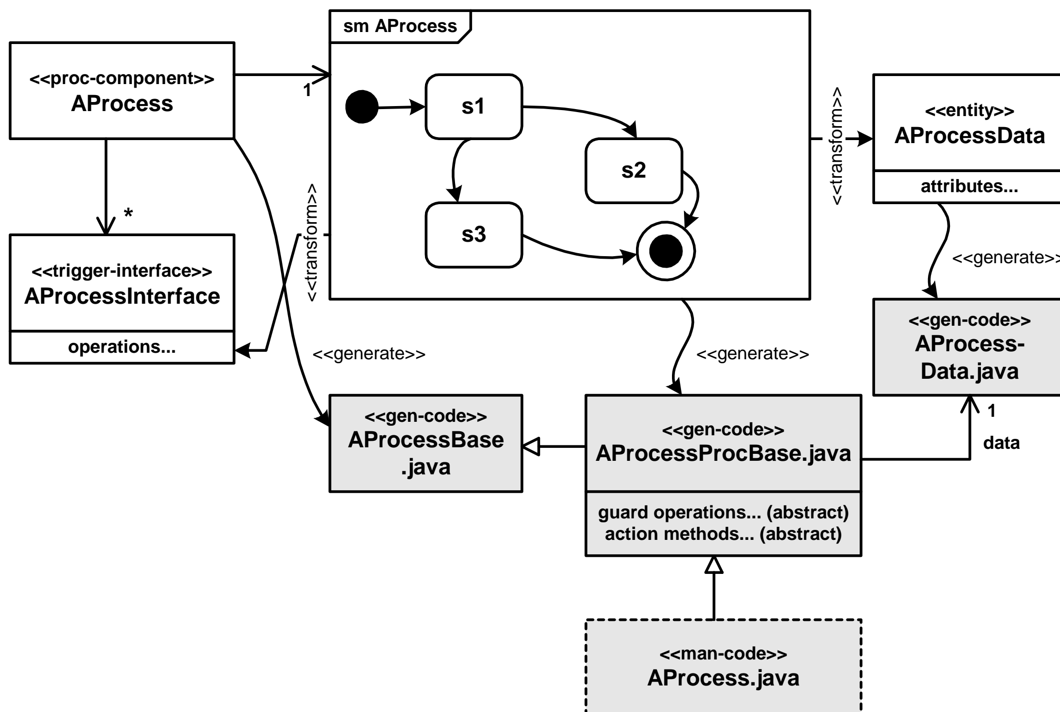

Abbildung 8: Transformationen für die Geschäftsprozesse

Generiert wird daraus dann insbes. die komplette Prozessimplementierung (unter Verwendung eines persistenten Prozesszustands, siehe *…Data* Entität), implementiert in der Klasse *…ProcBase*. Diese enthält die abstrakten Methoden für Actions und Guards. Von dieser Klasse muss der Entwickler nun erben und manuell die Action-Methoden sowie die Guard-Bedingungen implementieren. Er hat dabei Zugriff auf die Daten des Prozesses. Wenn eine andere Komponente den Prozess „weitertriggern" möchte, so muss sie ausschliesslich eine Referenz auf das Trigger-Interface modellieren (und dann in der Implementierung die betreffenden Methoden aufrufen). Gleiches gilt, wenn der Entwickler bei der Implementierung der Actions auf andere Komponenten zugreifen können möchte: dazu muss nur die Prozesskomponente eine Referenz auf das betreffende Interface modellieren. Das standardmäßige Programmiermodell bietet dann Zugriff auf die das betreffende Interface.

Fazit

Dieser Artikel will mehrere Aspekte „fortgeschrittener MDSD" darstellen. Zum einen ist es sinnvoll, fachlich motivierte DSLs auf einer architekturzentrierten Infrastruktur

aufzubauen. Man kann damit Generatorcode wieder verwenden und Komplexität reduzieren. Zweitens spielen Modelltransformationen – die mit heutigen Mitteln implementiert werden können! – dabei eine wichtige Rolle.

Viel Spaß beim Ausprobieren mit openArchitectureWare.

Referenzen

MV04 Markus Völter, Models and Aspects – Handling Cross-Cutting Concerns in the context of MDSD http://www.voelter.de/data/pub/ModelsAndAspects.pdf

MV05 Markus Völter, Architecture Patterns, http://www.voelter.de/data/pub/ArchitecturePatterns.pdf

OAW openArchitectureWare, http://www.openarchitectureware.org

RV05 Michael Rudorfer, Markus Völter, Domain-specific IDEs in embedded automotive software, EclipseCon 2005 und http://www.voelter.de/data/presentations/EclipseCon.pdf

SV05 Stahl, Völter, Modellgetriebene Softwareentwicklung – Technik, Engineering, Management, dPunkt 2005