MDSD und/oder AOSD?

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

Modellgetriebene Softwareentwicklung (MDSD) und Aspektorientierte Softwareentwicklung (AOSD) werden beide immer wichtiger im Rahmen der praktischen Softwareentwicklung – vor allem auch, weil die Werkzeugunterstützung immer besser wird. Ich mache im Rahmen meiner Arbeit allerdings immer wieder die Erfahrung, dass es vielen Entwicklern nicht klar ist, in welchem Verhältnis MDSD und AOSD stehen. Dabei gibt es zwei Aspekte: Zum einen stellt sich die Frage, wie die beiden prinzipiell (also mehr oder weniger theoretisch) zusammen passen. Andererseits stellt sich die vollkommen praktische Frage, ob und wie man beide Ansätze zusammen verwenden sollte. Ich möchte in diesem Artikel auf beide Themen kurz eingehen. Weitere Details finden sich unter [MV05].

MDSD

Von Modellgetriebener Softwareentwicklung (siehe [MV04b] für kurze Einführung, [SV05] für mehr Details) spricht man, wenn Software teilweise oder vollständig aus Modellen generiert wird. Dabei ist nicht, wie bei traditioneller Entwicklung üblich, die Software in einer 3GL Programmiersprache ausformuliert, sondern in Modellen spezifiziert. Solche Modelle müssen so exakt und so ausdrucksstark wie möglich die durch die Software zu erbringende Funktionalität beschreiben. Das ist nur dann möglich, wenn die Elemente eines solchen Modells mit Semantik hinterlegt sind, die eindeutig ein bestimmtes Verhalten zur Laufzeit oder eine bestimmte Struktur bestimmen. Eine Modellierungssprache, die in beliebigen Kontexten (Domänen) verwendet werden kann, sozusagen „general purpose" ist, wäre entweder unendlich groß, oder ihre Elemente so wenig abstrakt und problemspezifisch, dass sie einer herkömmlichen 3GL Sprache gliche, böte also auch nicht mehr Nutzen als so eine Sprache. Das Definieren einer Modellierungssprache rechnet sich nur dann, wenn Modellelemente den Problemraum prägnanter repräsentieren können als 3GL-Programmiersprachen, und das ist dann möglich wenn sie für eine spezielle Domäne entwickelt wird. Solche Modellierungssprachen nennt man Domänenspezifische Sprache oder DSLs (Domain Specific Languages).

Um dann letztendlich Software zu erhalten, die tatsächlich ausgeführt werden kann, müssen die Modelle durch Transformationen bzw. Codegenerierung in ausführbaren Code überführt werden.

AOSD

Bei der Aspektorientierung [AOSD] geht es darum, dass man Cross-Cutting Concerns (CCC) in Softwaresystemen zu modularisieren versucht. CCCs sind alle die Features eines

Systems, die sich (mit den Mitteln der verwendeten Programmiersprache oder des verwendeten Systems) nicht in einem Modul (Prozedur, Methode, Klasse, …) lokalisieren lassen. Die Aspektorientierung bietet verschiedene Möglichkeiten, solche CCC trotzdem modular zu beschreiben. Ein inzwischen recht bekannter Ansatz ist die Aspektorientierte Programmierung. Dabei werden neue Programmiersprachen-Konstrukte zur Verfügung gestellt, um Aspekte (so nennt man solche modularisierten CCCs dann) zu beschreiben, sowie Werkzeuge um diese Aspekte dann mit dem Basisprgramm (oder –system) zu verweben.

Gemeinsamkeiten von MDSD und AOSD

In diesem Abschnitt möchte ich zunächst einige Gemeinsamkeiten zwischen AOSD und MDSD erläutern.

Separation of Concerns. Beide Ansätze können zur Trennung von Belangen verwendet werden. AOSD macht das bekanntermaßen dadurch, dass querschnittliche Belange in Aspekte ausgelagert und später mit dem Basisprogramm verwoben werden. MDSD erreicht dieses Ziel indem abstrakte domänenspezifische Belange in einem Modell beschrieben werden, und die darauf folgenden Transformationen bestimmte (meist technische) Belange beisteuern.

Mechanik. Technisch arbeiten beide Ansätze mit Abfragen und Transformationen. AOSD verwendet eine Pointcut-Spezifikation, um relevante Punkte im Ablauf eines Programms zu identifizieren, an denen dann zusätzliches Verhalten (Advice) eingehängt wird. Im Falle von MDSD selektiert eine Modelltransformation einen Ausschnitt des Modelles und transformiert diesen Ausschnitt in ein anderes Modell, oder Code.

Metamodelle. In beiden Ansätzen spielen Metamodelle eine große Rolle. In MDSD ist dies offensichtlich, da jedes Modell letztendlich eine Instanz eines Metamodells darstellt. Modelltransformationen (oder Codegenerierung) werden bzgl. der Metamodelle definiert. Bei AOSD ist das Metamodell vielleicht nicht so offensichtlich; allerdings ist auch das Joinpoint-Modell eines AO Systems ein Metamodell. Ein laufendes Programm ist eine Instanz dieses Metamodells.

Unterschiede

Dynamisch vs. Statisch. MDSD transformiert statische Modelle (diese können Verhalten ausdrücken, aber das Modell selbst ist statisch). Die Transformation findet vor Ausführung des Systems statt. AOSD steuert Verhalten zum Ablauf eines Programmes bei. Man kann daher in vielen AO-System auch dynamische Dinge (z.B. den aktuelle Callstack) bei der Definition eines Pointcuts verwenden.

Einfluss auf die Entwicklung. Bei AOSD ist es in vielen Systemen möglich, Aspekte zum System beizusteuern *nachdem* das Basisprogramm vollständig implementiert wurde. Ein klarer Vorteil von AOSD, denn bei MDSD ist es offensichtlich so, dass man ein System mit

MDSD-Mittel *entwickeln* muss. Hat man es erst einmal ohne MDSD erstellt, hilft MDSD auch nicht mehr weiter.

Abstraktionslevel. AOSD spielt sich in aller Regel auf demselben (fachlichen) Abstraktionslevel ab wie das Basisprogramm. Domänenspezifische Ausdrucksmöglichkeiten sind nicht möglich. Im Falle von MDSD besteht die Idee ja gerade darin, Ausdrucksmittel (Sprachen, DSLs) für eine bestimmte Domäne zu entwickeln, um die "Programme" näher and die Problemdomäne anlehnen zu können, und damit die Ausdrucksfähigkeit für die betreffende Domäne zu steigern.

Nicht-Programmiersprachen-Artefakte. MDSD kann im Rahmen von Modell-zu-Code Transformationen natürlich beliebigen textuellen Code erzeugen – neben echtem Quellcode auch Konfigurationsfiles, XML oder Build-Skripte. Da AOSD ein laufendes System beeinflusst, können diese Artefakte bei AOSD nicht berücksichtigt werden.

Wie verwende ich MDSD und AOSD gemeinsam?

Aus Sicht des "praktizierenden Entwicklers" sind die Erläuterungen oben sicherlich hilfreich, die eigentliche Frage ist allerdings: Sollte ich mich für MDSD *oder* AOSD entscheiden, oder kann ich beides sinnvoll zusammen verwenden? Ich beschreibe im Folgenden sechs verschiedene Ansätze, um querschnittliche Belange im Kontext modellgetriebener Entwicklung zu adressieren; ich erörtere also die Frage, "Angenommen, ich mache MDSD, wie kann ich sinnvoll querschnittliche Belange adressieren?". Natürlich könnte man das Thema auch von der anderen Seite her aufziehen und AOSD in den Vordergrund stellen. Aufgrund meines Backgrounds und meiner Erfahrung habe ich mich für den MDSD-basierten Ansatz entschieden – das soll aber keine Wertung darstellen!

Ich werde mich bei der Betrachtung von MDSD übrigens auf (template-basierte) Codegenerierung konzentrieren, da Modell-zu-Modelltransformationen in der Praxis (noch) eine untergeordnete Rolle spielen. Außerdem kann ich aus Platzgründen nur sehr begrenzt auf die Konsequenzen bzgl. Performance, Komplexität, AO-Granularität eingehen. Auch hier verweise ich auf [MV05], da habe ich 25 Seiten mehr Platz, als in diesem Artikel ☺.

Template-Inhärente AOP

Bei der Verwendung von Codegenerierungstemplates bekommt man "ein bißchen AOP" ganz automatisch. Aus einer Template werden ja üblicherweise verschiedene konkrete Artefakte generiert. Ein einfaches *if* in der Template steuert also den Code in vielen Artefakten. Der Code in der folgenden Template fügt bei Operationen Sicherheitschecks ein, wenn dies nötig ist. Der CCC "Sicherheitschecks" ist also an einer Stelle – in der Template eben – lokalisiert.

```
«DEFINE OperationDef FOR Operation»
  public final «ReturnType» «Name» ( /* formal params */ ) {
    «IF checksRequired»
```

```
        if ( !Security.check("«Class.Name»", "«Name»") )
          throw new SecurityEx();
      «ENDIF »
      return «Name»Internal( /* actual params */ );
    }
  «ENDDEFINE»
```

AO Templates

Die Verwendung von Template-Inhärenter AOP wird, wenn man damit viele CCC behandeln will, irgendwann recht unübersichtlich, weil es eben viele solche *if*s in den Templates gibt. Um dies zu vermeiden, kann man nun AO-Weaving auf Template Ebene verwenden. Dies bedeutet, dass man mittels eines geeigneten Generators zusätzliche Templates an vorhandene "adviced". Dazu kann man beispielsweise in den regulären Templates einen Hook definieren, an den dann "von außen" weitere Templates angehängt werden können. Der folgende Quellcode (erstellt mit openArchitectureWare [OAW]) zeigt die Definition von zwei Hooks im obigen Beispiel.

```
«DEFINE OperationDef FOR Operation»
  public final «ReturnType» «Name» ( … as before … ) {
    «EXPAND HookMethodBegin»
    «ReturnType» res = «Name»Internal( … as before … );
    «EXPAND HookMethodEnd»
    return res;
  }
«ENDDEFINE»
```

Der folgende Code zeigt nun, wie man an diese Hooks zusätzliche Funktionalität – hier Logging – anbaut.

```
«DEFINE LoggingMethodBegin FOR Operation AT HookMethodBegin»
    «IF loggingRequired»
      // entering method such and such
    «ENDIF »
«ENDDEFINE»

«DEFINE LoggingMethodEnd FOR Operation AT HookMethodEnd»
    «IF loggingRequired»
      // leaving method such and such
    «ENDIF »
«ENDDEFINE»
```

AO Plattformen

Hier geht es nun darum, das Handling der CCC an die Plattform zu delegieren. Im Rahmen von MDSD generiert man ja nicht das gesamte System. Statt dessen implementiert man eine Basisplattform, auf die sich der generierte Code abstützen kann. Wenn diese Basisplattform bereits CCC adressiert, so kann man sich dieses zu Nutze machen. Üblicherweise kann eine solche Plattform – EJB ist ein Beispiel – nicht beliebige CCC behandeln, aber oft eben genau die, die in der betreffenden Domäne relevant sind. Aus

MDSD-Sicht wird dann nur noch eine Konfigurationsdatei generiert (bspw. EJB Deployment Deskriptoren). Das eigentliche Adressieren der CCC erledigt die Plattform.

Pattern-Basierte AOP

In Fällen wo die Plattform keine entsprechenden Features bietet, kann man trotzdem – bis zu einer gewissen Granularität – ähnliche Verfahren einsetzen. Ich bezeichne diesen Ansatz als Patter-basierte AOP, weil verschiedene bekannte Design Patterns kombiniert werden, um den gewünschten Effekt zu erreichen. Abbildung 1 zeigt das Prinzip.

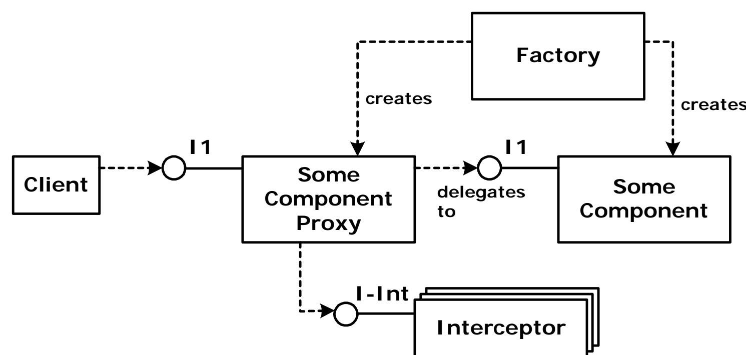


Abbildung 1: Pattern-basierte AOP

Die Idee ist, dass man auf Schnittstellenebene Proxies [GOF] einführt, deren Aufgabe es ist, an Interceptoren [POSA2] zu delegieren, die CCC adressieren. Eine Factory ist dafür verantwortlich, an den entsprechenden Stellen die Proxies einzuhängen. Ich habe diesen Ansatz im Zusammenhang mit EJB bereits in einem früheren Artikel beschrieben (siehe [MV04a]).

Pointcut Generierung

Falls alle oben genannten Verfahren nicht greifen, kann man darauf zurückkommen, eine AOP-Sprache in den Prozess einzubinden. Dafür gibt es natürlich die verschiedensten Möglichkeiten. Eine besonders nützliche ist es, den Advice von Hand zu implementieren, und dann, basierend auf Informationen aus dem Modell den Pointcut zu generieren. Unter Verwendung von AspectJ erreicht man dies beispielsweise durch abstrakte Aspekte. Das folgende Stück Code zeigt einen abstrakten Aspekt (also einen ohne Pointcut) der Tracing-Funktionalität beisteuert.

```
package aspects;
public abstract aspect TracingAspect {
  abstract pointcut relevantOperationExecution();
  before(): relevantOperationExecution() {
    // use some more sophisticated logging,
    // in practice
    System.out.println( System.currentTimeMillis()+"::"+
```

```
                               thisJointPoint.toString() );
    }
  }
```

Um diesen Aspekt nun mit dem generierten Code zu verweben, muss ein konkreter Aspekt abgeleitet werden, der dann den Pointcut beisteuert. Das Ergebnis könnte folgendermaßen aussehen:

```
package aspects;
public aspect SensorsOutsideTrace extends TracingAspect {
  pointcut relevantOperationExecution() :
    execution( * manual.comp.temperatureSensor..*.*(..) ) ||
    execution( * manual.comp.humiditySensor..*.*(..) );
}
```

Der oben gezeigte Code ist einem Beispiel entnommen, in dem ein System aus verteilten Komponenten erstellt wird; dieses besteht unter anderem aus einem Systemknoten SensorsOutside und zwei darauf laufenden Komponenten (ein Temperatursensor und ein Feuchtigkeitssensor). Das folgende Stück XML könnte z.B. dazu dienen, das Tracing für diesen Knoten einzuschalten.

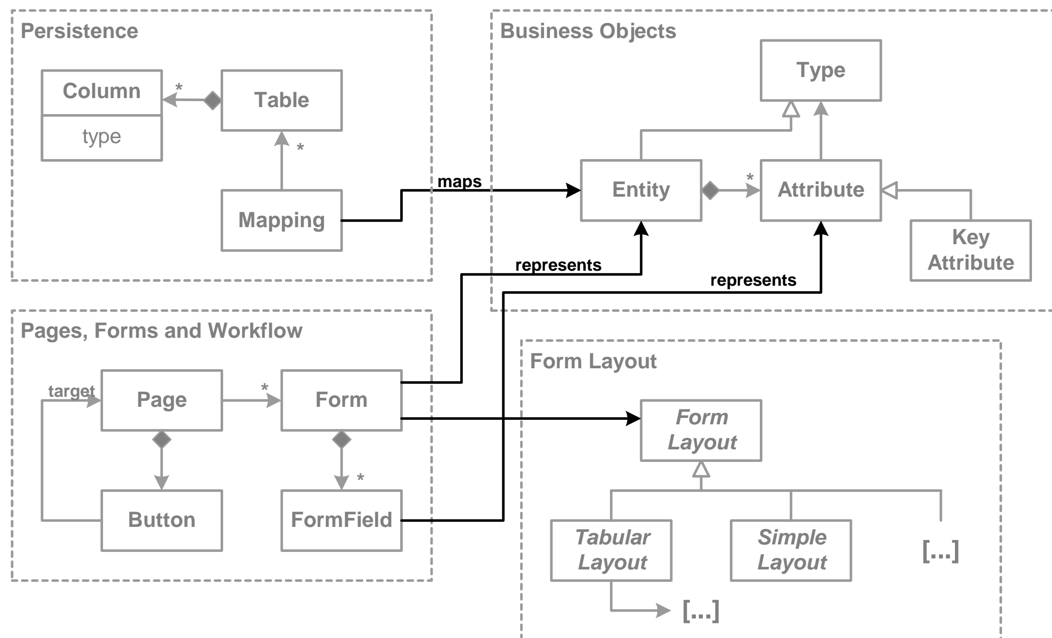
```
<node name="sensorsOutside" tracing="on">
  …
</node>
```

Aufgrund dieser Angabe (tracing="on") generiert der Generator obigen konkreten Aspekt.

AO Modellierung

Die vielleicht wichtigste Form, wie MDSD und AOSD gemeinsam verwendet werden können ist vielleicht auch die am wenigsten offensichtliche, da sie sich nicht um Aspekte im Zielsystem dreht, sondern *um die in dem Modellen*! Stellen Sie sich vor, sie erstellen eine typische Webanwendung. Diese enthält verschiedene Belange, unter anderem ein Business-Objektmodell, dessen Persistenz, Webseiten, Formulare und Workflow, sowie schlussendlich das Layout der Webseiten. All diese Dinge müssen modelliert werden, um eine Anwendung generieren zu können. Allerdings ist es völlig unpraktikabel, all dies in einem Modell zu tun. Das Modell wird erstens sehr groß, und vermischt zweitens die oben erwähnten Aspekte in einem großen "Kuddelmuddel". Außerdem werden Sie mir sicherlich zustimmen, dass man für diese verschiedenen Aspekte verschiedene Modellierungsprachen benötigt. Es ist beispielsweise absolut praktikabel, mittels UML das Business-Objektmodell zu beschreiben; für Webseitenlayout eignet sich UML allerdings nicht, Profile hin oder her. Um dieses Problem in den Griff zu bekommen, muss man also für jeden Aspekt eine eigene Modellierungssprache verwenden, und es dann dem Generator überlassen, die verschiedenen Modelle zu lesen und zu einem sinnvollen Ganzen zusammen zu fügen. Mit anderen Worten: Der Generator übernimmt die Funktion des Aspektweavers!

Natürlich müssen Sie, damit das funktioniert, auch ein Joinpoint-Modell definieren. Dies ist aber fast trivial, da Sie ja sowieso Metamodelle für die verschiedenen DSLs definieren. Im Rahmen dieser Metamodelle ist es ein leichtes, anzugeben, welche Metamodellelemente vom Generator zusammengefügt werden sollen. Die folgende Abbildung zeigt die vier (Teil-)Metamodelle des obigen Webapplikationsbeispiels, und ihren Zusammenhänge.



Zusammenfassung

Was lernen wir nun daraus? Erstens sind MDSD und AOSD wesensverwandt. Zweitenes lassen sich beide sinnvoll miteinander einsetzen – allerdings bedeutet dies nicht zwangsläufig, dass man deshalb auch eine AOP Sprache verwenden muss! Übrigens: Wenn Sie sich von dem Artikel nur eines merken, dann den letzten Abschnitt, AO Modellierung. Ohne diesen Ansatz werden sie keine ernsthaften, großen MDSD Projekte durchführen können! Feedback an voelter@acm.org ist wie immer erwünscht!

Referenzen

AOSD AOSD Website, http:// aosd.net

GOF Gamma et. al., Design Patterns, Addison-Wesley 1995

MV04a Markus Völter, Eigene „Aspekte“ in EJBs, JavaSpektrum ??/04

MV04b Markus Völter, Modellgetriebene Softwareentwicklung – eine Einführung, ObjektSpektrum ??/04

MV05 Markus Völter, Models and Aspects, Patterns for Handling CCC in MDSD, http://www.voelter.de/data/pub/ModelsAndAspects.pdf

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

POSA2 Schmidt, et. al., Pattern-Oriented Software Architecture Vol. 2, Wiley, 2000

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