Modellgetriebene, Komponentbasierte Softwareentwicklung

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

- Teil 1 -

Komponentenbasierte Entwicklung und Modellgetriebene Entwicklung passen sehr gut zueinander – dies hat sich im Laufe der letzen paar Projekte in den verschiedensten Domänen gezeigt. In diesem zweiteiligen Artikel möchte ich darauf eingehen, wie man konkret Modellgetrieben und Komponentenbasiert entwickeln kann in dem ich als zentralen Aspekt ein Metamodell vorstelle, welches CBD aus den drei wichtigsten Perspektiven beleuchtet. Im zweiten Teil des Artikels zeige ich dann typische Variationen dieses Metamodells auf.

About

Diesen Artikel wollte ich schon lange schreiben. Leider musste ich vorher eine ganze Reihe anderer, sowie das eine oder andere Buch produzieren – all dieses Material dient quasi als Quelle für den vorliegenden Artikel (und wird ggfs. referenziert). Letztlich beschreibt dieser Artikel die Quintessenz aus den letzten paar Projekten in den vergangenen zwei Jahren. Er zeigt auf, wie man Komponenbasierte und Modellgetriebene Softwareentwicklung kombiniert – und warum das Sinn macht. Natürlich kann man zu diesem Thema noch eine ganze Menge mehr schreiben als ich das in diesem Artikel tue; aber ein Großteil steht eben in dem anderen, referenzierten Material. Keine Angst, der Artikel ist aber auch verständlich ohne all die anderen Dinge gelesen zu haben ☺

Kasten: Modellgetriebene Softwareentwicklung

Von Modellgetriebener Softwareentwicklung (MDSD) spricht man, wenn Software teilweise oder vollständig aus Modellen generiert wird. Dabei ist nicht, wie bei traditioneller Entwicklung üblich, Applikationslogik 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 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. 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[1]. Dafür sind Tools notwendig, die für die entsprechende DSL erstellt und weiterentwickelt werden müssen. Die Vorteile dieses Ansatzes sind u.a. größere Entwicklungseffizienz, bessere Integration der Fachexperten, leichtere Änderbarkeit von Software, verbesserte (Umsetzung der) Softwarearchitektur, sowie die Möglichkeit, Fachlogik leichter auf andere Plattformen portieren zu können.

Hintergrund

In den letzen Jahren habe ich eine Reihe von Projekten betreut, bei denen ich entweder wegen Komponentenarchitekturen oder Modellgetriebener Softwareentwicklung [SV05] zum Einsatz kam. Interessanterweise war es in allen Projekten schlussendlich so, dass alle Projekte *beides* eingesetzt haben – unabhängig davon, mit welchem der beiden Ansätze sie gestartet waren. Das legt nahe dass es ein symbiotisches Verhältnis zwischen den beiden Technologien gibt. Die Projekte waren in verschiedenen Umfeldern angesiedelt, von eingebetteten Echtzeitsystemen im Fahrzeug über Spring/P2P Anwendungen bis zu klassischen Enterprise-Systemen basierend auf J2EE und EJB.

Kasten: Komponentenbasierte Entwicklung

Bei Komponentenbasierter Entwicklung strukturiert man die zu erstellende Software in einzelne Komponenten. Die Definition, was eine Komponente ist variiert, man ist sich aber weitgehend einig über die folgenden Punkte:

Eine Komponente ist ein funktional abgeschlossener Softwarebaustein. Sie definiert die Dienste die sie anbietet sowie die Ressourcen die sie zur Erbringung der Dienste benötigt.

Ich möchte diese Definition ergänzen um die Tatsache, dass Komponenten in einer wohl definierten Ablaufumgebung existieren (meist als Container bezeichnet), der bestimmte, wohl definierte (querschnittliche, meist technische) Dienste im Auftrag der Komponenten erledigt.

[1] Man könnte sie auch interpretieren, aber aus verschiedenen Gründen ist das in der Praxis oft nicht sinnvoll.

Grundsätzliches Vorgehen

MDSD und CBD ergänzen sich insofern sehr gut, als man die Struktur von Komponentenbasierten Systemen sehr gut mittels Modellen beschreiben kann um daraus dann einen Großteil der technischen Infrastruktur abzuleiten, sprich zu generieren. Dabei kommen (mindestens) drei Modelle zum Einsatz, wobei jedes einen spezifischen Viewpoint beschreibt.

- Typ-Modell: Hier werden die Komponenten als Datentypen (wie Klassen in der OO) beschrieben. Dabei werden üblicherweise die Operationen separat in Interfaces definiert. Wie bei Komponenten üblich [CS99] beschreibt eine Komponente nicht nur, welche Interfaces sie anbietet, sondern auch, welche sie benötigt. In aller Regel muss man auch in der Lage sein, eigene (komplexe) Datentypen zu definieren.
- Composition-Modell: Hier werden logische Instanzen[2] (in Abbildung 1 als *Deployment* bezeichnet, Grund siehe Fußnote 1) der Komponenten beschrieben, sowie ihre „Verdrahtungen" untereinander. Damit wird ein logisches System als Kollaboration von Komponenten definiert.
- System-Modell: Hier wird nun die Hardware- und Prozessstruktur definiert, auf die die Instanzen deployt werden sollen.

[2] Der Begriff Instanz ist hier etwas vorsichtig zu verwenden. Im Rahmen eines Containers (bspw. EJB) können von einer Komponenten durchaus viele Instanzen (im Sinne von Objekten) existieren, auch wenn es aus der Sicht des Komponentenspezifizierers genau eine (logische) Instanz gibt. Dies tritt bspw. auf, wenn der Applikationsserver Pooling verwendet [VSW02]. Aus diesem Grund verwende ich lieber den Begriff Deployment, in dem Sinne, dass eine „logische Instanz" deployt wurde; dabei ist es egal, wie viele technische Instanzen existieren.

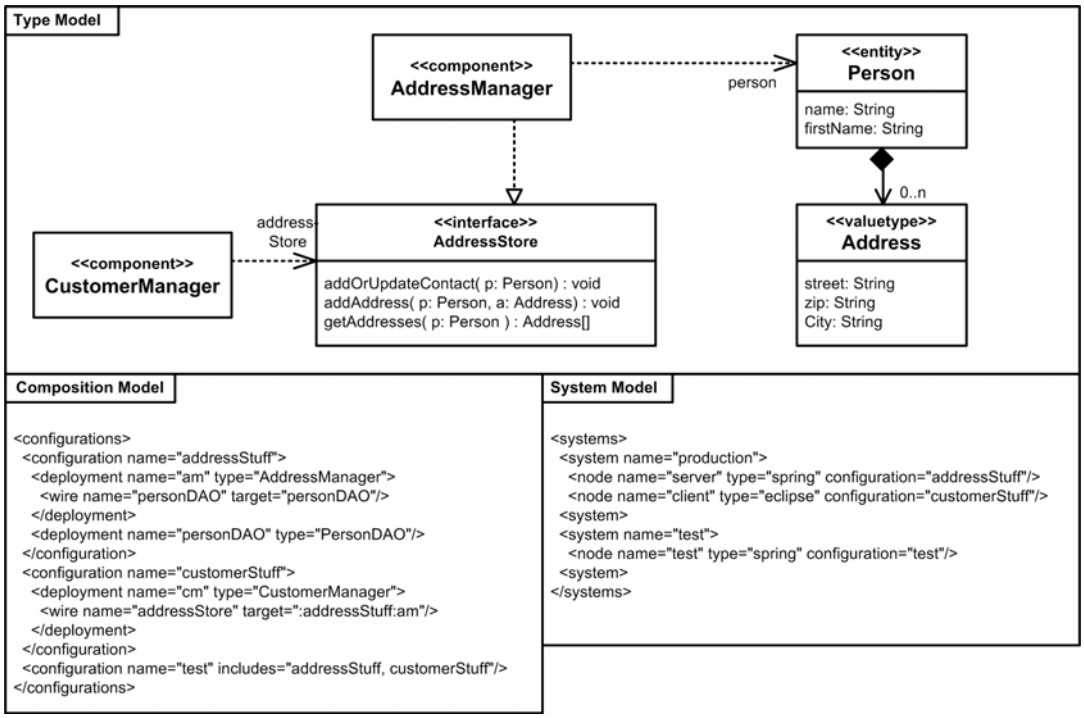

Abbildung 1: Beispiel für Systemmodell

Man beachte, dass wir hier nicht nur eine Reihe isolierter Komponenten definieren. Es wird tatsächlich ein System beschrieben – erkennbar an der Tatsache, dass wir die Verdrahtung von Komponenten, Prozesse, Systemknoten, Servertypen etc. beschreiben! Dieses Vorgehen – bei vielen Infrastrukturen und Komponentenmodellen nicht explizit vorhanden – ist *extrem* nützlich, da sich erst daraus das volle Generierungspotential ergibt. Dazu gleich mehr.

Eine oft gestellte Frage ist übrigens die nach der Modellierungssprache, oder besser: nach deren konkreter Syntax. Im Beispiel in Abbildung 1 verwenden wir eine Mischung aus UML (für das Typmodell) und XML für die anderen beiden Aspekte. Das ist in gewisser Weise willkürlich, man kann auch alles mittels Box-And-Line Diagrammen machen (UML, oder selbst generierte Editoren [RV05]) oder alles mit XML, oder anderen textuellen Sprachen. Der Ansatz oben hat sich aber durchaus bewährt – UML Tools sind quasi überall vorhanden und die Typmodellierung ist ähnlich der klassischen OO Klassenmodellierung. Da sich die anderen Aspekte üblicherweise regelmäßiger ändern und die Informationen darin a) weniger komplex, und b) nicht gut per UML beschreibbar sind, eignet sich da XML recht gut.

Was man nun daraus generieren kann

Stellt sich nun die Frage, was man aus diesem Modellen generieren kann. Nun, das hängt natürlich davon ab, in welchem Systemumfeld man sich bewegt. In eingebetteten Systemen wird man im Detail andere Artefakte generieren als in der EJB Welt. Es lassen sich allerdings einige Dinge allgemeingültig sagen:

- Basisklassen für die Komponentenimplementierung. Man wird hier üblicherweise das Dependency-Injection Pattern verwenden, um dem Implementierer den Zugriff auf die „verwendeten" Interface zu geben (und *nur* auf die!).
- Build-Skripte, die die Komponenten bauen und packen
- Je nach Zielplattform die entsprechenden Deskriptoren (EJB, Spring) oder den nötigen Code (Osek) um die betreffenden Instanzen zu erzeugen und zu verdrahten.
- Für den Fall das miteinander redende Komponenten auf verschiedenen (Netzwerk-)Knoten liegen, kann man Remote Proxies [VKZ04] generieren, bzw. die für die Plattform nötigen Artefakte und Deskriptoren erzeugen.
- Falls die Datenstrukturen persistent sind, wird man den nötigen Persistenzcode erstellen, ggfs. natürlich unter Verwendung eines entspr. Frameworks wie Hibernate oder JDO.

Eine essentielle Rolle in dieser Betrachtung spielt der Komponentencontainer [VSW02]. Der Container ist dafür zuständig, den Komponenten querschnittliche, meist technische Aspekte zur Verfügung zu stellen. Im Enterprise Umfeld sind das die üblichen Verdächtigen Transktionsmanagement, Security, Scalability, etc. Je nach Plattform ist bereits ein passender Container vorhanden (EJB) oder muss für das betreffende System erstellt werden – entweder per AOP und Server-Infrastruktur (Spring) oder per Codegenerierung (Osek). Letztendlich handelt es sich dabei um Aspektorientierung im Rahmen von MDSD, beschrieben in [MV05a].

Einordnung in der Architekturprozess

Generell bin ich der Meinung dass es wichtig ist, Softwarearchitektur zunächst unabhängig von der Realisierungstechnologie zu definieren [MV05b]. Klare Konzepte – also ein Architekturmetamodell – sind essentiell. In einem zweiten Schritt kann man nun diese konzeptionelle Architektur auf eine oder mehrere Plattformen abbilden. Mittels MDSD wird man diese Abbildung dann weitestgehend automatisieren. Letztendlich steigert dieses Vorgehen die Qualität der Architektur, weil das Vorgehen die genaue, formale Definition von Bausteinen und Konzepten erzwingt. Außerdem kann man damit leichter für mehrere Plattformen parallel entwickeln – man muss ja nur mehrere Abbildungen definieren.

Nun werden Sie vielleicht sagen: Multi-Plattform, wer braucht das schon. Meine Antwort darauf ist: Jeder. Und zwar nicht unbedingt so wie man das oft im Rahmen der MDA hört, also etwas überspitzt formuliert „per Knopfdruck von Java nach .NET“. Im Rahmen von Projekten muss man aber durchaus im Laufe der Evolution des Systems die Plattform zumindest auf eine neue Version upgraden. Außerdem wollen Sie ihre Unit-Tests ja nicht im Applikationsserver oder in einer verteilten Umgebung durchführen, sondern mit möglichst wenig „Overhead“ innerhalb der IDE. Sie brauchen also zumindest eine abgespeckte, lokale Umgebung und das (verteilte, skalierende, etc.) Zielsystem.

Einen absolut zentralen Stellenwert stellen also die Architektur*konzepte* dar, weswegen wir im Folgenden auch auf genau diese eingehen werden – in Form eines Metamodells, welches Sie in der Praxis tatsächlich direkt anwenden können. Im zweiten Teil dieses Artikels werden wir dann oft vorkommende Varianten dieses Metamodells diskutieren.

Basismetamodell

Beginnen wir mit dem Metamodell für Komponenten. Dieses Metamodell definiert, mit welchen Ausdrucksmitteln Komponenten beschrieben werden können. Es birgt relativ wenig Überraschungen und ist in Abbildung 2 gezeigt. Interessant ist höchstens noch die „Objektifizierung“ der Beziehung *requiredInterface*. Diese bekommt ein eigenes Assoziationsobjekt, damit das Ding einen Name bekommen kann und später noch weitere Attribute aufnehmen kann.

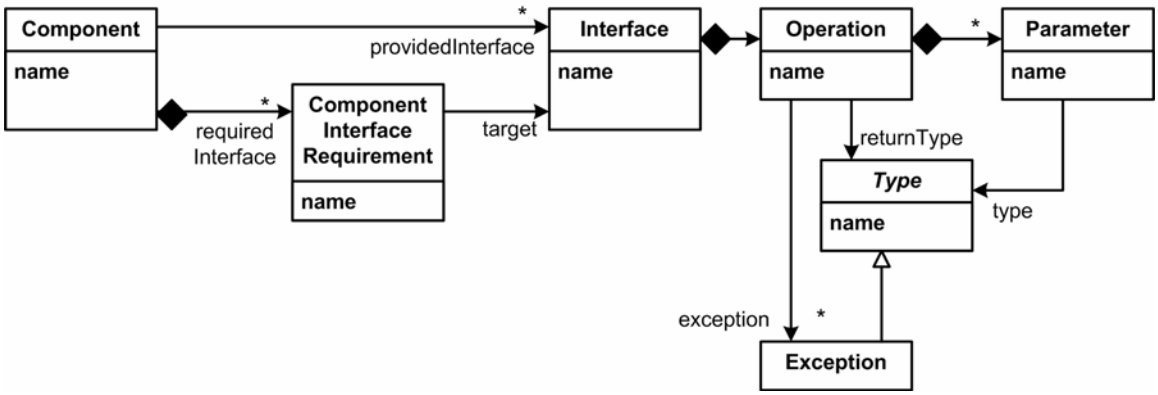

Abbildung 2: Komponenten-Metamodell

Auch das Metamodell für Datentypen in Abbildung 3 ist eigentlich wie erwartet. Datentypen beschreiben Datenstrukturen, wohingegen Komponenten „Stücke von Funktionalität“ beschreiben. Komponenten arbeiten also u.a. mit Daten(typen). Das Metamodell unterscheidet zunächst zwischen primitiven und komplexen Datentypen. Komplexe Datentypen können entweder (persistente) Entitäten sein, oder Data Transfer Objects. Komplexe Datentypen können Attribute haben die – neben einem Namen – einen

primitiven Typ besitzen müssen. Schlussendlich können Entitäten miteinander in Beziehung stehen. Dabei können Multiplizitäten und die Navigierbarkeit angegeben werden.

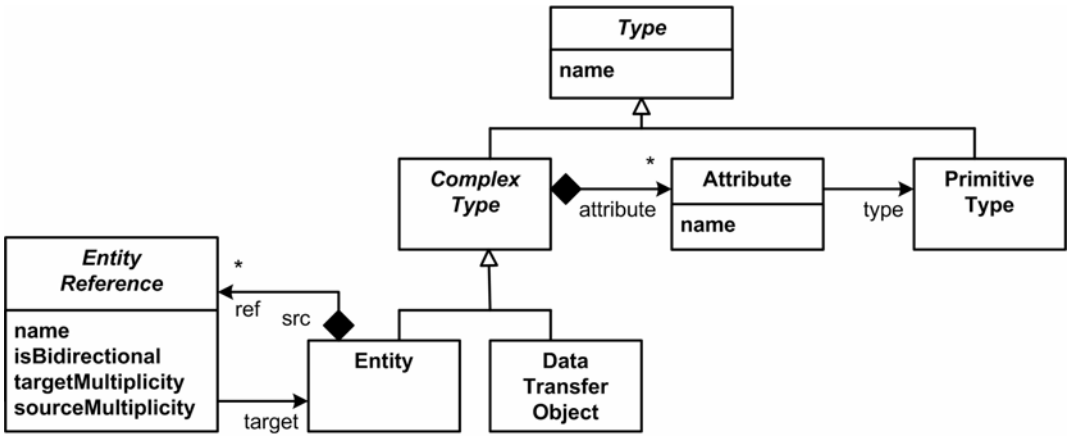

Abbildung 3: Metamodell für Datentypen

Nun zu der Definition der Instanzen und deren Beziehungen untereinander. Eine Konfiguration umfasst eine Menge von Instanzen, die wiederum Verdrahtungen haben (*Wires*). Zu beachten sind die beiden Constraints sowie der Bezug zum Typ-Modell.

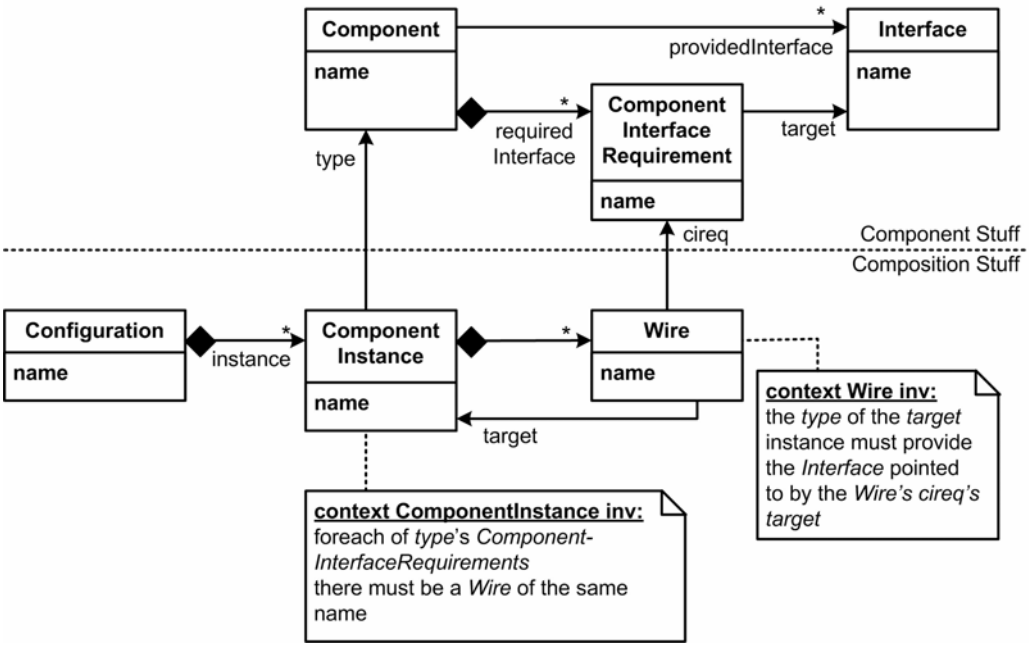

Abbildung 4: Instanzbildung und Verdrahtung

Schlussendlich das Systemmodell in Abbildung 5. Es definiert ein System bestehend aus Knoten und Containern, die jeweils Komponenteninstanzen beherbergen.

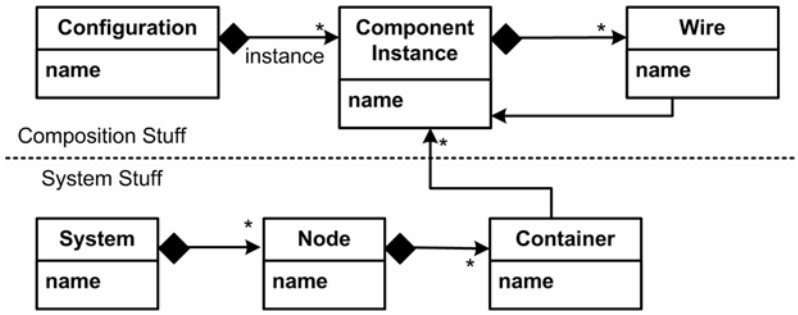

Abbildung 5: Das Systemmodell

Viewpoint-Abhängigkeiten

Um das Ziel zu erreichen, die gleiche Software auf mehreren Infrastrukturen laufen lassen zu können, sind verschiedene Dinge nötig – siehe unten. Unabdingbar ist jedoch, dass die Abhängigkeiten zwischen den Viewpoint-Modellen „richtig rum" sind (siehe Abbildung 6).

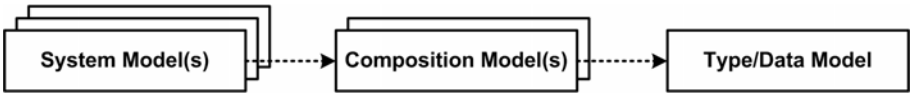

Abbildung 6: Abhängigkeiten zwischen den Metamodellen

Typischerweise wird man eine Menge von Komponenten/Datenstrukturen definieren – und diese dann auch manuell ausimplementieren. Ein bestimmtes Szenario betsteht nun aus einer Menge von Instanzen (einer Composition). Dieses Szenario will man möglicherweise lokal zu Testzwecken oder auch verteilt („ernsthaft") im richtigen System laufen lassen.

Man muss also in der Lage sein, für eine bestimmte Menge von Komponenten *mehrere* Compositions zu definieren und diese wiederum auf *mehreren* Systemen laufen zu lassen. Deshalb dürfen die Abhängigkeiten zwischen den Modellen (und damit den Metamodellen) nur in der in Abbildung 6 gezeigten Richtung verlaufen.

Aspektmodelle

Im obigen Abschnitt haben wir die drei grundlegenden Viewpoints besprochen, sowie deren Metamodell. Allerdings reichen diese drei Viewpoints nicht immer aus. In vielen Fällen braucht man für bestimmte Aspekte der Anwendung weitere Informationen. Statt

diese nun aber in die drei Viewpoints „reinzubauen“, hat es sich bewährt, diese Aspekte auch als solche zu behandeln und in separate Aspektmodelle zu verlagern (siehe Abbildung 7). Es ist dann Aufgabe des Generators, die betreffenden Modelle zu verweben und Code zu generieren, der alle nötigen Aspekte berücksichtigt. Übrigens ist es – ganz im Sinne von AOSD – möglich, dass ein Aspektmodell Modellelemente in mehr als einem Viewpoint betrifft.

Übrigens gilt für die allermeisten der unten beschriebenen Aspekte, dass das daraus abgeleitete Generat meist nicht Code ist, sondern Konfigurationsfiles für die Zielplattform, denn die typischen wichtigen Aspekte einer Domäne werden eben genau von den entspr. Plattformen die sich für die Domäne eignen unterstützt.

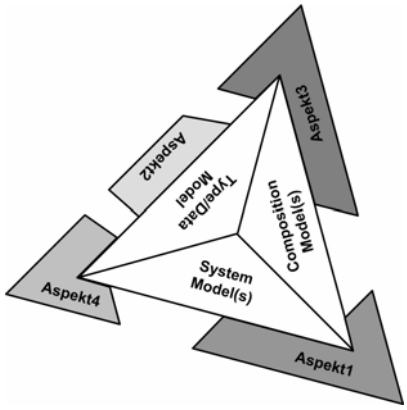

Abbildung 7: Die drei Grund-Viewpoints und Aspektmodelle

Ich möchte im Folgenden einige typische Aspekte beleuten, die sich als Aspektmodelle eignen. Ein typischer Aspekt, vor allem in Businessanwendungen ist **Persistenz**. Immer dann, wenn es nicht möglich ist, das Persistenzmapping direkt per fester Regeln aus dem Typmodell abzuleiten, macht es Sinn, ein separates Modell zu definieren, welches beispielsweise die Tabellennamen, den Umgang mit Polymorphismus, etc. angibt. Auf der anderen Seite ist es, wenn man gute Persistenzframeworks wie bspw. Hibernate verwendet oft unnötig, solche Dinge anzugeben. Was aber in aller Regel bleibt ist die Definition von Queries für DAOs. Diese müssen ja in einer Abfragesprache (SQL, OQL, HQL, EJB-QL) definiert werden. Es macht Sinn, dies in einem separaten, textuellen Modell zu tun.

Ein weiterer wichtiger Aspekt ist **Security** (im Sinne von Authorisierung). Die Definition von Benutzern und Gruppen sowie die Definition, wer worauf wie zugreifen darf ist ein Klassiker. Die Benutzer und Gruppen werden ja oft in entspr. Datenbanken verwaltet. Die Zuordnung der Zugriffsrechte muss sich in irgendeiner Art und Weise an den Komponenten, Interfaces und Datenstrukturen orientieren. Das bedeutet, dass man oft ein Modell haben wird, welches definiert, welche Benutzergruppen auf welche Artefakte zugreifen dürfen. Auch dies macht man sinnvollerweise in einem separaten Modell –

welches dann möglicherweise zur Laufzeit des Systems dynamisch ausgewertet („interpretiert“) wird, und nicht zur Codegenerierung dient. Wenn man eine entspr. Infrastruktur zur Verfügung hat, kann man diese Beschreibungen auch einfach in der betreffenden Infrastruktur definieren – dann wird aus dem Aspektmodell hier einfach eine bestimmte Security-Konfiguration.

Masken, Formulare und Pageflow (v.a. bei Webanwendungen) sind ein weiterer Aspekt, der sich für ein separates Modell eignet. Man wird dafür spezielle Komponententypen einführen, und die Formulare und Masken werden auf anderweitig definierte Datenstrukturen Bezug nehmen. Auch hier gibt es – bei Verwendung entspr. Frameworks wie Struts – ggfs. bereits fertige Editoren, um die betreffenden Modelle (Struts: Config-Datei) zu erstellen.

Im Bereich eingebetteter Systeme hat man oft das Problem, dass man **Timing und andere QoS-relevante Aspekte** (wie Speicherverbrauch oder Bandbreitennutzung) zusichern, oder zumindest zur Laufzeit überwachen muss. Dazu muss man diese zunächst in einem Modell beschreiben. Daraus kann der Generator

- entweder Code generieren, der dafür sorgt, dass die modellierten QoS-Aspekte zur Laufzeit eingehalten werden (nachdem er während der Generierung validiert hat, dass das möglich ist),
- die zugrunde liegende Infrastruktur (Echtzeit-OS) entsprechend konfigurieren,
- oder zumindest Watchdogs in den Code hineingenerieren, die melden, wenn es zur Laufzeit zu Problemen mit dem betreffenden Aspekt kommt.

In allen Fällen muss man (teils umfangreiche) Angaben im Modell machen; insbesondere für Timing ist es nötig, komplette Kollaborationen zu modellieren inkl. deren Zeitdauer und max. Auftrittshäufigkeit. Auch die Verifikation der Validität zur Generierungszeit bzw. die Ableitung von Konfigurationsdaten um das Ziel zu erreichen ist oft hochkomplex und eine Aufgabe für Spezialisten – und damit außerhalb des Scopes dieses Artikels. Wichtig ist mir nur zu erwähnen, dass eben auch dieser Aspekt oft sinnvoll in separate Modelle ausgelagert werden sollte.

Ein weiterer Aspekt, der in eingebetteten Systemen essentiel ist und auch in Enterprise-Anwendungen immer relevanter wird, ist die Beschreibung von **Diagnose und Monitoring**. Dabei werden Systemparameter beschrieben, die entweder extern sichtbar (also per Monitoringwerkzeug zugänglich) sind oder in irgendeiner Protokolldatei zur späteren Auswertung gespeichert werden.

Abschließen möchte ich mit einem weitaus profaneren, wenn auch nicht unbedingt weniger nützlichen Aspekt: **Packaging und Deployment**. Es ist ja altbekannt dass man idealerweise komplette Build-Skripte mit generiert, ggfs. sogar Batch-Dateien die die entspr. Artefakte auf das Zielsystem deployen (siehe Systemmodell!). Oft lassen sich die

nötigen Namen sowie Verzeichnis- und Paketstrukturen auf den oben eingeführten Modellen ableiten – manchmal ist die allerdings nicht vollständig möglich. In diesem Fall benötigt man eben ein entspr. Aspektmodell, um die nötigen Informationen im System unterzubringen.

Nochmal: Metamodell vs. Generierung

Das Metamodell oben sagt natürlich nichts über die Umsetzung in Code aus. Das soll es auch gar nicht: es beschreibt lediglich, *was* wir uns von unserer Architektur erwarten, aber nicht *wie* wir es umsetzen. Diese Trennung – und die damit verbundene Isolation der technischen, plattformspezifischen Aspekte – ist ein zentraler Baustein für technologieunabhängige Architektur [MV05b]. Die Codegenerierungstemplates die auf obigem Metamodell aufsetzen definieren dann das *wie*, nachdem man sich überlegt hat, welche Plattform man wie verwenden möchte. Hier ist der Punkt wo plattformspezifische Design Patterns (J2EE Blueprints, etc.) bzw. allgemein das Wissen im Umgang mit der Zieplattform zum Einsatz kommt. Und außerdem erlaubt dieser Ansatz natürlich auch die relativ schmerzfreie Abbildung auf verschiedene Zielplattformen – siehe die Diskussion zum Thema Testen weiter oben.

Eine weitere Sache sei erwähnt: Natürlich ist es besonders einfach (und nahe liegend) ein Feature ins Metamodell einzubauen, wenn die betreffende Zielplattform das Feature direkt unterstützt (also bspw. einen Event-Mechanismus vorzusehen, wenn die Zielplattform asynchrone Events unterstützt). Auf der anderen Seite sollte man sich davor hüten, nur „die“ Zielplattform im Metamodell abzubilden. Die allermeisten Plattformen sind nämlich gerade nicht in der Lage, all die oben genannten Konzepte direkt umzusetzen. Entsprechende, nicht-triviale Codegenerierung ist dann nötig.

Im allerschlimmsten Fall muss man sich alles selbst bauen. Doch auch dazu gibt es in Form entspr. Patterns Hilfe [POSA2, POSA3, VSW02, VKZ04])

Komponentenimplementierung

Da war doch noch was… wir haben bis jetzt noch nicht über die Implementierung einer Komponente gesprochen! Ohne Implementierung machen Komponenten natürlich keinen Sinn. Das wäre so, wie wenn man in der OO Programmierung nur die Signatur von Klassen definieren würde. Ein lauffähiges System stellt das noch nicht dar. Natürlich ist die Implementierung architekturell oft (wenn auch nicht immer) zweitrangig, weswegen sie hier keine prominente Rolle spielt. Trotzdem möchte ich kurz auf das Thema eingehen.

In den allermeisten pragmatischen Ansätzen zu MDSD wird die Implementierung *im allgemeinen* ausgeklammert. Das bedeutet, dass man *im allgemeinen* beispielsweise eine abstrakte Basisklasse generiert, von der der Entwickler dann erben muss, um in der Unterklasse mittels manueller Programmierung die Implementierung der Komponentenmethoden zur Verfügung zu stellen. Warum macht man das so? Weil es in den allermeisten Toollandschaften keine (modellbasierte) Sprache gibt, mit der man

Algorithmik – also eben die Implementierung – sinnvoll ausdrücken kann. Dies gilt allerdings nur für den *Allgemeinfall*[3]. Ich betone den *Allgemeinfall* deswegen so, weil man für bestimmte *Arten* von Komponenten (siehe Teil 2 des Artikels) natürlich schon definieren kann, wie sie implementiert werden. Beispiele:

- Komponenten, die Daten speichern können komplett generiert werden. Die Implementierung der CRUD Operationen ist direkt aus dem assoziierten komplexen Datentyp abzuleiten, wenn man die Persistenztechnologie kennt.
- Komponenten die Geschäftsprozesse realisieren kann man beispielsweise mittels einer Zustandsmaschine beschreiben. Aus einer solchen Beschreibung kann Code generiert werden, der den Prozess implementiert.
- Legacy-Adapter Komponenten bekommen eine (textuelle) Beschreibung der Datenstrukturen des Legacysystems. Ein Interpreter, der diese Beschreibungen interpretiert dient technisch gesehen als Implementierung.

Alle diese Beispiele haben gemein, dass man für eine spezielle *Art* von Komponente eine *für die Art passende* DSL verwendet. Die *im Allgemeinen* von Hand geschriebene Unterklasse lässt sich nun *für diese wohl definierten Spezialfälle* aus den spezifischen Modellen, mit einem für die Art passenden Generator, generieren. Ziel ist es also, im Rahmen der Architektur möglichst viele solcher spezieller Komponenten*arten* samt passender DSL zu identifizieren. Technisch gesehen handelt es sich hierbei bereits um Kaskadierte MDSD – siehe unten.

Code statt Text?

Man könnte sich nun natürlich fragen: Warum beschreibt der das alles in einem (bzw. zwei) Artikel, und baut nicht gleich das entspr. Metamodell für den Generator zur direkten Verwendung. Nun, dafür gibt es verschiedene Antworten:

- Zum einen geht es mir zunächst einmal darum, die Idee hinter dem ganzen zu kommunizieren. Ich möchte, dass (komponentenbasierte) Softwarearchitektur mit anderen Augen gesehen wird als „EJB rauf und runter“ und möchte zum Nachdenken anregen.
- Zum zweiten ist es so, dass die Metamodelle natürlich nicht *genau so* verwendet werden, sondern in Varianten, die teilweise in Teil zwei dieses Artikels beschrieben sind. *Ein* Metamodell ist es also gar nicht.
- Und außerdem gibt es natürlich noch verschiedene andere Generatoren – neben openArchitectureWare ☺. Würde ich einfach das Metamodell für oAW implementieren, wäre es für andere Generatoren anwendbar.

[3] Und das auch nur so lange, bis sich die Action Semantics in der Toolwelt durchsetzen.

Where do we go from here?

Natürlich hört MDSD nicht mit der Formalisierung und Automatisierung der Architektur auf. Schlussendlich wollen wir fachliche Konzepte modellgetrieben umsetzen. Allerdings – und das ist ja allgemein bekannt – ist Modularisierung eine sinnvolle Sache, und das gilt auch für MDSD. Konkret bedeutet das, dass wir die fachlich spezifischeren Dinge *auf Basis der Architektur-Infrastruktur* wie wir sie hier gebaut haben, umsetzen. Man nennt dies auch kaskadierte MDSD. Abbildung 8 zeigt das Prinzip, Details und ein Beispiel finden sich in [MV05c].

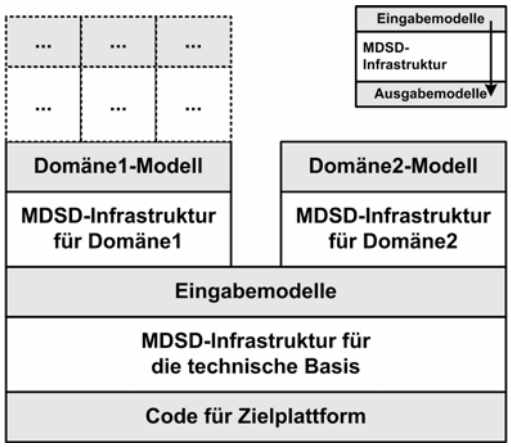

Abbildung 8: Kaskadierte MDSD

Im zweiten Teil dieses Artikels werden wir dann häufige Variationen dieser Metamodelle diskutieren. Dabei gehen wir auf Dinge wie Asynchronität, Events, Layering, Dynamische Verdrahtung, etc. ein.

Vielen Dank an Eberhard Wolff für sein Feedback zu dem Artikel!

Referenzen

[CS99] Clemens Szyperski, Component Software, Addison-Wesley 1999

[MV05a] Markus Völter, Models and Aspects, http://www.voelter.de/data/pub/ModelsAndAspects.pdf

[MV05b] Markus Völter, Software Architecture Patterns, http://www.voelter.de/data/pub/ArchitecturePatterns.pdf

[MV05c] Markus Völter, Kaskadierte, Modellgetriebene Entwicklung und Modelltransformationen, http://www.voelter.de/data/articles/CascadingAndMT.pdf

[POSA2] Schmidt, Stal, Rohnert, Buschmann, POSA 2 - Patterns for Concurrent and Networked Objects, Wiley, 2000

[POSA3] Kircher, Jain, POSA 3 – Patterns for Resource Management, Wiley, 2004

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

[SV05] Stahl, Völter, Modellgetriebene Softwareentwicklung, dPunkt, 2005

[VKZ04] Völter, Kircher, Zdun, Remoting Patterns, Wiley, 2004

[VSW02] Völter, Schmid, Wolff, Server Component Patterns, Wiley, 2002