Modellgetriebene Softwareentwicklung

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

Modellgetriebene Entwicklung (MDSD[1]) und MDA sind in aller Munde. Leider gibt es dabei viele Missverständnisse, Fehlvorstellungen und Vorurteile. Dieser Artikel soll hier für Klärung sorgen. Er stellt MDSD-Kernkonzepte vor und zeigt, wie sich modellgetriebene Softwareentwicklung im Allgemeinen von MDA (als Spezialisierung von MDSD) unterscheidet. Des Weiteren möchte ich Konsequenzen für den Entwicklungsprozess aufzeigen sowie darauf eingehen, wie Aspektorientierung mit MDSD in Bezug steht. Aus Platzgründen kann nicht auf alle Details und Aspekte eingegangen werden, der Artikel soll *ausdrücklich* einen möglichst breiten Überblick über das Gebiet geben – weitere Details finden sich z.B. unter [MV04, SVB04].

Einführung

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, 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. Dafür sind Tools notwendig, die für die entsprechende DSL erstellt und weiterentwickelt werden müssen. Die Vorteile dieses Ansatzes sind u.a.

- größere Entwicklungseffizienz

[1] engl. Model-Driven Software Development

- bessere Integration der Fachexperten
- leichtere Änderbarkeit von Software
- verbesserte (Umsetzung der) Softwarearchitektur
- sowie die Möglichkeit, Fachlogik leichter auf andere Plattformen portieren zu können.

Wie dieser kurze Abriss zeigt, sind einige der von MDSD verwendeten Konzepte doch etwas anders als im Rahmen traditioneller Softwareentwicklung. Die Übersetzung von abstrakten Spezifikationen in eine weniger abstrakte, ausführbare Implementierung ist zwar nichts neues (Compiler machen nichts anderes!), allerdings obliegt es nun dem Entwicklungsteam die Sprache und den Übersetzer zu definieren und zu implementieren – man entwickelt also nicht nur Software, sondern auch „Software Fabriken": Infrastrukturen, um Software zu entwickeln (siehe auch [GS04]).

Begriffsdefinition und Wichtige Konzepte

Dieser Abschnitt versucht auf sehr knappem Platz die wichtigsten Konzepte zu erklären – was zu einer sehr verdichteten Beschreibung führt. Bitte lesen sie trotzdem weiter, die späteren Abschnitte sind leichter verdaulich☺.

Abbildung 1 zeigt eine Mindmap die die wichtigsten Konzepte Modellgetriebener Softwareentwicklung darstellt und in Beziehung setzt. Im Zentrum von MDSD steht – naheliegenderweise – das Modell. Dieses ist kein Modell, wie wir es von klassischer UML kennen, in dem die programmiersprachlichen Abstraktionen wie Klassen, Objekte und ihre Beziehungen durch Rechtecke und Linien ausgedrückt sind, sondern ein Modell der Applikationsfunktionalität auf dem Abstraktionsniveau der Anwendungsdomäne, ausgedrückt mit einer Domänenspezifischen Sprache (siehe Abb. 3). Diese Sprache definiert die Bedeutung des Modells, also dessen Semantik. Das Modell ist also präzise in dem Sinne, dass seine Bedeutung durch die DSL exakt definiert ist, es stellt formalisiertes Wissen über die Anwendungsdomäne dar. Die DSL, genauer: deren konkrete Syntax, kann entweder textuell oder grafisch sein, auch tabellarische oder andere Notationen können verwendet werden. Entscheidender ist die abstarkte Syntax, auch als Metamodell bezeichnet (siehe Abb. 4). Das Metamodell definiert die Modellelemente, mit denen die Modelle der betreffenden Domäne erstellt werden können. Neben einer abstrakten und einer konkreten Syntax muss jede Sprache auch eine Semantik besitzen, die die Bedeutung der Modelle genau definiert. Im Falle von MDSD wird die Semantik üblicherweise transformationell definiert, was bedeutet, dass die Modelle mittels Transformationen auf eine andere, wohlbekannte Sprache (oft eine 3GL) abgebildet werden. Diese Abbildungsregeln definieren die Bedeutung der DSL bzw. der damit ausgedrückten Modelle.

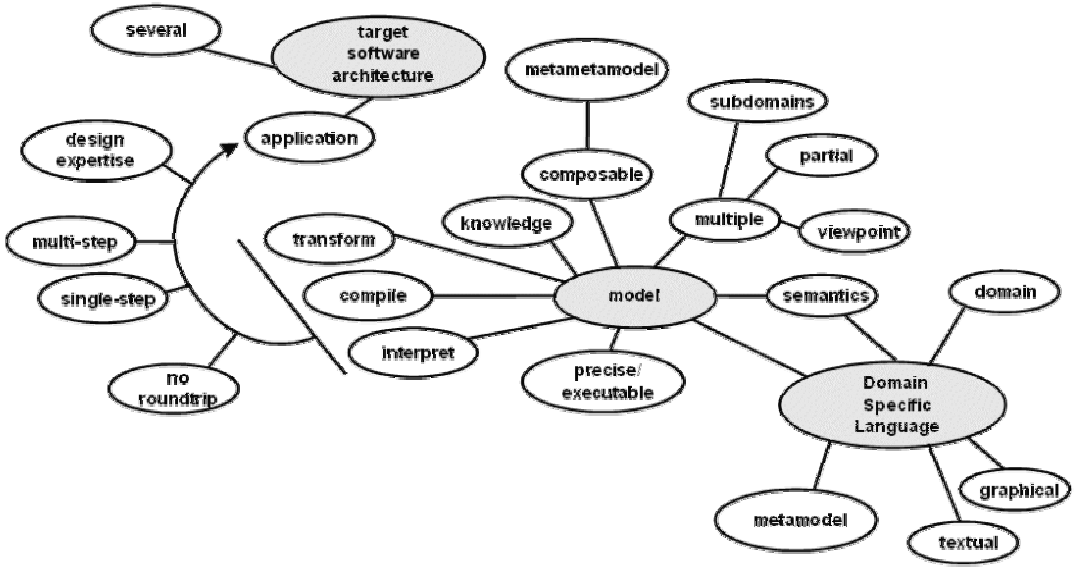

Abbildung 1: MDSD - Mindmap

Neben den Modellen und der DSL gibt es einen weiteren zentralen Bestandteil modellgetriebener Entwicklung: Die Plattform. Eine MDSD Plattform besteht aus wieder verwendbaren, domänenspezifische Komponenten, Frameworks und Utilities. Eine gute Plattform für MDSD besteht zunächst aus technischer Middleware wie CORBA, J2EE oder .NET. Darauf aufbauend enthält die Plattform aber auch eine Reihe fachlich spezifischer Frameworks, die fachliche Basisdienste im Rahmen einer bestimmten Domäne erbringen. DSL und Plattform stellen praktisch zwei Seiten derselben Medaille dar: die Plattform stellt Dienste zur Verfügung. Die DSL ermöglicht die einfache, effiziente und richtige Verwendung dieser Dienste.

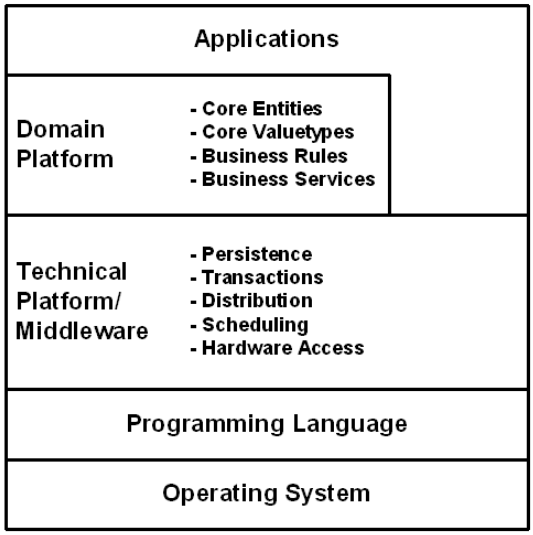

Abbildung 2: Typische MDSD Plattform

Die Überführung des Modells auf die auf der Zielplattform ausführbare Applikation wird üblicherweise mittels Transformationen oder, im Spezialfall der Codegenerierung, mittels Generierungstemplates erreicht. Diese Transformationsregeln enthalten damit also Wissen über die Verwendung der Plattform und sind, wie die Modelle und die Plattform, ein wertvolles, wieder verwendbares Asset.

Die Transformation vom Modell zur Zielapplikation kann in einem oder in mehreren Schritten geschehen, allerdings macht das automatisierte Rückführen von Änderungen am generierten Code ins Modell (Roundtripping) keinen Sinn, weil die Domänensemantik, die die Modelle enthalten aus den weniger semantisch reichen Quellcodes nicht automatisiert extrahierbar ist. Oft lässt sich eine komplexe Anwendung nicht mit *einer* DSL alleine sinnvoll abbilden. Man wird für verschiedene Aspekte des Systems verschiedene DSLs verwenden[2]. Da die Transformationen aber meist alle Aspekte gemeinsam berücksichtigen müssen, müssen die verschiedenen Modelle (und damit die DSLs) kombinierbar sein. Damit wird ein gemeinsames Metametamodell benötigt (also: eine gemeinsame Sprache, mit der die DSLs gebaut werden).

Aufgrund der oben erläuterten starken Fokussierung auf eine bestimmte Domäne und der Erstellung *domänenspezifischer* Tools sollte auch offensichtlich sein, dass dieser Ansatz nicht mit dem (glorreich gescheiterten) CASE Ansatz gemein hat (auch wenn das aus Sicht des einen oder anderen Toolherstellers sicherlich anders aussieht ☺).

[2] Man kann für bestimmte Teile des Systems wo sich die Erstellung einer DSL nicht lohnt, und wo sich die Anwendungslogik gut mittels einer 3GL ausdrücken lässt, natürlich auch in einer 3GL programmieren.

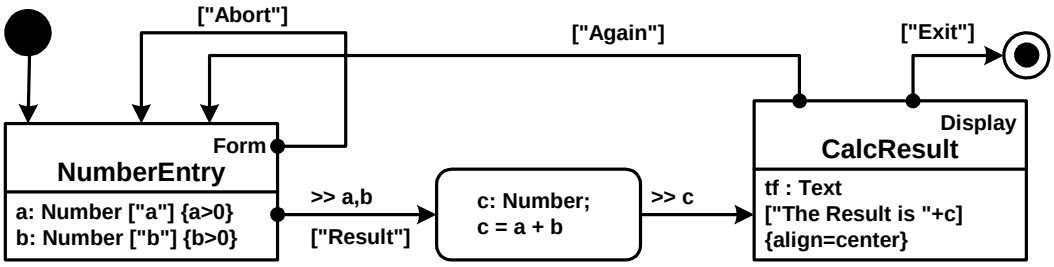

Abbildung 3: Beispiel einer DSL zu Modellierung von Java MIDP-Anwendungen, wie sie Beispielsweise auf Handys zum Einsatz kommen. Das Modell beschreibt wie zwei Zahlen mittels eines Formulars eingegeben werden; dann werden sie addiert und ausgegeben. Die Notation is

MDA als Spezialisierung von MDSD

Die MDA der OMG ist letztendlich eine Spezialisierung, eine bestimmte „Geschmacksrichtung" von modellgetriebener Softwareentwicklung. Die folgende Auflistung zeigt die Teile der obigen Mindmap, wo MDA konkretere Vorstellungen hat als der allgemeine MDSD-Ansatz.

- *DSL:* MDA-konforme DSLs sind alle Sprachen, die mittels der MOF (siehe Metametamodell) definiert werden. In der Praxis werden meist UML Profile verwendet also Anpassungen der UML mittels Stereotypen, Tagged Values, und Constraints.
- *Metametamodell:* Die OMG hat mit der Meta Object Facility (MOF) ein Metametamodell definiert das u.a. dazu dient, die UML und das Common Warehouse Metamodel (CWM) zu definieren. Das bedeutete, der Metamodell der UML und sowie das CWM sind mittels der MOF beschrieben.
- *Präzisierung von Modellen:* Um Modelle zu präzisieren und mit mehr Semantik anzureichern, bzw. um Verhalten besser spezialisieren zu können, können die OCL (Object Constraint Language) sowie seit UML 2.0, die Action Semantics verwendet werden.
- *Transformationen:* MDA-konforme Transformationen sollten auf Modelltransformationssprachen aufbauen, die derzeit im Rahmen des Query/Views/Transformations (QVT) RFPs standardisiert werden. Dazu sind bereits Einreichungen eingegangen. Eine Entscheidung bzgl. des Standards liegt noch nicht vor.
- *PIM, PSM,…:* MDA-Transformationen bilden immer PIMs (Platform Independent Models) auf PSMs (Platform Specific Models) ab. Die beiden Begriffe sind relativ zu einer Plattform, das PSM der einen Transformation ist das PIM der nachfolgenden.

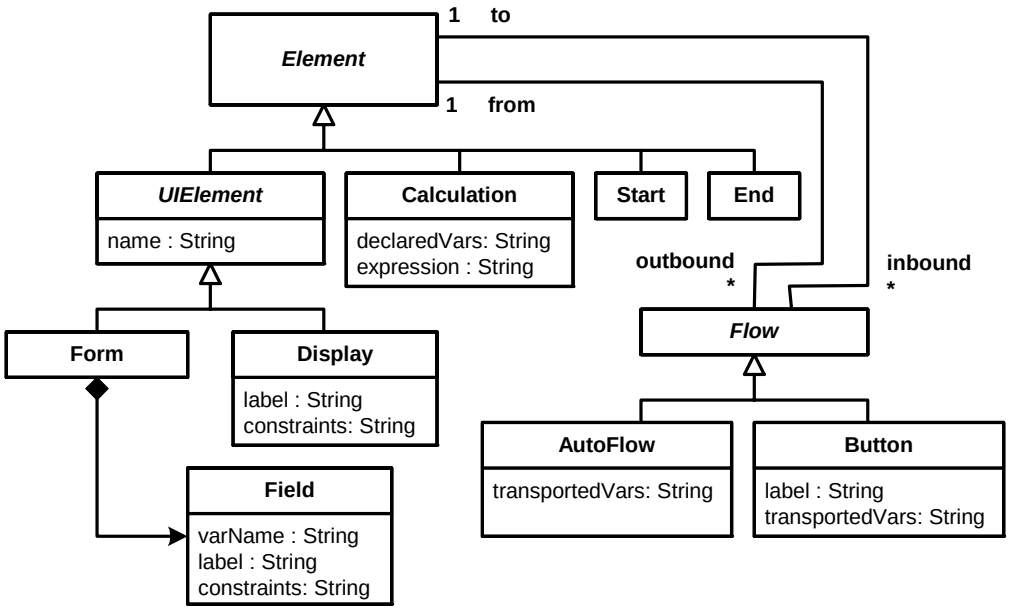

Abbildung 4: Vereinfachtes Metamodell der DSL für MIDP-Entwicklung

Warum MDSD?

Die Gründe, modellgetriebene Softwareentwicklung zu verwenden können vielfältig sein. MDA verfolgt vor allem das Ziel, die gleiche Anwendungslogik mittels verschiedener Transformationen auf verschiedenen Plattformen zur Ausführung zu bringen (weil ja leider ☺ nicht die ganze Welt OMG-Standards zur Softwareentwicklung einsetzt, sondern auch J2EE, .NET , …). Es gibt aber noch eine ganze Reihe weiterer Gründe für MDSD. Zum Beispiel der, dass die Entwicklung von Software zum Beispiel für J2EE sehr viele, sich oft wiederholende, fehleranfällige Schritte beinhaltet[3]. Diese können mittels MDSD sehr gut automatisiert werden.

Im Allgemeinen macht MDSD immer dann Sinn, wenn man im Kontext von Softwaresystemfamilien (SSF) arbeitet. Die Familienmitglieder zeichnen sich dadurch aus, dass sie eine Menge von Merkmalen gemeinsam haben, fachlich oder technisch. Oft setzen sie auf derselben technischen Infrastruktur auf. Damit eröffnet sich die Möglichkeit für Wiederverwendung, und zwar eben nicht nur von Komponenten und Frameworks, sondern auch von Modellen, Generatoren und Transformationen. Genau diese Wiederverwendungsmöglichkeiten sind es auch, die dafür sorgen, dass sich MDSD

[3] Man könnte nun natürlich über die Qualität der Plattform diskutieren… aber bei den Plattformen hat man in der Praxis oft keine Wahl.

rechnet. Es bedarf ja durchaus auch zusätzlichen Aufwands, die MDSD-Infrastruktur zu erstellen (dieser ist natürlich minimal, wenn man reine architekturzentrierte MDSD bzgl. einer technischen Mainstream-Plattform betreibt, und man die Tools kaufen bzw. downloaden kann; es macht also durchaus Sinn, typische J2EE Applikationen modellgetrieben zu Entwickeln).

Ein weiterer Vorteil ergibt sich aus der Verwendung von DSLs um die Anwendungslogik zu „programmieren“: Domänenexperten können leichter eingebunden werden. Es wird zwar weiterhin in den meisten Fällen eine Illusion bleiben, dass die „Fachler“ selbständig die Anwendungslogik implementieren. Es ist aber realistisch, dass sich ein Fachexperte mit einem Entwickler vor den Rechner setzt, und sie gemeinsam mittels der DSL „programmieren“. Durch die Nähe der DSL zur Domäne, und durch schnelle Umsetzung der Anforderungen in Code, wird es erheblich leichter, den Fachexperte direkt in die Entwicklung einzubinden, statt ihn hunderte oder tausende Seiten von Dokumenten schreiben zu lassen… Voraussetzung ist dabei, dass die DSL die Domäne tatsächlich gut repräsentiert. Zu einer solchen DSL kommt man nicht über Nacht. Fundierte Erfahrung in der Domäne und im der Definition von DSLs sind nötig. Ein iterativer Ansatz liegt hier nahe.

Die Rolle der Plattform und der Softwarearchitektur

MDSD hat einige sehr nützliche „Nebeneffekte“ im Zusammenhang mit Softwarearchitektur. Modelltransformationen bilden ja Konstrukte des Quell-Metamodells auf Elemente des Zielmetamodells ab. Damit man diese Abbildung prägnant formulieren kann, müssen diese beiden Metamodelle eine begrenzte Menge wohldefinierter Konzepte enthalten, es muss klar in Regeln fassbar sein, was worauf abgebildet werden soll. Im Falle der Transformation auf die Zielplattform bedeutet dies, dass die Plattform (bzw. deren Architektur) eine begrenzte Menge wohldefinierter Konzepte beinhalten muss. Genau dies ist eines der wesentlichen Merkmale guter Architektur. Insofern „erzwingt“ MDSD eine wohldefinierte Architektur und unterstützt die Entwickler, konform zu dieser (Produktlinien-)Architektur zu entwickeln. Regeln im Umgang mit der Architektur sind in den Transformationen codiert.

Abbildung 2 zeigt die typische Struktur einer Plattform wie sie im Zusammenhang mit MDSD oft Verwendung findet. Die Inhalte der einzelnen Schichten variieren von Domäne zu Domäne; die Schichtung ist aber praktisch überall gleich. Fragt sich nun, welche Teile davon generiert werden. In vielen Fällen sind es die Anwendungen ganz oben. Im Falle von verteilten eingebetteten Systemen, wo oft über 90% des Aufwands in die technische Plattform investiert wird, wird aber oft auch die Anwendung manuell erstellt, und die Middleware, sowie die Konfiguration des Betriebssystems generiert [MV03]. Dies hat zur Folge, dass das Image welches auf dem Gerät landet wirklich genau an die Anforderungen angepasst ist und keinen überflüssigen Code beinhaltet – Speicher ist in eingebetteten Systemen bekanntlich knapp.

MDSD und Aspektorientierung

In diesem Abschnitt möchte ich auf ein anderes wichtiges Thema, die Aspektorientierung (AO), eingehen und sie in Bezug zu MDSD setzen. Bei AO geht es darum, querliegende Belange eines Systems identifizieren und zu modularisieren um damit Änderungen dieses Belanges zu erleichtern, und, je nach Konfiguration, in das System einzubinden, oder auch nicht. Die populärste Form der Aspektorientierung ist die aspektorientierte *Programmierung* (das P in AOP), wo die querliegenden Belange auf Ebene des Quellcodes modularisiert werden. Je nach Tool werden sie statisch oder dynamisch mit dem Kern-Applikationscode verwoben. AOP kommt als Mittel, um Plattformcode und Applikationscode zu verbinden durchaus in Frage. Man kann aber im Zusammenhang mit MDSD den AO Gedanken auch anders umsetzen:

- Der Generator an sich ist schon eine Art AO-Werkzeug. Querliegende Belange des Generats sind im Generator oft sowieso schon lokalisiert (eben in der Template, die den betreffenden Code generiert). Mit einen einfachen *if* in der Template lässt sich die Struktur des Generats an allen Stellen die durch die betreffende Template generiert werden, ändern.
- Viele Aspekte (beispielsweise die die an Method Execution Pointcuts eingewoben werden) lassen sich auch durch die geschickte Anwendung von Design Patterns (Proxy, Adapter, Interceptor, Factory, …) abdecken. Die manuelle Implementierung dieser Ansätze ist aber lästig, weshalb man eben gerne auf AO*P* Werkzeuge zurückgreift. Mittels eines Generators ist es aber meist ein leichtes, zum Beispiel Proxies zu generieren, in die dann, gesteuert durch eine entsprechende Konfiguration, Interceptoren eingehängt werden können, die den Advice-Code enthalten.

Natürlich lassen sich mit den obigen Ansätzen nicht alle (wenn auch in der Praxis, recht viele) Aspekte abhandeln. In Fällen wo es nicht mit obigen mitteln klappt kann man den MDSD-Generator nutzen, um beispielsweise Aspekte zu generieren, oder zumindest die Pointcuts an denen die (manuell entwickelten) Advices in das Generat eingewoben werden sollen.

Ein weiteres interessantes Thema ist die Aspektorientierte Modellierung. Auch auf Modellebene macht es Sinn, die einzelnen Aspekte des Modells zu trennen und ggfs. mit verschiedenen DSLs abzudecken. Es ist dann Aufgaben des Generators, die Modelle der verschiedenen Aspekte zusammenzuführen, und Code zu generieren, der alle Aspekte gebührend berücksichtigt. Mit mächtigen und offenen Werkzeugen wie openArchitectureWare ist dieses Problem relativ einfach in den Griff zu bekommen.

Praktische Erfahrungen und Tools

Für viele Entwickler und Architekten stellt sich MDSD immer noch als nicht praxistaugliches Theoriegebilde dar – gut, dass man mittels XDoclet J2EE-Deployment-

Deskriptoren usw. generieren kann hat sich zwischenzeitlich herumgesprochen, aber die komplette Anwendungsentwicklung darauf umzustellen…? in ein paar Jahren vielleicht. Es ist auch durchaus sinnvoll, skeptisch zu sein. Wenn man einige der vollmundigen Versprechungen der OMG sowie einiger der Toolhersteller in ihrem Schlepptau betrachtet, sind tatsächlich noch nicht alle Ziele erreicht (siehe unten). Allerdings kann man bei pragmatischer Herangehensweise auch heute schon sehr produktiv mit MDSD arbeiten und von den Rationalisierungseffekten profitieren. Ich bin selbst an Projekten im Enterprise- und Embedded-Bereich beteiligt (bzw. beteiligt gewesen) und diese Projekte konnten die von MDSD gemachten Versprechungen zum großen Teil einlösen. Dabei setzen wir nicht auf große, kommerzielle Tools sondern zum großen Teil auf Open Source Werkzeuge, wie zum Beispiel openArchitectureWare [oAW]. Mittels dieses Tools können beliebige konkrete Syntaxen (UML, XML, textuell, …) verarbeitet werden, es können beliebige Domänen-Metamodelle sowie deren Constraints abgebildet werden und es lässt sich sehr gut in den Build-Prozess (z.B. in ant) einbinden. Natürlich können auch beliebige textuelle Zielformate (darunter eben auch, aber nicht ausschließlich, Programmiersprachen) erzeugt werden.

Die folgende Aufzählung soll einen knappen Eindruck vermitteln, wie und wo MDSD unter anderem eingesetzt wird:

- Im Enterprise-Bereich werden die verschiedensten J2EE Artefakte, das Deployment für den Applikationsserver sowie die Persistenz- und Workflowaspekte modellgetrieben entwickelt.
- Im EAI-Umfeld werden aus speziell dafür definierten DSLs SQL Statements generiert, die das Mapping, den Abgleich und die Migration von Daten zwischen verschiedenen Datenbanken erledigen.
- In einem verteilten Astronomieprojekt werden aus UML Modellen von Datenstrukturen die verschiedensten Artefakte generiert, unter anderem DDL Statements, XML Schema, sowie Datenzugriffsklassen in Java, C und Python.
- Im Bereich verteilter Embedded-Systeme wird die komplette Middleware sowie die Betriebssystemkonfiguration generiert, dies umfasst u.a. Buskommunikation, Scheduling, Hardware-Zugriff und Timing-Überwachung.

Im Rahmen dieses Abschnitts über Praxis und Tools sei mir noch eine Anmerkung zu Codegenerierung gestattet: Die Frage, ob Code verständlich, richtig eingerückt oder kommentiert ist hängt nicht davon ab, ob er generiert wurde oder nicht. Man sollte an generierten Code die gleichen Anforderungen stellen wie an handgeschriebenen, schließlich muss ich ihn im Fehlerfall debuggen und während der Entwicklung der Transformationen auf Korrektheit überprüfen. Dieses Ziel ist in der Praxis – notfalls mit einem Code Beautifier nach der Generierung – leicht zu erreichen. In Fällen in denen Teile des Systems manuell mittels einer 3GL implementiert werden, sollte man generierten und

manuell geschriebenen Code streng trennen, dies vereinfacht den Build sowie die Versionierung erheblich.

Auswirkungen auf Prozess und Organisation

Natürlich bedarf „richtige" Modellgetriebene Entwicklung eine passende Organisation und einen adäquaten Entwicklungsprozess. Zunächst gibt es zusätzliche Rollen im Projekt, beispielhaft seien drei genannt: Domänen-Analysten, Infrastrukturentwickler, und Anwendungsentwickler. Domänenanalysten analysieren die Anwendungsdomäne und definieren das fachliche Metamodell. Sie definieren Familienmitglieder, deren gemeinsame und unterschiedlichen Features (Featuremodellierung, siehe [CE00]). Der Infrastrukturentwickler erstellt die Plattform, wieder verwendbare Komponenten sowie vor allem die MDSD-Infrastruktur, also Generatoren bzw. deren Konfiguration, die DSLs sowie die entsprechenden Tools. Der Anwendungsentwickler verwendet DSLs und die restliche MDSD-Infrastruktur um Anwendungen in der Domäne zu entwickeln, zusammen mit den betreffenden Fachexperten.

Die zentralen Herausforderungen liegen im Umfeld der Koordination der beiden Zweige Infrastrukturentwicklung und Anwendungsentwicklung, weil die betreffenden Teams ja unterschiedliche Ziele haben (Infrastruktur-Teams: gute, wieder verwendbare Infrastruktur für die Softwaresystemfamilie, Anwendungsentwickler: möglichst schnell gute Produkte für den Kunden liefern). Insbesondere in der Anfangsphase wo die Infrastruktur parallel zu einer oder mehrerer Anwendungen entwickelt wird, ist dies potentiell schwierig. Der Schlüssel zur Lösung der Probleme liegt in den Techniken der agilen Softwareentwicklung. Kurze Iterationszyklen, am Ende derer jeweils die „neue Version" der Infrastruktur der Anwendungsentwicklung zur Verfügung gestellt wird sorgen für hohe Akzeptanz bei den Anwendungsentwicklern. Scope Trading, also das „Verhandeln" über wichtige Features und deren Priorisierung, sowohl fachliche, als auch Features der MDSD-Infrastruktur, zusammen mit den Kunden und den anderen Stakeholdern ist ein weiterer wichtiger Aspekt. Dabei spielen die fachlichen Teams die Rolle des Kunden bzgl. des Infrastrukturteams – ein „On-Site Customer" kann dabei helfen, dass die Infrastruktur tatsächlich die Probleme der Fachteams löst. Auch das Thema Testen ist wichtig, aus Platzgründen kann ich darauf hier aber nicht weiter eingehen und verweise auf [MV04, SVB04].

Einem recht weit verbreiteten Vorurteil möchte ich in diesem Zusammenhang noch entgegen treten. MDSD steht absolut nicht im Gegensatz zu agilen Konzepten. MDSD erfordert keinen Wasserfallprozess! Modelle sind bei MDSD kein Overhead sondern sie sind der Code, der im Kontext agiler Entwicklung „die alleinige Wahrheit" darstellt. Der Spruch „Code a little, test a little" wird bei MDSD sinngemäß genauso verwendet:

- Habe ich bereits eine MDSD-Infrastruktur für die Domäne, so wird aus daraus „model a little, test a little".

- Wenn ich parallel zur Entwicklung der Anwendung die Infrastruktur erstelle, erweitert sich das vorgehen zu „model a little, build a little transformation, test a little“.

Die Infrastruktur und die Anwendungen werden also iterativ entwickelt. Wenn man eine vernünftige DSL zur Verfügung hat wird auch die Einbindung des Kunden leichter, da er die Modelle eher versteht, als 3GL Code. Man kommt schneller zu ausführbarem Code – der Großteil wird ja generiert, und man kann auch schneller auf Veränderungen in den Anforderungen reagieren, weil man idealerweise nur an einer Stelle – im betreffenden Modell, oder bei querliegenden Belangen, an den Transformationen – ändern muss, und nicht an den verschiedensten Stellen im Code (GUI, Business Layer, Persistence Layer).

Übergang zu MDSD

Wie oben bereits erwähnt, bedeutet der Übergang zu MDSD erhebliche Umstellungen in den Techniken, der Organisation und im Prozess. Man wird daher sicherlich nicht von einem Tag auf den anderen komplett auf MDSD umstellen. Es macht daher wie immer bei der Einführung von Neuerungen Sinn, zunächst einen Versuchballon steigen zu lassen, um dabei die nötigen Techniken und Vorgehensweisen zu erlernen. Dabei ist es wichtig, dass es sich hierbei nicht um ein Trivialbeispiel handelt, weil dadurch weder die Vorteile, noch die Herausforderungen von MDSD zu Tage treten.

Besonders anspruchsvoll ist die Definition des Metamodells und der DSL. Vor allem für fachlich-spezifische DSLs bedarf es dafür einiger Erfahrung und Fingerspitzengefühl. Die dazu nötigen Techniken (Definition einer Sprache, Transformationen, etc.) sind in vielen Entwicklungsorganisationen nicht besonders weit verbreitet. Vor allem für Nicht-Informatiker sind diese Themen teils Bücher mit sieben Siegeln, und es bedarf einiges an Lernaufwand um hier wirklich fit zu werden (ich kann das beurteilen, bin selbst Ingenieur).

Es ist ratsam, sich zunächst auf Architekturzentrierte MDSD zu fokussieren. Dabei werden die Konzepte der Zielarchitektur mehr oder weniger direkt im Metamodell abgebildet (im Falle von J2EE z.B. Entity Beans, Session Beans, Servlets, Queues, …). Als konkrete Syntax eignet sich oft UML zusammen mit einem architekturzentrierten Profil. Im Laufe der Zeit wenn mehr Erfahrungen mit der Domäne, Metamodellierung, DSL-Definition sowie der Toolkette vorliegen, kann diese DSL immer mehr fachliche Konzepte aufnehmen. Man wird dabei auch in der Regel nicht mehr nur mit UML+Profil auskommen, sondern zum Beispiel auch textuelle Sprachen einbinden. Die Verknüpfung der verschiedenen Modellaspekte, die mittels verschiedener DSLs spezifiziert werden kann man mit Tools wie openArchitectureWare [oAW] gut in den Griff bekommen.

Um den Einstieg in die Techniken und Tools zu bekommen, macht es Sinn, zunächst einen Prototyp manuell – also ohne Modelle und Generierung – zu entwickeln. Aus diesem Prototyp lassen sich dann Transformationen, DSL, Metamodelle, etc. extrahieren. Damit

kann dann der identische Prototyp modellgetrieben entwickelt werden. Durch Ausführen der für den Prototyp erstellten Unit-Tests auf dem generierten Code kann man verifizieren, dass die MDSD-Infrastruktur (zumindest für diesen Fall) funktioniert. Nach erfolgreicher Implementierung dieses Prototyps kann in den oben beschriebenen, zweigleisigen iterativen Entwicklungsprozess eingestiegen werden, im Rahmen dessen Infrastruktur und Anwendungsprojekte parallel weiterentwickelt werden.

Wann lohnt sich MDSD?

Wie mit allen Neuerungen, stellt sich letztendlich die Frage, wo der wirtschaftliche Nutzen liegt. Ohne klare Vorstellungen, wann, wie und wie viel Geld eine Neuerung bringt bzw. einspart, scheitert deren Einführung spätestens am Management. Einige Einspareffekte sind relativ offensichtlich: Wenn große Mengen Code die normalerweise von Hand geschrieben werden müssten nun automatisch erstellt werden, so spart dies offensichtlich Implementierungsaufwand. Andere Dinge sind weniger offensichtlich, und auch weniger gut in Zahlen zu fassen. Zum Beispiel steigt die Qualität der Software, durch die erzwungene Architekturstrukturierung (s.o.), und es steigt die Codequalität, weil der Generator keine Flüchtigkeitsfehler macht: wenn der Generator Fehler macht sind die Transformationsvorschriften falsch, und der Generator macht den Fehler immer – was das Finden des Fehler erheblich vereinfacht! Die Formalisierung des Domänenwissens in den Modellen sowie die codifizierten Regeln im Umgang mit der Plattform in den Templates stellen auch einen Wert an sich dar. MDSD erlaubt auch die gezieltere Nutzung der Fähigkeiten und Vorlieben der Entwickler, die sich Entweder auf Technik (technische Plattform, Transformationen) oder die Domäne (Metamodell, DSL, fachliche Plattform) konzentrieren können.

Natürlich ist auch MDSD kein „free lunch“. Man muss den oben erwähnten Benefits die Kosten der Einführung von MDSD gegenüberstellen. Dazu zählt insbesondere, dass völlig neue Techniken zu erlernen sind, vor allem im Umfeld von Metamodellierung, Definition der DSL, Implementierung der Transformationen sowie Bereitstellung der Plattform. Genauso wie es in der Praxis nicht gerade einfach ist, gute, wieder verwendbare und praxistaugliche (!) Frameworks zu entwickeln und weiter zu pflegen, sind auch diese Tätigkeiten nicht trivial. Man braucht dazu den einen oder anderen fähigen Entwickler.

Die Frage, ob der Nutzen die Kosten rechtfertigen, lässt sich nicht im Allgemeinen beantworten. Es bietet sich aber ein Vergleich an. Die Entwicklung einer „Softwareproduktionsinfrastruktur“ – MDSD ist ja nichts anderes – rechnet sich vor allem, wenn damit möglichst viel produziert wird. Im Produzierenden Gewerbe werden auch keine automatischen Montagestraßen für Einzelstücke erstellt. Hier kommt wieder der Gedanke der Softwaresystemfamilie zum tragen. Je mehr Produkte mit einer MDSD Infrastruktur erstellt werden (oder anders ausgedrückt: je größer die Familie) desto eher lassen sich signifikante Einsparungen erzielen.

Ein kleiner Wehrmutstropfen bleibt aber: Die Kosten fallen zum großen Teil am Anfang an, die Benefits zahlen sich oft erst mittelfristig aus. Insofern benötigt man einen eher etwas längeren Atem. In unserer kurzlebigen IT-Welt ist das oft nicht einfach zu „verkaufen". Mehr zu diesem Thema in [JB03].

Offene Punkte – Herausforderungen

Auch wenn man mit MDSD heutzutage bereits sehr produktiv arbeiten kann, sind noch lange nicht alle Herausforderungen zufrieden stellend gelöst – dies gilt, wie oben erläutert, vor allem für MDA. Beispielsweise fehlen Werkzeuge, große Mengen von Modellen effizient zu verwalten (versuchen sie mal, mit CVS XMI-Files semantisch korrekt zu mergen, oder ein UML Modell mit 1000 Klassen zu überblicken). Die größte Herausforderung liegt allerdings im Umfeld der DSLs. Es ist ein zumindest aus Praktikersicht völlig offenes Thema, wie man effizient DSLs bauen kann [ERLS]. Man kann Sprachbausteine nicht ohne weiteres wieder verwenden, sie einfach zusammensetzen sodass sie syntaktisch und semantisch korrekt zusammenarbeiten. Auch die Toolunterstützung ist oft nicht sehr gut. Die Entwicklung von IDEs für die eigene DSL ist immer noch sehr aufwendig, auch wenn Eclipse eine gute Basis dafür zur Verfügung stellt. Auch mit der Anpassung vorhandener UML-Tools an die eigene Domäne, also die Berücksichtigung der durch ein Profil definierten Semantik bei der Modellierung ist es nicht besonders weit her. Es ist hier zwar in absehbarer Zeit mit einer Verbesserung der Situation zu rechnen, aber wir sind noch nicht da! Last but not Least sei noch darauf hingewiesen, dass Modelltransformations- und Templatesprachen praktisch alle proprietär sind. Sie binden sich also immer an ein bestimmtes Tool (im Idealfall an ein frei verfügbares!). Und ob der MDA-QVT RFP wirklich praktisch nutzbare Ergebnisse bringt muss noch abgewartet werden. Außerdem fehlen – im Allgemeinen – Debugger auf Modellebene, komfortable IDEs die es erlauben, verschiedene DSLs zu kombinieren, Modell-Refactoring-Tools, und, und, und. Das hier verfolgte Ziel entspricht praktisch dem des Intentional Programming.

Weitere Herausforderungen liegen in der Mentalität vieler Entwickler die Modelle als „Overhead" verstehen, und lieber direkt mit Code arbeiten. Es bedarf oft einiges an Überzeugungsarbeit, bis eingefleischte Entwickler Modelle als „Code" betrachten. Eine weitere Herausforderung liegt in der Tatsache, dass spezifisch für eine Domäne entwickelt: um eine MDSD-Infrastruktur zu entwickeln, müssen sich die betreffenen Entwickler im Detail mit der Domäne auseinandersetzen. Für viele Entwickler klingt das nicht sehr verlockend, da sie sich lieber mit der neuesten Version irgendeiner technischen Infrastruktur (J2EE, .NET, …) beschäftigen wollen – ist besser für den Lebenslauf oder für viele Entwickler schlicht spannender. Wer allerdings bereits die Erfahrung machen durfte, eine MDSD-Infrastruktur für eine bestimmte Domäne zu bauen, wird bestätigen, dass dies eine sehr interessante Aufgabe darstellt, und richtig Spass machen kann.

Fazit

Was bedeutet dies nun alles für die konkrete Softwareentwicklung heute? Wir sind von den Zielen (vor allem wie sie die MDA vorstellt) sicherlich noch weit weg. Dennoch werden bereits heute verschiedenste Projekte mit Mitteln der MDSD entwickelt. Die verfügbaren Tools sind aus den Kinderschuhen heraus und bereit für den Praxiseinsatz. Die Änderungen am Prozess und der Organisation können schrittweise eingeführt werden; Migrationsstrategien liegen vor. Best Practices [VB04] und Erfahrungen sind verfügbar. Nutzen Sie die Chance ☺

PS: Vielen Dank an Krzysztof Czarnecki, Martin Lippert, Alexander Schmid, Christa Schwanninger und Eberhardt Wolff. Klasse, dass man von Euch auch innerhalb sehr kurzer Zeit sehr hilfreiches Feedback bekommt! Danke!

Referenzen

[CE00] Czarnecki, Eisenecker, Generative Programming, Addison-Wesley 2000

[ERLS] ECOOP `04 Workshop on Evolution and Reuse of Language Specifications for DSLs, http://www.ifi.uio.no/ecoop2004/workshops.html#WS23

[GS04] Greenfield, Short, *Software Factories*, Wiley, 2004