Modellgetriebene Softwareentwicklung

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

Modellgetriebene Entwicklung (MDSD[1]) und MDA sind in aller Munde. Dieser Artikel stellt MDSD-Kernkonzepte vor und zeigt, wie sich modellgetriebene Softwareentwicklung im Allgemeinen von MDA (als Spezialisierung von MDSD) unterscheidet. Des Weiteren möchte der Autor Konsequenzen für den Entwicklungsprozess aufzeigen sowie darauf eingehen. Aus Platzgründen kann nicht auf alle Details und Aspekte eingegangen werden. Weitere Details finden sich z.B. unter [SV05].

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. 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[2]. 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

[1] engl. Model-Driven Software Development

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

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

Begriffsdefinition und Wichtige Konzepte

Im Zentrum von MDSD steht – naheliegenderweise – das Modell. Dieses beschreibt Applikationsfunktionalität auf dem Abstraktionsniveau der Anwendungsdomäne, ausgedrückt mit einer Domänenspezifischen Sprache (siehe Abb. 1). 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 abstrakte Syntax, auch als Metamodell bezeichnet (siehe Abb. 2). Das Metamodell definiert die Modellelemente, mit denen die Modelle der betreffenden Domäne erstellt werden können. Die Semantik der Modelle 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.

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.

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.

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.

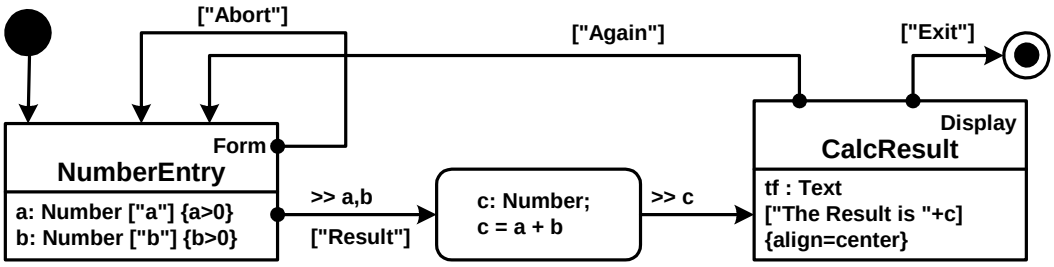

Abbildung 1: Beispiel einer DSL zu Modellierung von Java MIDP-Anwendungen, wie sie Beispielsweise auf Handys zum Einsatz kommen. Das Modell beschreibt wie zwei Zahlen mittels eines Formulars eingegeben werden; dann werden sie addiert und ausgegeben. Die Notation is

MDA als Spezialisierung von MDSD

Die MDA der OMG [DF03] ist letztendlich eine Spezialisierung, eine bestimmte „Geschmacksrichtung" von modellgetriebener Softwareentwicklung. Die folgende Auflistung zeigt die Aspekte 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.
- *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.

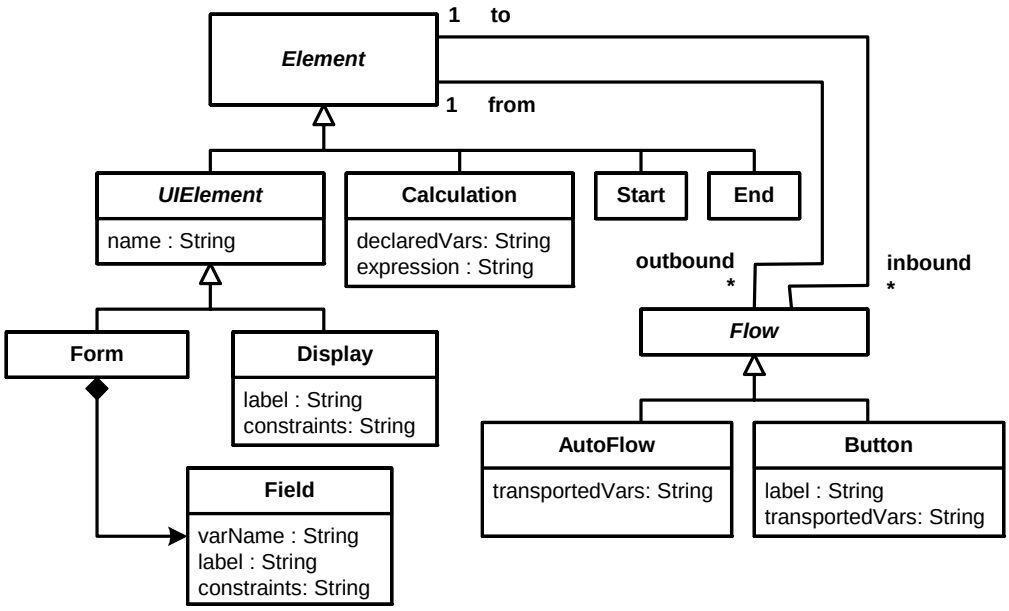

Abbildung 2: Vereinfachtes Metamodell der DSL für MIDP-Entwicklung

Warum MDSD?

MDA verfolgt vor allem das Ziel, die gleiche Anwendungslogik mittels verschiedener Transformationen auf verschiedenen Plattformen zur Ausführung zu bringen. 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. 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 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.

Praktische Erfahrungen und Tools

Für viele Entwickler und Architekten stellt sich MDSD immer noch als nicht praxistaugliches Theoriegebilde dar. Wenn man einige der vollmundigen Versprechungen der OMG sowie einiger der Toolhersteller in ihrem Schlepptau betrachtet, sind tatsächlich noch nicht alle Ziele erreicht. Allerdings kann man bei pragmatischer Herangehensweise auch heute schon sehr produktiv mit MDSD arbeiten und von den Rationalisierungseffekten profitieren. Der Autor ist 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.

MDSD und Datenbanken

Aus Sicht des Softwarearchitekten sind Datenbanken eine – zugegebenermaßen sehr wichtige – Implementierungstechnologie, vergleichbar mit J2EE oder .NET. Sie stellen also im Rahmen von MDSD das *Ziel* dar, gegen das ggfs. Code generiert wird. Aus domänenspezifischen Modellen müssen also üblicherweise Datenbankzugriffsschichten oder DDL-Skripte erstellt werden. In diesem Fall tritt die Datenbank im Rahmen der DSL nicht auf. In manchen Fällen lassen sich aber die nötigen Artefakte nicht vollautomatisch erstellen, weitere, jetzt datenbank-spezifische, Informationen sind für den Generator notwendig. In diesem Fall stellt die Datenbank einen Modell-Aspekt dar (siehe AO-Modelling Pattern in [MV05]) der mit einer entsprechenden DSL beschrieben werden muss. Sieht man die Datenbank selbst als *Domäne* für MDSD an, dann gibt es dafür natürlich bereits DSLs und Tools – namentlich ER-Modellierung und die entsprechenden Werkzeuge. Gegebenenfalls ist es dann nötig, diese Notation (bzw. die dafür verwendeten Tools) mit den Tools für die anderen Modellaspekte zu integrieren. Dies ist ja auch der Grund, warum man immer mehr Datenbankmodellierungsfunktionalitäten in UML Werkzeugen findet.

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 [SV05] und Erfahrungen sind verfügbar. Nutzen Sie die Chance ☺

Referenzen

[DF03] David Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley, 2003

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

[MV05] Markus Völter, Models and Aspects – Handling Cross-Cutting Concerns in MDSD, http://www.voelter.de/data/pub/ModelsAndAspects.pdf

[oAW] openarchitectureware.org, openArchitectureWare Generator Framework, http://www.openarchitectureware.org

[SV05] Thomas Stahl, Markus Völter, *Modellgetriebene Softewareentwicklung – Technik, Engineering, Management*, dPunkt, 2005