

Sprachen, Modelle, Fabriken

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

Rund um die Modellgetriebene Softwareentwicklung etablieren sich eine ganze Reihe von Techniken und Vorgehensweisen die unter verschiedenen Namen bekannt sind. Dazu zählen Software Factories, Model-Integrated Computing, Language-Oriented Programming sowie die beiden alteingesessenen Vertreter, MDA und Generative Programmierung. Dieser Artikel soll die verschiedenen Ansätze zueinander in Bezug setzen.

Einführung und Begriffsbildung

Bei Modellgetriebener Softwareentwicklung (Model-Driven Software Development, MDSD, siehe auch [SV05]) geht es im Kern darum, die Softwareentwicklung durch größere Anlehnung an Konzepte der Problemdomäne effizienter zu gestalten. Dazu werden domänenspezifische (Modellierungs-)Sprachen eingesetzt mit Hilfe derer Entwickler Strukturen und Verhaltensweisen von Systemen der betreffenden Domäne effizient beschreiben können. Mittels Transformationen werden diese Modelle dann in ausführbaren Code überführt. Dieser Artikel soll die verschiedenen „Geschmacksrichtungen“ dieses Ansatzes die derzeit am Markt verfügbar sind aufzeigen.

Im Folgenden wollen wir einige zentrale Begriffe einführen, die später wichtig sein werden. Im Mittelpunkt des MDSD Ansatzes steht die *Domäne*. Um ausdrucksstarke, semantisch eindeutige und damit transformierbar DSLs erstellen zu können, muss der Gültigkeitsbereich der DSL – die Domäne – ausreichend stark eingeschränkt werden. Ein tiefes Verständnis der Domäne ist Voraussetzung für eine erfolgreiche Domänenanalyse, diese wiederum ist Voraussetzung für den erfolgreichen Bau einer DSL. Die Konzepte der Domäne, sowie ihre Zusammenhänge werden dann mittels eines *Metamodells* formal definiert. Die Konzepte im Metamodell tauchen in instantiiertes Form in den Modellen wieder auf, die konkretes Sachverhalte in der Domäne ausdrücken. Jedes Element eines Modells ist eine Instanz eines Elements des betreffenden Metamodells. Für die Compilerbauer unter den Lesern: das Metamodell einer DSL stellt deren abstrakte Syntax dar und ist damit – wie auch im Compilerbau – die Basis für Transformationen. Daneben benötigt man aber auch eine für die Entwickler bzw. Domänenexperten geeignete konkrete Syntax. Diese kann grafisch oder textuell sein; im einfachsten Fall können UML Profile oder ein XML Schema verwendet werden. Die Erstellung von guten, die Syntax aktiv unterstützende Editoren ist ein durchaus noch nicht abschließend gelöstes Problem – auch wenn es vielversprechende Ansätze dafür gibt (insbesondere das Eclipse Graphical Modelling Framework [GMF]). Man beachte dass man ein und dieselbe abstrakte Syntax mit verschiedenen konkreten Syntaxformen darstellen kann. Der Informationsgehalt ist derselbe, die Darstellungsformen mögen für verschiedene Anwendungsfälle unterschiedlich gut geeignet sein.

Neben dem Metamodell spielt auch das Metametamodell eine wichtige Rolle; es ist das Metamodell, mit Hilfe dessen Metamodelle definiert werden. Üblicherweise ist bei MDSD Werkzeugen Wissen über das Metametamodell „fest verdrahtet“, weswegen es eine gute Idee ist, das Metametamodell zu standardisieren (solche Standards sind bspw. die MOF der OMG oder das Ecore Metametamodell des Eclipse Modelling Frameworks [EMF]).

Neben der DSL spielt bei MDSD auch die Plattform eine wichtige Rolle. Variabilität lässt in der Softwareentwicklung bekanntermaßen mit verschiedensten Mitteln abbilden [MV06a]. Modellierung und Codegenerierung ist eine davon. Allerdings können auch Frameworks, Bibliotheken, Aspekte, Middleware oder andere Artefakte bestimmte Arten von Variabilität abbilden. Desweiteren haben natürlich die unterschiedlichen Systeme im Rahmen einer Domäne auch eine ganze Reihe Dinge gemeinsam. Daher macht es Sinn, eine Plattform zu erstellen, die alle wieder verwendbaren Artefakte einer Domäne sammelt. Dazu gehören wie gesagt typischerweise Bibliotheken, Frameworks, Middleware, Komponenten, Aspekte oder Interpreter. Die Transformatoren erstellen also Artefakte, die die Plattform direkt verarbeiten kann. Dies können neben Code also auch Konfigurationsfiles oder zu interpretierende Spezifikationen sein.

Transformationen lassen sich in zwei Unterarten unterteilen: Modell-zu-Modell- (M2M-) Transformationen sowie Modell-zu-Code (M2C-) Transformationen. Erstere bilden ein oder mehrere Eingabemodelle auf ein oder mehrere Zielmodelle ab. Die nötigen Transformationsregeln werden also letztendlich als Abbildungsvorschriften zwischen dem Eingabe- und dem Ausgabemetamodell definiert. Modell-zu-Code Transformationen (auch manchmal als Modell-zu-Text oder Modell-zu-Plattform bezeichnet) bilden Modelle auf textuellen Output ab, der direkt von der Plattform verarbeitet werden kann. Auch hier spielt das Metamodell des Eingabemodells eine zentrale Rolle: die Transformationen werden daran verankert. Das Zielmetamodell (also die Abstrakte Syntax der Zielsprache) tritt in aller Regel nicht in Erscheinung, da man meist mittels Templatesprachen direkt den textuellen Output erzeugt. Man beachte, dass die Transformationen eine Formalisierung der Best Practices im Umgang mit der Zielplattform darstellen.

Als letzten zu definierenden Begriff möchte ich die Domänenarchitektur einführen. Diese umfasst eine Plattform, eine oder mehrere DSL, die nötigen Transformationen sowie alle nötigen Werkzeuge – also alles, um für eine bestimmte Domäne modellgetrieben zu arbeiten.

Architekturzentrierte MDSD

Architekturzentrierte, modellgetriebene Softwareentwicklung (AC-MDSD = Architecture Centric MDSD) ist eine praktisch orientierte, auf die Automatisierung von Softwarearchitektur fokussierte Spezialisierung von MDSD, die konzeptionell auch einen gemeinsamen Schnitt mit MDA besitzt (siehe unter anderem MV06b)). Diese Variante hat folgende Eckpfeiler:

- Die Domäne ist architektonisch motiviert – z.B. „Architektur für Business Software“ oder Komponenteninfrastrukturen für Embedded-Systeme.
- Das Metamodell der DSL enthält dementsprechend möglichst abstrakte Architekturkonzepte.
- Es gibt in der Außensicht nur einstufige Modell-zu-Code-Transformationen. Diese können allerdings intern zwecks Modularisierung als Sequenz mehrerer (M2M-) Transformationen strukturiert sein.
- Die Modell-zu-Plattform-Transformation liegt typischerweise in Form von Templates vor, die eine große Ähnlichkeit zum Generat aufweisen und daher bequem aus einer Referenzimplementierung extrahiert werden können.
- Die Transformation hat nicht den Anspruch, die gesamte Anwendung zu erzeugen, sondern nur einen Implementierungsrahmen, in den dann die manuell in der Zielsprache implementierte Anwendungslogik integriert wird. Generierter und nicht-generierter Code werden mittels geeigneter Design-Patterns integriert.

AC-MDSD versucht, die Lücke zwischen Modell und Zielplattform dadurch zu verkleinern, dass man eine Plattform schafft, die die wichtigsten Architektur-Konzepte bereits zur Verfügung stellt. Man kann dann das für die Anwendungsmodellierung verwendete Metamodell sehr stark auf diese Zielarchitektur/Plattform ausrichten – daher der Name. Als Folge dessen kann man oft auch gut mit einem Generierungsschritt auskommen und braucht nicht zwangsläufig verschiedene aufeinander folgende Transformationsschritte.

Bekannte Vertreter von Tools für AC-MDSD sind bspw. im Open Source Bereich zu finden. Dabei sind openArchitectureWare [oAW] und AndroMDA [AND] sicherlich die bekanntesten Vertreter.

Model-Driven Architecture

MDA (siehe [MDA]) ist bezüglich der oben eingeführten Begriffe eine Spezialisierung von MDSD. Der Fokus der MDA ist ursprünglich die Möglichkeit, aus einem Modell Implementierungen für verschiedene Plattformen generieren zu können, und damit – ganz im Sinne der OMG – die Plattformneutralität von Anwendungssoftware zu stärken. MDA spezialisiert MDSD in folgender Hinsicht:

- MDA sieht vor, dass die DSLs auf Basis der MOF definiert werden. Es sind also beliebige Notationen und Metamodelle möglich, solange sie mit Hilfe des OMG-Metametamodells definiert wurden.
- In der Praxis regt MDA die Verwendung von UML-Profilen als konkrete Syntax für eine DSL an. Damit ist die DSL im Kern dann auf UML festgelegt. Die statische Semantik wird dementsprechend mit OCL-Ausdrücken spezifiziert.

- Es werden verschiedene Sichtweisen auf formale Modelle definiert: Ein fachliches Modell kann relativ zu einer Plattform spezifisch (PSM) oder unspezifisch (PIM) sein. Die MDA legt mehrschrittige Transformationen zwischen Modellen nahe, verbietet aber auch eine direkte PIM-zu-Code-Transformation nicht.
- Für Modell-zu-Modell Transformationen legt die MDA QVT nahe. QVT steht für Query/Views/Transformations (QVT) und ist ein (fast fertiger) OMG Standard.
- Die Action Semantics bietet die Möglichkeit, in abstrakter Form Algorithmen zu spezifizieren. Unter Verwendung einer (toolspezifischen, weil nicht standardisierten) konkreten Syntax kann man damit programmieren wie in anderen Programmiersprachen auch, allerdings unter Einbeziehung der Informationen im statischen Modell.

Ausführbare UML-Modelle (*Executable UML*) stellen eine Besonderheit und für viele MDA-Vertreter eine Zielvorstellung von MDA dar: Sie sind entweder mehr oder weniger direkt auf einer entsprechend mächtigen und generischen Plattform ausführbar – d.h. sie werden von einer virtuellen UML-Maschine interpretiert oder mittels Transformationen vollständig kompiliert, so dass sie auf einer „niedrigeren“ Plattform ausführbar werden. Im Gegensatz zu einer fachlich fokussierten Domäne haben wir es in diesem Falle also mit einer „Domäne“ zu tun, die der Ausdrucksmächtigkeit einer Programmiersprache entspricht. Ein UML-Profil ist daher nicht notwendig. Offensichtlich ist dadurch aber auch die Distanz zu einem fachlichen Problemraum deutlich größer. Ein Beispiel für ein entsprechende Tooling findet sich bei KennedyCarter [iUML].

Generative Programmierung

Der Begriff Generative Programmierung (GP) ist bereits seit einigen Jahren geläufig. Bekannt wurde der Begriff vor allem durch das „Katzenbuch“ von Krzysztof Czarnecki und Ulrich Eisenecker (offiziell heißt das Buch *Generative Programming* [EC00]). Die treibenden Faktoren von GP sind:

- Anlehnung an industrielle Fertigungsparadigmen wie sie z.B. im Automobilbau zu finden sind. Die Metapher einer Produktionsstraße und eines Bestellscheins wird gerne verwendet.
- Generative Programmierung hat den Anspruch, komplette Produkte (z.B. Anwendungen) aus Spezifikationen zu erzeugen – 100% Automation.
- GP betont die Erstellung („Konfiguration“) von Anwendungen aus vordefinierten, atomaren Komponenten.
- Herstellung bzgl. bestimmter Aspekte (z.B. Performance und Codegröße) optimierter Produkte

Ziel ist es also, aus einem Modell einer (formalen) Anforderungsspezifikation möglichst automatisiert genau passende und optimierte Produkte zu erstellen. Auch Generative Programmierung ist bezüglich der Ontologie eine Spezialform von MDSO mit folgenden Eigenschaften:

- Bei GP spielt der Gedanke der Software-Systemfamilie eine zentrale Rolle.
- Traditionell ist der Gedanke der (UML-)Modellierung weniger stark vorhanden. Viel mehr definiert man basierend auf der Domänenanalyse eine oft textuelle DSL, mit der Produkte der Familie beschreibbar werden. Oft werden Feature-Modelle als Basis für die DSL bzw. das Metamodell verwendet. Die konkrete Ausprägung eines Feature-Modells, also die Spezifikation des Produkts, spielt dann die Rolle des formalen Modells im MDSO-Sinne. Allerdings ist dies keine zwingende Vorgabe. Prinzipiell kann auch in GP jede Art von Metamodell bzw. DSL verwendet werden.
- Die Domäne wird auch als Problemraum bezeichnet, die Plattform und die Komponenten, aus denen das „Produkt“ zusammengesetzt wird, als Lösungsraum.
- Das Konfigurationswissen ist in einen Generator gegossen, der ähnlich wie bei AC-MDSO eine (meist einschrittige) Modell-zu-Code-Transformation durchführt. Die statische Semantik (Erkennung ungültiger Produktkonfigurationen) wird ebenfalls durch das Konfigurationswissen realisiert.
- Die Plattform besteht typischerweise aus maximal kombinierbaren und minimal redundanten Komponenten, welche letztlich die Variabilität der DSL, d.h. deren Ausdrucksmöglichkeiten abbilden.

Auch wenn es durch die Definition von GP nicht zwangsläufig so vorgegeben ist, trifft man auf Grund der Betonung von „optimiert“ oftmals auf statische Generierungstechniken. Die Konfiguration von Frameworks oder das Erstellen einer virtuellen Maschine ist eher unüblich. Trotzdem ist es wichtig zu verstehen, dass GP nicht einfach nur Codegenerierung bedeutet. Auch ist GP nicht gleichzusetzen mit C++ Template-Metaprogrammierung, welches lediglich eine Implementierungstechnologie für GP darstellt.

Software Factories

Der Begriff Software Factories wurde von Microsoft geprägt und wird in dem gleichnamigen Buch von Greenfield und Short ausführlich beschrieben [GS04]. Im Grunde ist eine Software Factory eine für eine bestimmte Domäne angepasste IDE – typischerweise dann eben Visual Studio – die die Entwicklung von Software für die betreffende Domäne stark vereinfacht. Dabei kommen Wizards, Patterns, Frameworks, Vorlagen sowie DSLs und dafür passende Editoren zum Einsatz. Manche sagen, Software Factories sei “Product

Line Engineering" à la Microsoft. Der Bezug zu PLE ist dabei sicher richtig. Microsoft legt aber viel Wert darauf zu betonen, dass es sich bei dem Ansatz nicht um eine reine Microsoft Sache handelt – auch wenn das in der Praxis anders aussieht!

Da SF den gesamten PLE Prozess betrachten, machen modellgetriebene Ansätze nur einen Teil des gesamten Ansatzes aus – wenn auch einen wichtigen. Wir werden daher zunächst den allgemeinen Ansatz betrachten und dann spezifisch den Bezug zu MDSD.

Das vielleicht wichtigste Konzept von Software Factories ist das Schema. Es beschreibt die Viewpoints die sinnvoll und notwendig sind um ein System der betreffenden Domäne zu entwickeln. In einem Enterprise-System sind dies beispielsweise: Präsentation, incl. Formularabfolge und -layout, Komponententruktur und Geschäftsdatenmodell, Persistenzmapping und Deployment.

Für jeden dieser Viewpoints identifiziert das Schema die relevanten Artefakte sowie den effektivsten Weg um diese zu erstellen. Dazu zählen die manuelle Programmierung, die Anwendung bestimmter Patterns, die Benutzung und Konfiguration bestimmter Frameworks oder die Erstellung und spätere Verwendung einer DSL (sowie nachfolgende Generierung). Das Schema stellt also ein konzeptionelles Framework dar, um die verschiedenen Gesichtspunkte eines Systems zu identifizieren und zu verwalten. Dabei spielt der Abstraktionslevel des Gesichtspunkts, sowie seine Position im Entwicklungsprozess eine zentrale Rolle. Weiterhin beschreibt das Schema die Gemeinsamkeiten und Unterschiede der verschiedenen Anwendung der betreffenden Domäne.

Um nun aber eine DIE auf diese Sichten anzupassen wird eine formale, tool-verarbeitbare Form dieses Schemas benötigt: die Software Factory Template. Templates können in Visual Studio eingelesen werden um das Werkzeug für die betreffende Domäne anzupassen. Die Template

- Stellt die nötigen Frameworks und Bibliotheken zur Verfügung
- Definiert zusätzliche Projekttypen die für die betreffende Domäne vorstrukturiert sind
- Bringt auch passende Buildscripte mit
- Und erweitert die DIE mit zusätzlichen DSLs, sowie deren Editoren und Transformatoren

Um diese Template zu erstellen, benötigen wir natürlich – wiederum in Visual Studio – die nötigen Werkzeuge. Beispielsweise werden Werkzeuge zur Definition von Metamodellen, konkreter (grafischer) Syntax und Transformationen zur Verfügung gestellt.

Wie aus obiger Diskussion hervorgehen sollte, kann man MDSD und SF nicht direkt vergleichen, weil SF einerseits einen weiteren Fokus hat („PLE à la Microsoft“) und andererseits de facto spezifisch für Visual Studio ist.

Im Rahmen von SF werden die MDSD-Begrifflichkeiten, wie sie am Anfang des Artikels eingeführt wurden, mehr oder weniger unverändert übernommen. Die Modelle sind in aller Regel grafisch (jedenfalls ist Microsoft's aktuelles Tooling auf grafische Modelle ausgelegt). Microsoft verwendet keinen der OMG Standards: es wird nicht nur kein UML zur Modellierung verwendet (eine Entscheidung, die ich gut nachvollziehen kann), es wird aber auch nicht auf die MOF gesetzt, sondern Microsoft verwendet dafür sein eigenes (sehr EMF-ähnliches) Metametamodell, genannt MDF (für Meta Data Framework).

Model-Integrated Computing

Model-Integrated Computing kommt ursprünglich aus der Ecke der verteilten Echtzeitsysteme (Distributed Realtime and Embedded (DRE) Systeme). Solche Systeme werden in verschiedenen Domänen verwendet, darunter Steuerungs- und Prozessautomatisierungssysteme, Verteidigungstechnik und Avionik. Daher wird der Begriff MIC auch insbesondere von den Leuten aus diesen Branchen verwendet, beispielsweise Vanderbilt University's Institute for Software Integrated Systems (ISIS). Vom ISIS kommt auch ein im Rahmen von MIC oft eingesetztes Werkzeug: das Generic Modelling Environment [GME]. MIC ist konzeptionell quasi identisch mit MDSD. Insbesondere spielt bei MIC UML nur eine untergeordnete Rolle. Einige Dinge sollten jedoch speziell erwähnt werden:

- Modelle werden für die Beschreibung des kompletten Lifecycles eines Systems verwendet, nicht nur für dessen Entwicklung. Analyse, Verifikation, Integration und Wartung werden genauso betrachtet.
- Da MIC oft für sicherheitskritische Systeme verwendet wird, ist die Verifikation von Modellen besonders wichtig - insbesondere die Simulation von Modellen spielt hierbei eine große Rolle.
- Modell-zu-Modell Transformationen sind wichtig. Allerdings nicht unbedingt für MDA-artige mehrschrittige Transformationen, sondern um (bestimmte Aspekte von) Modellen in andere Repräsentationen zu überführen um damit verschiedene, bereits existierende Simulations- und Verifikationswerkzeuge zu integrieren.

Interessant ist, dass MIC nun auch von der OMG unterstützt wird. Diese Unterstützung umfasst derzeit die MIC PSIG (Platform Special Interest Group) und den jährlichen Industrieworkshop. MIC hat aber formal nichts mit MDA zu tun.

Language Oriented Programming

Der Begriff Language-Oriented Programming wird dieser Tage insbesondere von Sergey Dmitriev und seiner Firma JetBrains (Hersteller der IntelliJ IDE) verwendet. JetBrains arbeitet an einem neuen Produkt namens MPS, das Meta Programming System [MPS]. Mit diesem Werkzeug kann man seine eigenen textuellen DSLs definieren, die dann direkt in der IDE integriert sind, insbesondere in Editor und Debugger. Auch hier spielen domänenspezifische Metamodelle eine zentrale Rolle, die sich auch (textuell) mit MPS definieren lassen. MPS bringt auch DSLs zur Definition von Editoren mit.

MPS ist ein Exemplar dessen, was Martin Fowler als Language Workbench bezeichnet [MF05]. Er argumentiert, dass das die Frage der Akzeptanz von DSLs und MDS davon abhängt, ob es praxistaugliche Werkzeuge zur Definition und Integration von DSLs gibt. Aus seiner Sicht stellen Language Workbenches die "Killer App" für DSLs/MDS dar. Neben MPS gibt es noch weitere derartige Werkzeuge:

- GME, in vorigen Abschnitt bereits erwähnt, ist durchaus auch eine Language Workbench für grafische DSLs. Auch MetaEdit+ [ME] und Xactiums XMF Mosaic [XMF] fallen in diese Kategorie.
- Historisch gesehen hatte das Intentional Programming Projekt von Microsoft (von Charles Simonyi geleitet) die gleichen Ziele. Laut [EC00] waren die IP Tools sehr imposant – leider haben sie die Microsoft Research Labs nie verlassen.
- Übrigens haben Charles Simonyi und ein paar andere Leute ein Firma namens Intentional Software gegründet. Man darf gespannt sein, was diese Firma produzieren wird [IS].

Zusammenfassung

Man kann sich natürlich darüber streiten, ob nun MDS eine Spezialform von GP oder eine Spezialform von MDA ist, oder umgekehrt, und welches der allgemeinere Ansatz ist. Historisch war GP sicherlich einer der ersten, aber der Begriff hat es nie in den Mainstream geschafft. Im Endeffekt ist es ja auch egal – feststellen lässt sich jedoch zweifelsohne, dass die Idee, mittels für die Domäne passenden Sprachen, die man ggfs. selbst definiert sich immer weiter durchsetzt. Die Werkzeuge sind natürlich noch nicht perfekt, aber durchaus sehr praxistauglich.

Referenzen

- AND AndromDA, www.andromda.org
- EC00 Eisenecker, Czarnecki, Generative Programming, Addison-Wesley 2000
- EMF Eclipse.org, Eclipse Modelling Framework, www.eclipse.org/emf

GME	ISIS, Generic Modelling Environment, www.isis.vanderbilt.edu/projects/gme/
GMF	Eclipse.org, Graphical Modelling Framework, www.eclipse.org/gmf
GS04	Greenfield, Short, Software Factories, Wiley 2005
IS	Intentional Software, www.intentsoft.com
iUML	Kennedy Carter, iUML, www.kc.com/products/iuml.php
MDA	OMG, Model-Driven Architecture, www.omg.org/mda
ME	Metacase, MetaEdit+, www.metacase.com/
MPS	JetBrains, Meta Programming System, www.jetbrains.com/mps/
MF05	Martin Fowler, Language Workbenches, www.martinfowler.com/articles/languageWorkbench.html
MV06a	Markus Völter, Abbildung von Variabilitäten in Softwaresystemen, www.voelter.de/conferences/index/detail1387037663.html
MV06b	Markus Völter, Software Architecture Patterns, www.voelter.de/publications/index/detail360729014.html
oAW	openArchitectureWare, www.openarchitectureware.org
SV05	Stahl, Völter, Modellgetriebene Softwareentwicklung, dPunk, 2005
XMF	Xactium, XMF Mosaic, www.xactium.com/