# Einführung in die modellgetriebene Software-Entwicklung

Udo Kelter

03.11.2009

**Zusammenfassung dieses Lehrmoduls**

Die modellgetriebene bzw. modellbasierte Software-Entwicklung ist eine kommende wichtige Methode der Software-Entwicklung. Dieses Lehrmodul skizziert die grundlegenden Vorgehensweisen (Übersetzeransatz und Interpreteransatz) und zugehörige Infrastrukturen. Ferner werden Kriterien beschrieben, wann die modellbasierte Software-Entwicklung sinnvoll ist.

**Vorausgesetzte Lehrmodule:**

empfohlen:
- Metamodelle
- Die Unified Modelling Language (UML) Version 2

**Stoffumfang in Vorlesungsdoppelstunden:** 0.5

# Inhaltsverzeichnis

# 1 Motivation

Die modellgetriebene Software-Entwicklung (englisch *model-driven (software) development*, abgekürzt **MDD** oder **MDSD**) ist einer der wichtigsten Trends der letzten 10 Jahre in der Softwaretechnik und wird noch auf Jahre hinaus ein wichtiges Thema bleiben. Man verspricht sich von dieser Entwicklungsmethode bzw. Technologie, die Entwicklung von Software stark zu beschleunigen und damit zugleich preiswerter zu machen, indem möglichst große Teile bisher von Hand geschriebener Anteile eines Softwaresystems durch kompaktere Modelle ersetzt werden. Zugleich kann hierdurch die Qualität der Software verbessert werden.

Es sind mehrere andere Bezeichnungen für diese Technologie üblich: das englische *driven* kann man eigentlich besser mit "gesteuert" oder "orientiert" übersetzen; daß Modelle Entwicklungsprozesse antreiben, ist im wörtlichen Sinne schlecht vorstellbar. Häufig wird die Bezeichnung **modellbasierte Software-Entwicklung (MBSE)** (*model-based (software) development*) verwendet, die inhaltlich am passendsten erscheint. Weitere Bezeichnungen sind *model-driven engineering* (**MDE**) und *model-driven architecture* (MDA; ein völlig mißlungener Begriff, mehr hierzu später).

# 2 Grundlegende Ansätze

Die MBSE unterstellt eine stufenweise Konkretisierung eines System analog zum Phasenmodell, in dem beispielsweise folgende Konkretisierungsstufen auftreten können: Analyse-Modell, Entwurfsmodell und Programmcode. Die Kernidee der MBSE ist,

- entweder möglichst große Teile des Programmcodes durch *Übersetzen* eines Modells zu generieren oder
- ein Modell zu *interpretieren*.

Beide Ansätze unterscheiden sich in einigen Details, auf die wir anschließend eingehen. Auf jeden Fall werden die Modelle damit in gewisser Weise zu ausführbaren Programmen. Ähnlich wie man beim Übergang

 

von Assemblersprachen zu höheren Programmiersprachen den Umfang des zu schreibenden Programmtexts stark reduzieren kann, kann beim Übergang zu Modellen als ausführbaren Einheiten der Erstellungsaufwand deutlich verkleinert werden.

## 2.1 Übersetzungsansatz

Beim Übersetzungsansatz geht man davon aus, daß man i.a. nicht den kompletten Quelltext des zu entwickelnden Systems aus den Modellen ableiten kann, sondern bestimmte Teile weiterhin von Hand schreibt. Beispielsweise kann man aus Datenmodellen die Datenhaltungsschicht einer Applikation weitgehend generieren, nicht hingegen qualitativ gute Bedienschnittstellen.

Oft ist es günstiger, die Modelle nicht in einem einzigen Schritt in den Quellcode zu transformieren, sondern in mehreren Schritten. Bei jedem dieser Schritte kann z.B. eine bestimmte Technologieentscheidung umgesetzt werden, ferner können ggf. manuell geschriebene Teile hinzugemischt werden.

Insgesamt erfordert der Übersetzungsansatz eine **MBSE-Infrastruktur**, die folgende Funktionen anbietet:

- einen oder mehrere Übersetzer für Modelle und weitere Werkzeuge, beispielsweise Mischfunktionen, die manuelle und generierte Teile zusammenfügen, insb. nach Modelländerungen
- ggf. Bibliotheken, die unverändert oder in angepaßter Form zum Teil des entwickelten Systems werden

Die Architektur des entwickelten Systems sollte natürlich möglichst so gestaltet werden, daß die generierten Teile die Form kompletter Module bzw. Subsysteme haben, was die Mischung mit den handgeschriebenen Teilen sehr vereinfacht. Sofern allerdings mehrere unterschiedliche Modelltypen eingesetzt werden, die unterschiedliche Aspekte eines Systems modellieren, können komplexe Mischprobleme auftreten.

 

## 2.2 Interpreteransatz

Beim Interpreteransatz benötigt man ebenfalls ein Modell - es kann im Prinzip das exakt gleiche Modell wie beim Übersetzeransatz sein -, es wird aber nicht übersetzt, sondern von einer vollständigen Applikation direkt "interpretiert".

Beispielsweise kann man Analyseklassendiagramme und ähnliche Datenmodelle so verstehen, daß diese einen Datenbestand spezifizieren, dessen Struktur ein Graph ist, in dem die Knoten und Kanten durch die Klassen und Assoziationen des Analyseklassendiagramms definiert werden. Man kann nun einen "generischen" Editor realisieren, der beliebige derartige Graphen editieren kann und der durch ein konkretes Klassendiagramm gesteuert wird. Der Editor ist generisch in dem Sinne, daß er mit beliebigen Typen von Knoten und Kanten arbeiten kann.

Wie die Graphen optisch dargestellt werden, wie Editierdialoge aussehen, wie Graphen z.B. in Dateien gespeichert werden usw. ist in einem Klassendiagramm nicht definiert, diese Entwurfsentscheidungen werden alleine durch den generischen Editor definiert und sind im einfachsten Fall in dessen Code hart verdrahtet. Entsprechende Funktionen sind beim Übersetzungsansatz in den handgeschriebenen (oder aus zusätzlichen Modellen generierten) Quelltexten zu realisieren.

Man kann die Rolle des Klassendiagramms in diesen Strukturen verschieden auffassen, entweder als **Konfigurationsparameter** des Editors, vor allem dann, wenn der Editor noch viele weitere Konfigurationsparameter hat, oder als "Quellprogramm", das der Editor interpretiert, analog zu einem Interpreter für eine textuelle Programmiersprache.

## 2.3 Vergleich beider Ansätze

Beim Interpreteransatz braucht man nur noch das Modell zu liefern und hat danach sofort ein lauffähiges System, ohne irgendwelchen Quellcode kompilieren zu müssen, was wiederum eine komplette Programmentwicklungsumgebung erfordert. Insofern ist der Interpreteransatz einfacher handhabbar und führt schneller zum Ergebnis.

Andererseits ist der Interpreteransatz wesentlich unflexibler als der

Übersetzeransatz: beim Übersetzeransatz hat man völlige Freiheit bei der Gestaltung der nicht generierten Systemteile, während man die generische Applikation mit vertretbarem Aufwand nur wenig ändern kann oder gar nicht, wenn man deren Quellcode nicht hat.

# 3 Einsatzkriterien

Beide Ansätze sind bei der erstmaligen Anwendung deutlich aufwendiger (!) als eine direkte Implementierung eines Systems, die auf explizite Modelle verzichtet und bei der die Modelle sozusagen hart verdrahtet im Quellcode realisiert sind.

Besonders hohen Mehraufwand kann der Übersetzeransatz verursachen, wenn hier zusätzliche Modelltransformatoren und andere Werkzeuge, die in konventionellen Software-Entwicklungsumgebungen nicht enthalten sind, realisiert werden müssen. Wir diskutieren das Aufwandsproblem daher i.f. vor allem unter der Annahme des Übersetzeransatzes. Auf die Frage, wie der Aufwand für die speziellen MBSE-Werkzeuge reduziert werden kann, gehen wir erst später ein.

Modellbasierte Entwicklungsmethoden lohnen sich somit letztlich nur, wenn der initiale Mehraufwand auf die Dauer amortisiert wird; dies ist wahrscheinlich, wenn eine oder mehrere der folgenden Bedingungen zutreffen:

1. wenn eine *große Zahl ähnlicher Systeme* erstellt werden muß, z.B. eine Systemfamilie: die gemeinsame Funktionalität kann als generische Programmfunktionalität des Interpreters realisiert werden bzw. als fester, nicht generierter Programmcode, der ggf. als Bibliothek eingebunden wird. Die gemeinsame Funktionalität kann natürlich selbst konfigurierbar gestaltet werden.
2. wenn beim evolutionären Vorgehen *viele Varianten* eines Systems benötigt werden und wenn die Varianten i.w. durch Änderung der Modelle realisiert werden können: ähnlich wie beim ersten Fall werden auch hier viele Systemvarianten erstellt, nur zeitlich nacheinander.
3. wenn man auf diese Weise billig einen *schnellen Prototypen* realisieren kann

4. wenn sich *Implementierungstechnologien häufig ändern* und deswegen in einem konventionellen System umfangreiche Anpassungsarbeiten anfallen würden: bei einem modellbasierten Verfahren muß (im Idealfall) nur der Modelltransformator angepaßt werden[1].
5. wenn die gleiche Applikation für *mehrere Zielplattformen* realisiert werden muß und dies durch unterschiedliche Generierung erzielt werden kann: unter dem Schlagwort "*model once, run anywhere*" war dies eine Hauptmotivation für den MDA-Ansatz. Pro Zielplattform braucht man im Prinzip einen eigenen Transformator. Wenn die Zielplattformen relativ ähnlich sind, ist es geschickter, einen einzigen konfigurierbaren Transformator zu entwickeln, der mittels passender Konfigurationsparameter für eine gewünschte Zielplattform eingestellt werden kann.

Um diese Bedingungen gut ausnutzen zu können, müssen die MBSE-Infrastrukturen passend gestaltet werden.

# 4 Modelltransformationen und MDA

Wie schon erwähnt kann man ein Modell in *mehreren Schritten* in Code übersetzen. Nach jedem Schritt entstehen eigene Zwischenstufen mit bestimmten Merkmalen. Eine derartige Sequenz von Transformationsschritten und Zwischenergebnissen hat die OMG unter dem Schlagwort "Model Driven Architecture" [OMG03] publiziert.

Zunächst muß hier leider davor gewarnt werden, daß es sich hier im Gegensatz zum Titel nicht um eine Architektur handelt, sondern um einen Begriffsrahmen, der eine modellbasierte Entwicklungsmethode bescheibt[2]. Diese begriffliche Schlamperei ist sehr ärgerlich, da sie unnötig verwirrt. Abgesehen davon ist [OMG03] sehr lesenswert. Der in

[1] Die Anpassung von Modelltransformatoren mag zwar einen geringeren Umfang haben, erfordert aber u.U. eine wesentlich höhere Qualifikation.

[2] Der MDA Guide widerspricht sich in dieser Hinsicht selbst. Einerseits wird MDA als "approach" definiert, z.B in Abschnitt 2.1.1 Background: " Model Driven Architecture™or MDA™... is not, like the OMA and CORBA, a framework for implementing distributed systems. It is *an approach to using models in software development.* ".

[OMG03] definierte Begriffsrahmen charakterisiert vor allem sinnvolle Zwischenstufen in einem mehrstufigen Transformationsprozeß:

1. Ausgangspunkt ist ein **Computation Independent Model** (**CIM**), unter dem man auch mehrere Teilmodelle, die unterschiedliche Sichten einnehmen, verstehen kann. Neben dem eigentlichen System kann hier auch der Anwendungskontext beschrieben werden. Ein wichtiges Ziel ist es, allgemeine Begriffe zu definieren, um die Kommunikation zwischen Entwicklern und Anwendern zu unterstützen. Offen bleibt, ob und wie derartige Modelle maschinell übersetzbar sind, man wird sie eher von Hand übersetzen müssen.
2. **Platform Independent Model** (**PIM**): Das plattformunabhängige Modell ist der eigentliche Startpunkt für maschinelle Transformationen. Es können beliebige geeignete Sprachen zur Formulierung der Modelle genutzt werden, sowohl Standardsprachen wie domänenspezifische. Die Plattformunabhängigkeit besteht darin, von den speziellen Eigenschaften aller infrage kommenden vorhandenen oder zu erwartenden Zielplattformen zu abstrahieren.
3. **Platform Specific Model** (**PSM**). Das plattformspezifische Modell eines Systems implementiert das PIM auf Basis einer konkreten Plattform, die aus konkreten Basistechnologien wie Programmiersprache, Komponentenmodell, Kommunikationsprotokolle, Datenverwaltungssystem usw. besteht.

# 5 MBSE-Infrastrukturen

## 5.1 Entwicklung einer MBSE-Infrastruktur

Wie schon oben erwähnt wird bei der MBSE der Umfang der originären Entwicklungsdokumente gegenüber der klassischen Programmierung deutlich gesenkt: umfangreicher textueller Quellcode wird durch Modelle ersetzt, die wesentlich kompakter sind. Das bedeutet umgekehrt,

Andererseits wird in Abschnitt 2.2.4 der Begriff Architektur wie allgemein üblich definiert: "The architecture of a system *is a specification of the parts* and connectors of the system and the rules for the interactions of the parts using the connectors.".

daß nicht mehr jedes Detail der entstehenden Applikation nach Belieben gewählt werden kann. Die Transformatoren oder Interpreter geben sehr viele Detailentscheidungen vor. Anders gesehen sinkt die Anzahl der realisierbaren Systeme gegenüber der freien Programmierung deutlich. Die Nützlichkeit des MBSE-Ansatzes steht und fällt daher damit,

- wie sehr diese vorgegebenen, mit vertretbarem Aufwand nicht änderbaren Details dem Bedarf entsprechen und
- wie gut die konfigurierbaren Teile handhabbar sind und ob alle gewünschten Systeme in Rahmen der Konfigurationsmöglichkeiten realisierbar sind.

Um für eine bestimmte Applikationsdomäne eine gute MBSE-Infrastruktur zu entwickelt, müssen vor allem die Gemeinsamkeiten und Unterschiede der Einzelprodukte in dieser Applikationsdomäne gut verstanden sein. Hierzu ist es sinnvoll, die Applikationsdomäne zunächst generell hinsichtlich grundlegender Begrifflichkeiten, Standardarchitekturen usw. zu durchleuchten (vgl. oben CIM). Eine gezielte Domänenanalyse ist auch üblich, wenn andere, nicht modellbasierte Formen von Systemfamilien *geplant* entwickelt werden.

In der Praxis entstehen oft zunächst ungeplant einige ähnliche Einzelprodukte aus der Applikationsdomäne, die erst im nachhinein zu einer Familie integriert werden sollen. Hierzu sind die Implementierungen der Einzelprodukte zu analysieren und folgende Codeteile zu unterscheiden:

1. in allen Einzelprodukten identisch oder fast identisch auftretende Codeteile
2. aus Modellen ableitbare Codeteile (“schematische” Codeteile, Pattern-Instanzen)
3. individueller Code

Sowohl bei geplanter wie ungeplanter Entstehung einer MBSE-Infrastruktur besteht der nächste Schritt in der Entwicklung einer Standardarchitektur, die die einheitlichen und die variablen Komponenten der Systeme möglichst voneinander trennt. Die variablen Komponenten sollen natürlich aus Modellen oder anderen Konfigurationsdaten

generiert werden. In diesem Zusammenhang sind folgende Technologien auszuwählen:

1. Modellierungssprache (Metamodell, konkrete Syntax)
2. Modellübersetzer(technologie)
3. ggf. Auswahl von Zielplattformen

Sofern schon Einzelprodukte vorhanden sind, können diese in die nun definierte MBSE-Infrastruktur portiert werden.

## 5.2 Modelltransformatoren und -Übersetzer

Wie schon erwähnt müssen Modelle ggf. mehrfach transformiert werden, wobei in jedem Schritt bestimmte technologische Entscheidungen "materialisiert" werden. Ein Werkzeug, das wieder ein (verfeinertes) Modell erzeugt, bezeichnen wir i.f. als **Modelltransformator**, während ein **Modellübersetzer** einen Programm-Quelltext ausgibt.

Die vorgenannten Werkzeuge müssen Modelle einlesen und ggf. ausgeben. Dies führt zu der Frage, wie Modelle repräsentiert werden können. Benötigt werden einerseits persistente Repräsentationen zur dauerhaften Speicherung und für Werkzeuge, die mit Dateien arbeiten. Transiente Repräsentationen werden zum Datenaustausch zwischen Werkzeugen benötigt, die in einer Entwicklungsumgebung parallel laufen. Alle vorgenannten Repräsentationen hängen von den Merkmalen des Datenverwaltungssystems bzw. der Programmiersprache ab und sind insofern technologiespezifisch. Technologieabhängigkeiten sind natürlich unerwünscht, daher sollte man die Struktur der Modelle zunächst mittels Analyseklassendiagrammen modellieren[3]; letztere werden auch als **Metamodelle** bezeichnet. Aus ihnen können dann technologiespezifische Datenstrukturen abgeleitet werden.

Die benötigten Modelltransformatoren und -Übersetzer können im Prinzip als *monolithische Einzelwerkzeuge* für einen speziellen Modell-

[3]Bei den in der Praxis sehr bedeutenden UML-Modellen ist dies nicht der Fall; die UML-Metamodelle modellieren die Modelltypen in Form von Entwurfsklassendiagrammen, die auf gängige objektorientierte Programmiersprachen als Implementierungssprache ausgerichtet sind (vgl. auch Lehrmodul "Metamodelle").

typ realisiert werden. Das Dokumentschema ist hier typischerweise im Quellcode "hart verdrahtet". Da relativ viele Übersetzer benötigt werden, ist dieses Vorgehen i.a. zu aufwendig.

Naheliegenderweise wendet man daher die Grundidee der modellgetriebenen Software-Entwicklung wieder auf das *Problem, eine Familie von Transformatoren zu realisieren an*! Sowohl der Interpreter- wie der Übersetzeransatz kommen infrage, in der Praxis werden sehr häufig "generische" Transformatoren benutzt, die durch ein Metamodell oder eine daraus abgeleitete DTD gesteuert werden. Derartige "generische" Werkzeuge werden auch als **Meta-Werkzeuge** bezeichnet.

## 5.3 Das Eclipse Modeling Framework (EMF)

Das **Eclipse Modeling Framework** (**EMF**) ist eine häufig benutzte MBSE-Infrastruktur. EMF ist gemäß dem Übersetzeransatz aufgebaut: zentraler Bestandteil von EMF ist ein *Übersetzer*, der Entwurfs-Klassendiagramme, die die Rolle von Metamodellen spielen, in Java-Code übersetzt.

Wie schon erwähnt kann man aus Datenmodellen vor allem die Datenhaltungsschicht einer Applikation ableiten, dies ist auch hier der Fall: der generierte Java-Code kann Modelle anlegen, abfragen, manipulieren, serialisieren, validieren und auf Änderungen überwachen. Der Funktionsumfang dieser Datenhaltungsschicht ist relativ umfangreich und orientiert sich vor allem am Bedarf interaktiver Modelleditoren.

Neben dieser Datenhaltungsschicht, die Teil des entwickelten Systems wird, kann man auch JUnit-Code als Testtreiber generieren. Dieses Beispiel zeigt, daß man mit MBSE-Methoden beliebige im Rahmen eines Entwicklungsprozesses benötigte Dokumente generieren kann und sollte, soweit inhaltlich möglich.

Das Metamodell, das als Eingabe für den Übersetzer benötigt wird, kann im Prinzip mit einem beliebigen Werkzeug zur Bearbeitung von Klassendiagrammen erstellt werden; EMF bietet hierzu nur einen rudimentären Editor an, der ein Klassen"diagramm" als Baum darstellt[4].

[4]Die Bezeichnung *Modeling Framework* ist insofern irreführend, als das EMF kein Werkzeug ist, mit dem man Modelle entwickelt.

# Literatur

[EMF] The Eclipse Modeling Framework (EMF);
`http://www.eclipse.org/modeling/emf/`

[MM] Kelter, U.: Lehrmodul "Metamodelle"; 2009

[OMG03] MDA Guide Version 1.0.1; OMG, Document Number: omg/2003-06-01; 2003-06-12;
`http://www.omg.org/docs/omg/03-06-01.pdf`

 