Konzept JavaToJava-Generator

# Model Driven XYZ

Begriffe die mit „Model Driven“ beginnen, haben in der Software-Technik bereits einige Jahre auf dem Buckel, ihren Zenit aber bereits überschritten. Buzzwords wie „Model Driven Architecture“ (MDA), „Model Driven Design“ (MDD) und „Model Driven Development“ (ebenso MDD) galten (wiedermal) einige Jahre als neue Silver Bullets.

Richtig eingesetzt, mögen sie eine wertvolle Ergänzung im Toolbaukasten eines Architekten oder Software-Ingenieurs sein. Genau dieses „richtig“ einsetzen ist allerdings die hohe Kunst.

## Die Theorie

Grundlegendes Element eines MDx-Ansatzes stellt das Modell dar. Das Modell beschreibt gegebene Sachverhalte in Form einer Modellierungssprache (z.B. einer domain specific language). Die Modellierungssprache ist eine formale Sprache, und hat eine fachliche Natur, konzentriert sich also auf die Problemstellungen, die in einer Anwendung oder einem Anwendungsteil inhaltlich gelöst werden sollen. Durch diese Konzentration auf Fachlichkeit hat die Modellierungssprache einen höheren Abstraktionsgrad als general purpose programming languages. Die Idee ist: Sobald man die Modellierungssprache zur Verfügung hat, geschieht die weitere „Programmierung“ oder „Erstellung“ der konkreten Anwendungsteile durch Programmieren in der Modellierungssprache, oder anders ausgedrückt: Durch Erweiterung oder Anpassung der zugrundeliegenden Modelle. Natürlich muss die formale Modellierungssprache letztlich in ausführbaren Code einer gegebenen Zielplattform umgesetzt werden. Diese möglichst eindeutige Abbildung des Modells auf eine Zielprogrammiersprache bzw. auf ausführbaren Zielcode ist eine wesentliche Eigenschaft einer MDA (model driven architecture).

Der wesentliche Vorteil, den man sich durch Verwendung der MDx-Techniken erhofft, basiert auf dem Grundprinzip der höheren Abstraktionsebene der Modellierungssprache:

* Die Modellierungssprache orientiert sich an fachlichen Begriffen der Problemdomäne. Dies ermöglicht es potentiell, Programmcode in der Modellierungssprache zu schreiben, der besser verständlich, und einfacher wartbar ist, wegen des Weglassens „unwichtiger“ technischer Details. Diese werden durch Generierung des ausführbaren Codes bzw. durch eine Laufzeitumgebung umgesetzt.
* Es wird potentiell eine Entkopplung der fachlichen Domäne und des fachlichen Anwendungscodes von der technischen Ausführungsplattform erreicht.
* Wahrung der Konsistenz von Spezifikation und ausführbarer Anwendung, das Modell soll also auch als („seine eigene“) Spezifikation dienen können oder zumindest als Basis einer Spezifikation
* Validierungstools können auf fachlicher Ebene auf das Modell losgelassen werden und bereits automatische oder halbautomatische Verifikation von semantischen Anteilen bieten
* Potentiell kann einfacher eine hohe Testabdeckung erreicht werden, indem nur die „primitiven“ Bestandteile der Modellierungssprache durchgetestet werden, deren Instaniziierungen für die konkrete Anwendung jedoch nicht notwendigerweise nochmals getestet werden müssen.
* Durch die fachliche Modellierung werden in generellen Programmiersprachen meist unumgängliche Redundanzen reduziert

## Die Praxis

Natürlich funktioniert das alles meist nur in ganz speziellen Fällen, und wenn erfahrene Experten genau das richtige tun.

Der erste Knackpunkt ist sicherlich das Finden einer passenden Abstraktion für die Problemdomäne. Kann man alle definierten Anforderungen mit der Abstraktion umsetzen, oder muss man Teile des Modells doch durch „handgeschriebenen“ Code umsetzen? Hat sie genügend Potential für Anpassungen und Erweiterungen, oder lassen sich damit nur genau die aktuellen Anforderungen umsetzen? Dieser Punkt besagt also: Wenn jede neue Anforderung in der Anwendung auch immer zu einer Erweiterung der Modellierungssprache führt, dann taugt diese nichts. Gleichzeitig sollte die Abstraktion nicht zu mächtig sein, also keine pathologischen Fälle erlauben, die zu Fehlern führen (entweder in der Abbildung auf lauffähigen Code, oder schlimmer, zur Laufzeit). Und drittens darf sie nicht zu „allgemein“ sein. Grafisches Modellieren eines Programms in UML erinnert doch recht stark an per Hand programmieren, ist aber deutlich weniger sinnvoll. Ist der Abstraktionsgrad der Modellierungssprache wieder nah an einer general purpose language, dann kann man sich diese auch schenken.

Natürlich gibt es auch technische Anforderungen an die Abstraktion: Sie muss formal genug sein, um das Modell eineindeutig in eine ausführbare Repräsentation umsetzen zu können. Diese Umsetzung muss dann auch performant möglich sein.

Da das Definieren der Modellierungssprache an erster Stelle steht, wirken sich solche Fehler oft gravierend auf die gesamte Anwendung bzw. das Projekt aus. Deshalb ist so eine Architektur sorgfältigst zu Designen und durch Prototypen zu erproben. Die technische Umsetzung in eine ausführbare Zielrepräsentation muss natürlich zwingend in der Designphase mit erwogen werden. Damit ist das initiale Design der Modellierungssprache und die Erstellung der technischen Abbildung ein riesiger Aufwandsposten in der Erstellung der gesamten Anwendung. Nach diesem Initialaufwand ist noch kein einziger Anwendungsfall der Anwendung (bis auf ggf. prototypische oder beispielhafte Anwendungsfälle) umgesetzt.

Die Probleme gehen natürlich weiter: Ohne eine geeignete Toolunterstützung für die Anwendungsfälle kann der erhoffte Vorteil ggf. ins Gegenteil verwandelt werden. Die Erstellung der Anwendungsarchitektur soll vereinfacht werden. Damit werden sinnvollerweise bestimmte Features, die man von general purpose programming languages bereits gewohnt ist, mit umgesetzt:

* Syntax highlighting
* Refactoring
* Automatischer „on-the-fly“ Syntax-Check
* Das Modell selbst muss sinnvoll verwaltet werden können, beispielsweise sollte es zumindest eine sinnvoll versionierbare Textdarstellung besitzen, es sollte durchsucht werden können, Teile des Modells sollten schnell auffindbar sein

Bei handgeschriebenen Anteilen der Anwendung, also Software, die in der Anwendung zur Laufzeit läuft, jedoch nicht modelliert wurde, sondern in einer general purpose programming language geschrieben ist, gilt Vorsicht. Zu unterscheiden ist hier:

* Libraries und utility Funktionen, die von der generierten technischen Umsetzung aufgerufen werden
* Technische Rahmenkomponenten, in denen das Modell läuft (sozusagen eine selbst geschriebene Laufzeitumgebung des Modells), falls vorhanden
* „Programmmodule“, die vom Modell aus eingebunden und aufgerufen werden

Die ersteren beiden Arten handgeschriebenen Codes sind oft unvermeidbar, und sind sicher teilweise anspruchsvoll, wirken sich aber idealerweise nicht auf das Modell selbst aus, denn sie sind nicht Bestandteil des Modells. Die zuletzt genannten „Programmmodule“ sind allerdings sehr wohl Teile des Modells selbst, und sie setzen teilweise sogar fachliche Anteile um. Ihr Ziel ist es meist, „in ausgesuchten Sonderfällen“ die „limitierte“ Funktionalität der Modellierungssprache zu umgehen bzw. zu erweitern, um etwas Besonderes zu erreichen. Hier muss man sich überlegen, wie sie ins Modell integriert werden können, wie sie zusammen mit dem Modell verwaltet werden usw. Vielfältige Probleme gibt es mit der Einbettung handgeschriebender Anteile, die in der Zielsprache formuliert sind.

Persönliche Meinung des Autors: Erste Praxiserfahrungen legen nahe, dass handgeschriebene Erweiterungen des Modells so gut wie nie vermieden werden können, sondern meist häufiger vorkommen, als manchem lieb sein mag. Oberstes Ziel sollte dennoch sein, solche weitgehend zu vermeiden. Es sollte sich maximal um Ausnahmefälle handeln. Hat man irgendwann mehr Programmmodule als Modellanteile, dann kann man wiederum die Sinnhaftigkeit und Angemessenheit der Modellierungssprache anzweifeln.

Angenommen, man hat eine passende Abstraktion gewählt, und hat es fertiggebracht, auch eine gute Toolunterstützung bereitzustellen. Dann sind das erste Voraussetzungen, dass zumindest die Initialversion der Anwendung das Licht der Welt erblicken kann. Man sollte aber noch folgende Punkte bedenken:

* Performance-Aspekte: Erst nach Fertigstellung der Anwendung zeigt sich, ob die Umsetzung der Modellierung in die Zielsprachen in allen Fällen zu performantem ausführbarem Code geführt hat. Ggf. sind hier noch Nachbesserungen in der Zielrepräsentation erforderlich, im schlimmsten Fall gar Nachbesserungen in der Modellierungssprache selbst.
* Fehleranalysen: Nicht-fachliche Probleme in der Anwendung sind ggf. schwieriger einzugrenzen. Handelt es sich um einen grundlegenden Fehler in der Modellierung, oder in der Modellierungssprache, oder in der technischen Umsetzung, oder in der technischen Infrastruktur?

Auch ist es meist ein Irrglaube, zu vermuten, das Modell ersetzt eine Spezifikation. Zumindest die Anforderungsanalyse muss „herkömmlich“ und gründlich erfolgen, und sollte in irgendeiner Form zu geschriebener Prosa führen. Die Modellierungssprache selbst muss gründlichst spezifiziert werden. Eine Skizzierung der fachlichen Anwendungsfälle in Prosa sollte auch erwogen werden. Modelle sind meist doch wieder zu formal und zu speziell, um daraus ohne weitere Hilfen sofort die Fachlichkeit wie in einem Buch nachlesen zu können. Ggf. kann aus einem Modell eine erste Übersicht oder ein Rumpf für eine Spezifikation generiert werden.

Größte Vorsicht ist vor der Herausgabe der Modellierungssprache an Anwender oder Kundenbetreuer der Anwendung „zur eigenen Erweiterung“ geboten: Was zunächst als tolles Feature erscheint, sorgt meist für riesige Probleme:

* Wie kriegt man es hin, dass Nicht-Informatiker ohne Programmiererfahrung plötzlich sinnvolle Modelle erzeugen können? Wie erreicht man es, dass sich diese Anteile sinnvoll in die Architektur und strategische Ausrichtung der Anwendung einfügen? Die Modellierungssprache mag noch so „fachlich“ und ggf. sogar „grafisch“ und „nett anzuschauen“ sein. Dies ersetzt in aller Regel keine grundlegenden konzeptionellen, Datenmodellierungs- und/oder Software-Entwicklungsfähigkeiten.
* Wie verwaltet man die Erzeugnisse der Anwender zusammen mit dem „Kern“-Modell? Wie stellt man sicher, dass solche Änderungen nicht wahllos durch jeden erfolgen können, und so ein Versionszoo von ausgerollten Anwendungsvarianten entsteht?

## Zusammenfassung

Der Autor hat bisher nur folgende rudimentäre Erfahrungswerte mit MDx sammeln können:

* Projekt bei einem Automobilhersteller: Verwendung einer UML-basierten grafischen Modellierungssprache zum Modellieren ausführbarer Testfälle am Fahrzeug. Einziger möglicher Vorteil, der sich hierdurch ergeben hat: Die Grafische Darstellung ist in einigen Sonderfällen etwas übersichtlicher als der Code. Aber in vielen Fällen galt: Die grafische Repräsentation ist sehr nah an echter Programmierung. Sogar die Abstraktion war recht löchrig: Waren doch Python-konforme Ausdrücke an vielen Stellen Voraussetzung. Damit hatte das Modell nur geringfügig höheren Abstraktionsgrad als direkte Programmierung in Python. Immerhin wurde eine Fehlbenutzung des technischen Frameworks durch Zwängen in ein rigides Korsett nahezu vermieden. Allerdings suggeriert die programmiernahe Ausrichtung der Modellierungssprache eine Mächtigkeit, die nicht vollständig gegeben ist, was dann zu Enttäuschung führt, wenn einige Python-Sprachmittel, die eine einfachere Formulierung ermöglicht hätten, eben doch nicht genutzt werden können. Negativpunkt war die Versionierung von Modellbestandteilen, da keine sinnvolle textuelle Repräsentation der Modellbestandteile vorhanden war.  
  Auch hier war ein wesentlicher Kerngedanke die Erstellung der meisten Modellbestandteile durch die Endanwender der Software selbst. Insbesondere führte dies zur Frage: Wie verwaltet man die dadurch unglaublich vielfach kursierenden „Anwenderbibliotheken“ und –Erweiterungen zusammen mit den Kernbestandteilen?
* Verwendung eines ebenso grafischen Modellierungsansatzes in einem Nachbarprojekt: Das Modell ist quasi ausführbar. Es muss aber an vielen Stellen durch handgeschriebenen Code erweitert werden. Größter Minuspunkt ist hier, dass der Kunde selbst die Modelle erstellen kann, und dass quasi keine fachliche Spezifikation der Anwendung existiert.

Ein erstes grundlegendes Fazit ist daher:

* Eine fachliche Anforderungsanalyse ist natürlich auch hier erste Pflicht.
* Erst dann sollte initial sorgfältigst abgewogen werden, ob der Einsatz von MDx-Methodiken in Kernbestandteilen der Anwendung sinnvoll und machbar ist, oder nicht. Diese Entscheidung muss zusammen mit dem Kunden gefällt werden. Zudem sollte dann möglichst früh eine Entscheidung für eine passende Toolkette und technische Laufzeitumgebung gefällt werden.
* Ein Team und insbesondere ein Architekt und ein Projektmanager, die jahrelange (erfolgreiche) Erfahrung mit MDx haben, sollten von Anfang an einem solchen Projekt teilnehmen (auch natürlich bereits bei der Abwägung der Entscheidung für oder gegen eine MDA).
* Der wichtigste Knackpunkt ist zu Beginn des Projektes das Design einer passenden Modellierungssprache zusammen mit einer technischen Infrastruktur. Insbesondere sollte zumindest ein Prototyp oder ein ganzer Anwendungsfall damit umgesetzt werden.
* Es sollte von Anfang an in die Planung eingehen und an den Kunden kommuniziert werden, dass die initiale Modellerstellung ein wesentlicher Aufwandsfaktor ist. Diesen sollte man unter keinen Umständen unterschätzen.
* Erweiterung des Modells durch „Programmmodule“, die in einer general purpose language geschrieben sind, sollte - wenn überhaupt möglich - nur in seltenen Spezialfällen in Erwägung gezogen werden. Und dann sollte es sich möglichst um wenige fachliche Inhalte handeln.
* Während des Designs ist die verwendete Toolkette sorgfältig an die eigenen Bedürfnisse anzupassen, ggf. zu erweitern und zu dokumentieren (insbesondere Nutzungskonzept). Ggf. müssen Features zur bequemen Verwendung der Modellierungssprache erstellt werden. Insbesondere ist die Definition eines passenden Konfigurationsmangements der Anwendung inkl. Modellanteile notwendig. In der Regel sollten „teure“ tolle Produkte mit Argusaugen akribisch auf alle Anforderungen geprüft werden, bevor man sie verwendet.
* Auch wenn die Modellierungssprache „fachlich“ ist: Sie sollte dennoch nur den (programmiererfahrenen) Software-Entwicklern der Anwendung vorbehalten bleiben. Nur in außergewöhnlichen Sonderfällen sollte erwogen werden, die Modellierung und ggf. sogar Erweiterung der Anwendung den Kundenmitarbeitern oder gar Usern der Anwendung zu überlassen. Sollte dies erwogen werden, muss klar definiert werden, wer unter welchen Bedingungen selbst Modellanteile erstellen kann, wie diese verwaltet werden, und welchen Qualitätsanforderungen sie genügen sollen. Es muss ein klarer Prozess definiert werden, wie solche Erweiterungen in den Entwicklungsprozess der Anwendung integriert werden. Die entsprechenden Nutzer müssen von vornherein die nötige Qualifikation zur Modellierung haben und entsprechend geschult werden.

Wenn man sich diese ganzen Probleme nicht aufhalsen will, dann kann man zwei Schritte zurücktreten, eine MDA nicht umsetzen, und sich lediglich für die Verwendung „passiver“ oder Einmal-Generatoren zu entscheiden, und die Software-Entwicklung durch „Case-Tools“ lediglich zu unterstützen, statt Kernbestandteile der Anwendung selbst durch eine Modellierungssprache und entsprechende Generatoren zu implementieren – wie im nächsten Abschnitt diskutiert.

# Generatoren

Bei MDx sind Software-Generatoren meist ein wesentlicher Bestandteil: Sie setzen das Modell in ausführbaren Code um. Dies erfolgt meist zur „Build-Zeit“.

## Generierung ohne MDA

Will man keine MDA für eine Anwendung nutzen, dann kann man von Generatoren dennoch profitieren: Ihre Verwendung ist in den meisten Fällen nicht mehr wegzudenken. Wir wollen hier Compiler für general purpose programming languages nicht als Generatoren im engeren Sinne sehen. Wir meinen hier letztlich Tools, die *während der Entwicklungsphase* einer Anwendung bei der Erstellung der Artefakte der Anwendung unterstützen.

Hier gibt es unterhalb des Levels MDA vielfältige Abstufungen:

* In den 90ern träumte man von „CASE-Tools“, die aus UML-Diagrammen (im Idealfall „lauffähige“) Programme erstellen können. Dies ist letztlich der Urgedanke der MDA.
* Hier konnte man sich zur Weiterpflege des Modells und erneuten Generierung entschließen („MDA“) oder es bei einer „Einmalgenerierung“ belassen. Da häufig oder eigentlich manuelle Erweiterungen der generierten Sourcen erforderlich war, musst man in ersterem Fall Maßnahmen definieren (durch Tools unterstützt), wie die handgeschriebenen Teile mit geänderten Modellanteilen zusammengebracht und verwaltet werden können. Dieses Problem ist weiterhin bei MDA erhalten geblieben.
* Einen Abstrich stellt die „Einmalgenerierung“ einer Rumpf-Anwendung zu Beginn der Anwendungsentwicklung dar. Beispielsweise ist die Architektur statisch und ggf. zusätzlich dynamisch in UML modelliert, und das jeweilige Tool generiert daraus dann Rümpfe von Klassen, Interfaces und Methoden, die dann zumeist per Hand ausgefüllt werden. Eine ständige Synchronisierung zwischen Modell und handgeschriebenen Code ist schwierig und wird dabei meist nicht genutzt, sodass das ursprüngliche Modell schnell veraltet. Alternativ gibt es spezielle Generatoren für Anwendungen, die auf bestimmten Frameworks wie EJB oder Spring basieren.
* Dann gibt es noch in zahlreichen Tools der Entwicklung integrierte Ad-Hoc-general-purpose Generatoren, beispielsweise in IDEs integrierte Wizards, die Klassen-Rümpfe oder spezielle Methoden generieren.

## Ziele der Verwendung von Generatoren

Ziel der aufgeführten Arten von Generierung ist es meist, bestimmte Limitierungen der Programmierumgebung zu umgehen. Beispielsweise „boiler plate code“ generieren zu lassen, statt ihn per Hand zu schreiben. Dieses Vorgehen ist Usus. Zu viel Notwendigkeit für solche Art von Generierung spricht ggf. dafür, dass die verwendete Programmierumgebung (Sprache, Framework, Laufzeitumgebung etc.) ggf. für die Aufgabe ungeeignet ist.

Wiederholungen sind oft unumgänglich, und manchmal sogar durch die Architektur bedingt. Beispielsweise führt das Aufteilen eines Anwendungsfalles in Schichten (z.B. Fassade, Logik, Persistenz) zwangsläufig zu einer gewissen Form der Redundanz in den Schnittstellen der umsetzenden Klassen. Zweites Beispiel ist das Definieren von Transportobjekten mit nahezu identischen Attributen wie Entitätsklassen. Weitere Beispiele stellen Implementierungen von toString(), hashCode() und equals() in Java dar, die häufig nach exakt gleichen Mustern erfolgen, und deren Generierung in vielen Fällen daher auch aus den Klassenattributen erfolgen kann.

Solche Redundanz lässt sich dann durch Generierung nicht vermeiden, aber etwas erträglicher gestalten, indem ein Teil aus dem anderen generiert wird.

## Generierungs-Arten

Im Idealfall ist die Generierung aber „transparent“ für den Entwickler, indem sie zur Build- oder Laufzeit per entsprechender Konfiguration „vollautomatisch“ erfolgt. Ein solches Generat ist dann zwangsläufig ein „Wegwerfprodukt“ und zeichnet sich nicht ungern durch schlechte Lesbarkeit aus. Beispiele dafür:

* Durch Anwendungsserver generierte Boilerplate-Implementierungsklassen zur Umsetzung des wahnwitzigen EJB 2.1-Standard
* Aus XML-Schemata via JAXB generierte Java-Klassen
* Dynamische Java-Proxies

Wir können also im Wesentlichen drei Arten von generierten Artefakten unterscheiden:

* **Einmal-Artefakte:** Artefakte, die nur einmalig generiert und dann per Hand weitergepflegt werden. Die Generierung erfolgt meistens aus einem während der Spezifikationsphase erstellten Modell oder aber durch ein Tool der IDE.
* **Boilerplate-Anwendungs-Artefakte:** Artefakte mit „boiler-plate“-Charakter, die aufgrund eines verwendeten Standards oder Programmiermodells notwendig sind und immer wieder (wenn nötig) verlustfrei aus einem speziellen „Basis-Modell“ (XML-Datei, Java-Interface etc.) generiert werden, aber niemals per Hand erweitert oder angepasst werden. Diese Artefakte erzeugen zwingenderweise eine gewisse Redundanz im Code, allerdings eine automatisch generierte. So lässt sich Konsistenz der redundanten Anteile i.d.R. leichter sichterstellen, weil das Generat aus einer per Hand gepflegten Quelle generiert wird.
* **Misch-Artefakte:** Artefakte, die beides kombinieren, also Bereiche enthalten, die aus einer Quelle (ggf. immer wieder bzw. bei Änderungen) generiert werden, und solche Bereiche, die per Hand gepflegt werden.

Bei den letzten beiden Varianten ist die Aufgabe: Wie kann erkannt werden, dass eine Änderung in der Quelle erfolgt ist, und das Artefakt daher aktualisiert bzw. neu erstellt werden muss? Bei den Boilerplate-Anwendungsartefakten ist die Aufgabe meist einfach dadurch zu lösen, dass man die Generierung geeignet in den Buildprozess integriert. Werden die Schnittstellen so generierter Klassen jedoch in umgebendem Code genutzt (beispielsweise Aufrufe auf durch JAXB generierte Klassen), so entstehen durch eine Neugenerierung ggf. Fehler in der nutzenden Restanwendung. Idealerweise sind die Boilerplate-Anwendungsartefakte „self-contained“, in dem Sinne, dass die Restanwendung sie nicht referenzieren muss. Beispiele dafür sind die von Application Servern generierten Dummy-Implementierungen der EJB-2.1-Interfaces.

Die Mischartefakte stellen darüber hinaus eine besondere Herausforderung dar. In aller Regel handelt es sich um „von Außen“ genutzte Artefakte, es gelten also dieselben Probleme wie bei „Boilerplate“-Artefakten. Darüber hinaus können sie nicht einfach „weggeschmissen“ werden. Es muss bei jeder Aktualisierung entschieden werden, was mit handgeschriebenem Code passiert. Schwierigkeiten entstehen in dem Artefakt selbst, wenn dieser generierte Anteile des Artefaktes referenziert. Sollen diese Referenzierungen während der Generierung angepasst werden? Dies ist häufig nicht automatisch unter Wahrung der gleichen (oder einer sinnvollen geänderten) Semantik möglich. Gleiches gilt für Aufspaltungen und sonstige Refactorings des Quell-Artefaktes.

## Zusammenfassung

Auch für Anwendungen ohne MDA stellen Generatoren und Generierungsschritte ein meist unvermeidbares Hilfswerkzeug zur Sicherstellung der Konsistenz bei unvermeidbarer Redundanz dar.

Notwendige Programm- oder Artefaktskelette können durch Einmalgeneratoren erzeugt werden und erhöhen so in gewisser Hinsicht die Produktivität. Für die Verwendung von Frameworks erforderliche Artefakte (z.B. EJB-Implementierungen bestimmter Interfaces), die zwar erforderlich sind, aber manuell nicht angepasst werden müssen, können wiederholt im Rahmen des Buildprozesses der Anwendung generiert werden.