mehr zum thema:
www.LHsystems.com
www.kombiverkehr.de

ERFOLGREICHER EINSATZ VON AndroMDA BEI LUFTHANSA SYSTEMS

Bereits seit geraumer Zeit wird auf Konferenzen und in Zeitschriften über modellgetriebene Softwareproduktion berichtet. Obwohl die dargestellten Möglichkeiten auf großes Interesse stoßen, fehlt scheinbar immer noch das Vertrauen in die Effektivität von MDA. Eine der häufigsten auf Konferenzen gestellten Fragen ist: „Haben Sie damit schon mal ein großes Projekt durchgeführt?" Dieser Artikel berichtet über ein großes Projekt, in dem bei Lufthansa Systems erfolgreich das Open-Source-MDA-Framework AndroMDA eingesetzt wurde. Neben einer Beschreibung des Projektumfelds werden ganz konkrete Empfehlungen für den Einsatz von MDA in großen Projekten gegeben.

Ende 2003 wurde Lufthansa Systems mit der Realisierung eines neuen DV-Systems für das Geschäft der Firma Kombiverkehr beauftragt.

Die Firma Kombiverkehr organisiert und vermarktet ein europaweites Netz für den kombinierten Verkehr Schiene-Straße. Beim kombinierten Verkehr unterscheidet man unter anderem zwischen begleitetem und unbegleitetem Verkehr. Die Firma Kombiverkehr konzentriert sich auf den unbegleiteten Verkehr, bei dem die Ladeeinheit im Nahverkehr über die Straße zu einem Terminal des kombinierten Verkehrs gebracht und dort abgeliefert wird. Die Kombigesellschaft übernimmt die Ladeeinheit, lädt sie auf einen Güterwagen, stellt den Zug zusammen und fährt die Ladeeinheit über die so genannte „weite Strecke" in die Zielregion. Dort wird in einem Terminal die Ladeeinheit vom Zug auf das Straßenfahrzeug umgesetzt – mit oder ohne kurze Zwischenlagerung. Während der langen Strecke wird die Ladeeinheit nicht von einem Fahrer begleitet (vgl. [Sei97]).

Das neu zu entwickelnde System sollte aus fachlicher Sicht sowohl Buchungs- und Operationskomponenten als auch eine Planungskomponente enthalten. Hauptziel der Anwendung ist die Erhöhung der Auslastung der Güterzüge.

Kombiverkehr hatte bereits in einem früheren Projekt gute Erfahrungen mit der Sprache Java im Rahmen einer Zweischichtenarchitektur gesammelt, weshalb das neue System auf jeden Fall auch diesmal in Java realisiert werden sollte. Im Rahmen einer Vorstudie stellte sich sehr schnell heraus, dass eine Zweischichten-Architektur die Anforderungen, die an das System gestellt wurden, nicht tragen würde. Vor allem die Integration der rechenintensiven Optimierungsroutinen machte den Einsatz einer Mehrschichten-Architektur unabdingbar. Ende 2003 gab es für mehrschichtige Java-Anwendungen prinzipiell zwei Architekturausprägungen: „Enterprise Java Beans" (EJB) und „Spring" (vgl. [Spr]). Verteilte Systeme sind von sich aus sehr komplex. Spring ist gegenüber EJB zwar im Vorteil, verfügt allerdings über eine Konfigurationsdatei, die bei großen Projekten sehr komplex werden kann. Um dieser Komplexität Herr zu werden, sollte ein MDA-Generator eingesetzt werden.

Gründe für den Einsatz von AndroMDA

Der Einsatz einer neuen Technologie ist immer mit einem gewissen Risiko verbunden: Ist die Technologie reif für den produktiven Einsatz? Kann Support eingekauft werden? Gibt es ausreichend Literatur? Wie weit verbreitet ist die Technologie – kann man auf eine starke Community bauen oder steht man alleine da und muss alle Probleme mühsam selbst lösen? Wird die Technologie ihr Versprechen einlösen und die Entwicklung vereinfachen, beschleunigen oder wenigstens die Qualität erhöhen?

Diese Fragen betreffen eher die Risiken, die von einer neuen Technologie ausgehen. Genauso stellt eine neue Technologie aber auch immer eine Chance für Verbesserungen dar: Kann die Entwicklung zügiger voranschreiten? Kann das Projektteam mehr Use-Cases in derselben Zeit implementieren? Steigt die Qualität, weil Flüchtigkeitsfehler vermieden werden? Wird die Homogenität des Programmcodes gefördert? Auch diese Fragen wollen bedacht sein.

Schlussendlich sollte die Auswahl einer Technologie idealerweise unter Abwägung aller Chancen und Risiken stattfinden, was jedoch nicht immer möglich ist. Im vorliegenden Fall lagen bereits Erfahrungen mit AndroMDA (vgl. [And]) vor, die in einem überschaubaren Projekt gewonnen worden waren. Im Folgenden wollen wir auf die einzelnen genannten Fragen eingehen und retrospektiv unsere Erfahrungen weitergeben.

Verfügbarkeit von Support

Einer der am häufigsten geäußerten Vorbehalte gegenüber Open-Source-Software lautet: „Es gibt keinen Support, auf den man sich verlassen kann". Um Support leisten zu können, muss ein Open-Source-Projekt gewisse Voraussetzungen erfüllen: Der Quellcode muss so stabil sein, dass sich die Entwickler des Open-Source-Projekts nicht permanent auf das Beheben von Fehlern oder die Weiterentwicklung konzentrieren müssen. Die Zahl der Entwickler muss ausreichend hoch sein, damit die Support-Arbeit nicht an einigen wenigen hängen bleibt.

Gegenüber proprietärer Software, für die ein einzelner Hersteller Support leistet, hat ein weltweit verteiltes Open-Source-Projekt aber auch Vorteile: Üblicherweise sind die Entwickler auf mehrere Zeitzonen verteilt, sodass nahezu rund um die Uhr wenigstens ein Entwickler online ist. Auch die Identifikation der Entwickler mit ihrem Projekt ist üblicherweise sehr hoch. Diese

die autoren

Peter Friese (E-Mail: peter.friese@LHsystems.com) arbeitet als Softwarearchitekt für Lufthansa Systems. Im hier beschriebenen Projekt war er für die Gesamtarchitektur verantwortlich und hat das Implementierungsteam geleitet.

Matthias Bohlen (E-Mail: mbohlen@andromda.com) ist unabhängiger Softwarearchitekt und Berater für IT-Projekte. Er war im Kombiverkehr-Projekt als Architekt und Coach tätig und ist Gründer von AndroMDA.org.

Leidenschaft kann sich durchaus positiv auf die Support-Moral auswirken. All dies trifft auf AndroMDA zu. Das AndroMDA Team besteht derzeit aus 11 Entwicklern, die über den ganzen Globus verteilt sind.

AndroMDA gibt es seit 2001 und es liegt mittlerweile in der Version 3.1 vor. Weitere Entwicklungsschritte sind geplant und auch die bestehende Codebasis erfreut sich guter Pflege. Nahezu alle *Committer* setzen AndroMDA in ihrer täglichen Arbeit ein, sodass auftretende Fehler zügig behoben werden und auch neue Features zeitnah realisiert werden. Beiträge der Community werden begutachtet und ebenfalls zeitnah in die Codebasis aufgenommen. Die Qualität des Quellcodes wird dabei durch diverse Qualitätssicherungsmaßnahmen garantiert, unter anderem durch kontinuierliche Integration mittels des Tools *CruiseControl (*vgl. cruisecontrol.sourceforge.net*)*.

Open-Source-Projekte wie „JBoss" und „Spring" haben es vorgemacht: Wieso sollte man nicht zu einem Open-Source-Projekt eine professionelle Support-Organisation aufbauen, die kostenpflichtige Dienstleistungen und Schulungen anbietet? Für AndroMDA steht diese Art von Support über den *Professional-Open-Source*-Ableger „AndroMDA.com" ebenfalls zur Verfügung.

Im Projekt haben wir von beiden Formen des Supports Gebrauch gemacht: Alltägliche Anfragen zu bestimmten Features wurden über das Onlinesupport-Forum forum.andromda.org gestellt, wobei die Reaktionszeit üblicherweise innerhalb einer Stunde lag. Darüber hinaus hat das Projekt *KMS (Kombiverkehr Kapazitätsmanagementsystem)* kostenpflichtigen Support und Coaching über AndroMDA.com eingekauft, um einen Teil des Entwicklungsteams während einer wichtigen Phase zu schulen.

Das Argument, für Open-Source-Projekte gebe es keinen vernünftigen Support, können wir also nicht bestätigen.

Verbreitungsgrad

Modellgetriebene Softwareentwicklung wird zunehmend eingesetzt. Das kann man sowohl an der öffentlichen Diskussion des Themas (in Zeitschriften und auf Konferenzen) als auch an den steigenden Zahlen der Anwender ablesen. Die Anzahl der Anwender eines Open-Source-Tools lässt sich allerdings nur schwer abschätzen: Die Zahl der Downloads ist sicher nicht aussagekräftig, denn nicht alle Anwender laden immer die neueste Version herunter. Eventuell wird die Software auch nur herunter geladen und nie eingesetzt. Aussagekräftiger ist da schon die Zahl der Anwender, die auf Mailing-Listen und Foren angemeldet sind. Im AndroMDA Support-Forum sind derzeit ca. 600 Anwender registriert. Die Zahl der tatsächlichen Anwender dürfte also sogar noch etwas höher liegen, da sich nicht unbedingt jeder Anwender auf dem Forum anmeldet. Von den acht Entwicklern des hier beschriebenen Projekts sind z. B. nur drei angemeldet.

AndroMDA verfügt über eine starke Community: Gegenseitige Hilfe auf dem Forum ist eine Selbstverständlichkeit; darüber hinaus wurde vor Kurzem ein Schwesterprojekt („andromda-plugins") gestartet, das der Community eine Plattform für die Entwicklung und Veröffentlichung von eigenen *Plug-Ins* außerhalb des Release-Zyklus von AndroMDA selbst bietet.

Steigerung der Produktivität

Einer der maßgeblichen Aspekte bei der Entscheidung für AndroMDA war die zu erwartende Steigerung der Entwicklerproduktivität. Da man durch den Einsatz von *Model-Driven Architecture (MDA)* bzw. *modell-getriebener Softwareentwicklung (MDSD)* den Abstraktionsgrad erhöht, sollte die Anzahl der zu implementierenden Codezeilen deutlich sinken. Außerdem sollten die Entwickler sich idealerweise nicht mehr im Detail mit der Architektur auseinandersetzen müssen, sondern sich auf die Fachlichkeit konzentrieren können. Diese Erwartungen haben sich voll und ganz erfüllt.

Erfreulicherweise konnte die Produktivität jedoch auch noch an einer anderen Stelle gesteigert werden. Ursprünglich hatten wir erwartet, die für die Zielarchitektur notwendigen Transformationsregeln (die bei AndroMDA in Form von so genannten *Cartridges* zusammengefasst werden) zumindest teilweise selbst erstellen zu müssen. Rechtzeitig zum Projektbeginn wurde jedoch die „AndroMDA Spring Cartridge" veröffentlicht, die die Zielarchitektur fast vollständig unterstützt. Diejenigen Aspekte der Zielarchitektur, die von der *Cartridge* noch nicht unterstützt wurden, konnten im Laufe des Projekts durch die Erstellung von *Patches* ergänzt werden und sind mittlerweile integraler Bestandteil der *Cartridge* geworden.

Erhöhung der Qualität

Menschen machen Fehler, Computer nicht (und wenn, dann nur, weil sie falsch programmiert wurden). Der Einsatz eines Generators soll also auch die Softwarequalität erhöhen. Ganz wie gewünscht, hat der Einsatz von AndroMDA im Projekt zu einer stabilen Qualität der Software geführt. Am deutlichsten fällt auf, dass der Code sehr homogen ist. Bei herkömmlicher Entwicklung lässt sich schnell die „Handschrift" der Entwickler erkennen – nicht so bei einem MDA-Ansatz: Hier wird sehr viel Code durch den Generator erstellt und die Entwickler füllen ihre Implementierung nur noch an den vorgesehen Stellen ein. Somit werden die Entwickler auch zur Einhaltung der vorgegebenen Architekturstrukturen gezwungen – die Softwarearchitekten können sich also darauf verlassen, dass das System nach ihren Vorgaben erstellt wird. Der Bedarf an Code-Reviews sinkt dadurch drastisch.

Man sollte sich aber nicht nur auf den Generator verlassen – schließlich implementieren die Entwickler in nicht unerheblichem Umfang auch eigenen Code. Diesen sollte man durch den Einsatz geeigneter Werkzeuge, wie z. B. *PMD*, *CheckStyle* oder *FindBugs* (alles Open-Source-Produkte, vgl. [PMD], [Che], [Fin]) automatisiert überprüfen lassen. Für die genannten Tools gibt es *Plug-Ins* für die gängigsten Entwicklungsumgebungen, sodass die Entwickler ein direktes Feedback erhalten und nicht erst auf das Ergebnis der *Continuous Integration* warten müssen.

Bezüglich der Qualität des erzeugten Codes von Codegeneratoren hört man immer wieder, dass diese verbesserungswürdig sei. Oft wird zum Einsatz von so genannten *Code Beautifiern* geraten. Das ist unserer Meinung nach bei AndroMDA jedoch nicht erforderlich, da der erzeugte Code sehr ordentlich formatiert ist. Nur wenn man einen anderen Codestil als den von AndroMDA präferierten bevorzugt, benötigt man einen solchen *Code Beautifier*.

Erfahrungen und „Best Practices"

Projekte haben es an sich, dass nicht nur das Wissen über die Fachdomäne im Laufe der Zeit zunimmt, sondern auch der technische und organisatorische Erfahrungs-

Anteil	Funktion
Fachliche Fassade	Orientiert sich am Thema der Optimierung und orchestriert als *Controller* die Zusammenarbeit von Kollektor, Transformator und Optimierungskern.
Kollektor	Sammelt Stamm- und Bewegungsdaten für die Optimierung.
Transformator	Wandelt die gesammelten Daten in eine komprimierte, für den Optimierungskern leicht verständliche Darstellung (die „Objektschale") um.
Objektschale	Dünne Schicht aus Wertklassen (ValueObjects), enthält Eingabe- und Ergebnisdaten des Optimierungskerns.
Optimierungskern	Wandelt die Objekte der Objektschale in Matrixdarstellung um und übergibt sie an die in C++ geschriebene *Third-Party*-Bibliothek für lineare Optimierung.

Tabelle 1: *Architektur der Optimierungskomponenten*

schatz wächst. Aus den Lehren dieses Projekts möchten wir daher einige wichtige Punkte weitergeben.

Architektur-Patterns und Metamodelle

Verbesserungen entstehen immer auf der Basis von Erkenntnissen. Die Architektur (also Struktur, Verhalten und Stil) eines Systems wie *KMS* ist charakteristisch für eine bestimmte Klasse von Anwendungen, nämlich eine Mischung aus Informations- und Optimierungssystem. Wir haben in der Architektur wiederkehrende Muster erkannt, die bei der Betrachtung ganz deutlich auffallen. Komponenten des Systems folgen immer wieder einem bestimmten Zweck, anhand dessen wir sie in Kategorien einteilen können:

- Stammdatenmanagement,
- Bewegungsdatenmanagement,
- Realisierung der eigentlichen Geschäftsprozesse Planung, Reservierung, Buchung,
- Optimierungsverfahren und
- Reporting.

Innerhalb der Komponentenkategorie „Optimierungsverfahren" haben wir wiederum (eine Stufe feinere) Muster identifiziert. Die Optimierungskomponenten bestehen nämlich für jedes der verschiedenen Optimierungsthemen (Reservierungen, Buchungen, Beladung) jeweils aus denselben charakteristischen Anteilen (siehe Tabelle 1).

Hat man diese Systematik einmal erkannt, so lassen sich daraus Metamodelle (siehe Abb. 1 und Abb. 2) ableiten, die die im Projekt gefundenen Muster klar machen und zueinander in Beziehung setzen. Dadurch wird sichtbar, wie Optimierungskomponenten geschnitten, modelliert und implementiert werden müssen. Einmal gefunden, brachte dies für alle Beteiligten erhellende Einsichten.

Im Projekt war das Optimierungsmetamodell lange Zeit nur implizit im Gedächtnis der Modellierer vorhanden. Es gab eine Abbildungsvorschrift vom Optimierungsmetamodell in das Metamodell der technischen Architektur (siehe Tabelle 2).

Aufgrund dieser Abbildungsvorschrift

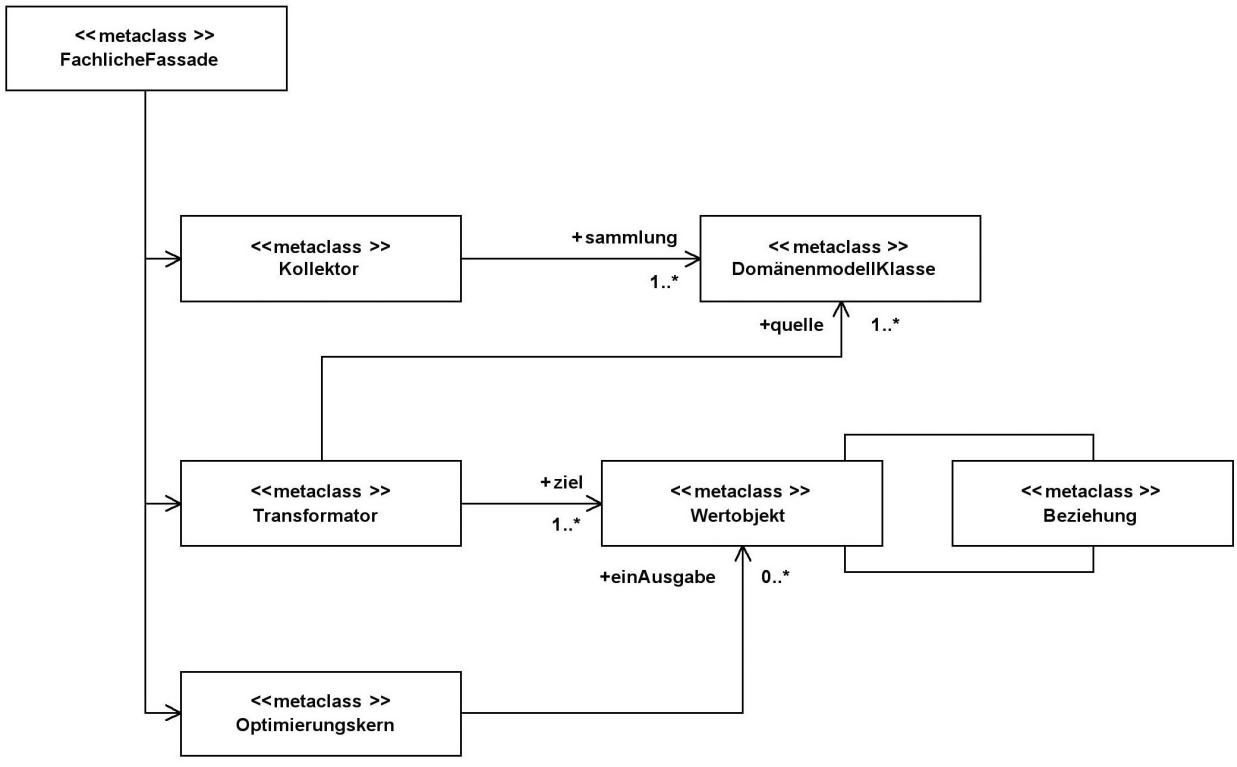

Abb. 1: *Metamodell für Optimierungskomponente*

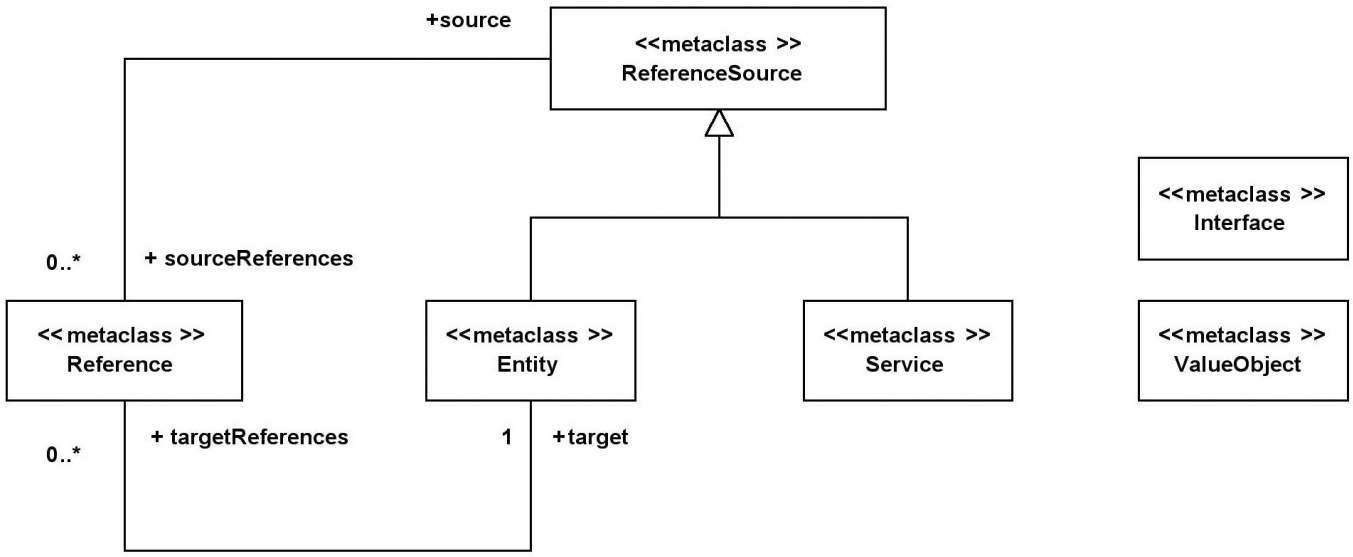

Abb. 2: Aussschnit aus dem Metamodell der technischen Architektur

modellierten wir fachliche Fassaden, Kollektoren und Transformatoren direkt als Services, Wertobjekte als ValueObjects und den Optimierungskern als Interface. Nach der Erkenntnis der wiederkehrenden Metamodelle würden wir dies natürlich nicht mehr tun, sondern dafür Modell-zu-Modell-Transformationen aus dem Optimierungsmetamodell in das Metamodell der technischen Architektur einsetzen (siehe unten).

Architekturänderungen im Projektverlauf

Architekturen unterliegen im Projekt denselben Lernprozessen wie der Quellcode. Menschen wissen nicht alles von Anfang an, sondern lernen erst im Laufe der Zeit, was es bedeutet, solch ein System zu bauen. Daher müssen Architekturen genau so gut änderbar sein wie der Quellcode.

Projektmanager konventionell (d.h. nicht mittels MDA) geführter Projekte werden jetzt aufhorchen: Architekturänderung? Das klingt nach erheblichem Aufwand! Richtig. Änderungen an der Architektur und ihrem Metamodell verursachen in der Regel großen Aufwand, da sich Prinzipien ändern und eine große Menge von Quellcode an die geänderten Architekturprinzipien angepasst werden muss.

Beim modellgetriebenen Vorgehen ist das anders. Man kann hier zwei Fälle unterscheiden:

- Änderungen der fachlichen Architektur
- Änderungen der technischen Architektur

Änderungen an der technischen Architektur werden beim modellgetriebenen Vorgehen durch Änderungen der Transformation vom Modell in den Quellcode realisiert. Da diese Transformation zum großen Teil automatisiert erfolgt, ist der Aufwand einer solchen Architekturänderung gering, sofern die Schnittstelle zum handgeschriebenen Code konstant bleibt. Ist diese Schnittstelle auch von der Änderung betroffen, steigt der Aufwand natürlich etwas an, ist jedoch immer noch nicht so groß wie im Falle des komplett von Hand entwickelten Quellcodes.

Änderungen an der fachlichen Architektur äußern sich beim modellgetriebenen Vorgehen meist zuerst in Änderungen des Modells. Klassen oder Methoden werden umbenannt, in andere *Packages* verschoben, Abhängigkeiten zwischen den Klassen ändern sich, neue Klassen werden eingeführt usw. – dieselben Dinge, die man auch bei Refactoring im Quellcode tun würde.

Refactoring im Modell

Refactoring in Modellen wurde bisher von den MDA-Werkzeugen nicht besonders gut unterstützt – da bildet AndroMDA keine Ausnahme. Ändert man den Namen einer Klasse, generiert AndroMDA einfach eine neue und die Klasse mit dem alten Namen bleibt unverändert stehen. Der Entwickler muss diese beiden Klassen nun manuell zusammenführen.

Wir behelfen uns im Projekt zurzeit damit, dass wir die Klassen zuerst in der verwendeten Entwicklungsumgebung (in diesem Fall „Eclipse") auf Quellcodeebene umbenennen, wobei Eclipse konsequent alle Stellen im Code (auch im generierten!) mit verändert. Anschließend ändern wir im Modell, AndroMDA überschreibt den alten generierten Code, und alles stimmt wieder.

In Zukunft wird das anders sein. AndroMDA wird Änderungen im Modell bemerken und Refactoring-Kommandos an Eclipse senden, um dieselben Änderungen im Code durchzuführen, die auf Modellebene bereits stattgefunden haben. Das wird es für die Entwickler deutlich einfacher machen, auch größere Änderungen einfach und sicher durchzuführen. So wird modellbasiertes Refactoring möglich gemacht.

Instanz von	wird realisiert als Instanz von
Fachliche Fassade	Service
Kollektor	Service
Transformator	Service
Wertobjekt	ValueObject
Optimierungskern	Interface

Tabelle 2: Abbildung der Optimierungsarchitektur auf Spring

Transformationsschritt	Beispielhaft: Was geschieht?
Von UML ins Optimierungsmetamodell	Klassen mit Stereotypen wie «Kollektor», «Transformator» usw. werden in Instanzen der gleichnamigen Metaklassen des Optimierungsmetamodells übersetzt.
Vom Optimierungsmetamodell ins Enterprise-Anwendungsmetamodell	Aus Kollektoren, Transformatoren, usw. werden Services.
Vom Enterprise-Anwendungsmetamodell ins Spring-Metamodell	Aus Services werden Spring-Beans.
Von Spring zu 3GL	Aus Spring-Beans werden 3GL-Interfaces, Basisklassen und Implementierungsklassen.
Von 3GL zum Code	Aus den Elementen des 3GL-Modells wird durch sehr einfache Templates der Quellcode erzeugt.

Tabelle 3: Mehrstufige Transformation in AndroMDA

Mehrstufige Generierung

AndroMDA 3, das für das Projekt benutzt wurde, transformiert aus Modellen in den Quellcode. Dafür instanziiert es zunächst das Modell als abstrakten Syntaxbaum im Speicher. Transformationsklassen (so genannte Metafassaden) greifen auf den Syntaxbaum zu und transformieren diesen in handliche Stücke, die von Templates in Code umgesetzt werden können. Dieses Vorgehen ist wesentlich leistungsfähiger als eine direkte Umsetzung vom Modell in den Code, hat sich jedoch bei großen Projekten als immer noch etwas unhandlich herausgestellt. Der Abstand vom Modell zum Code ist einfach zu groß, um ihn in nur einem großen Schritt (sprich: mit einer Transformation in den Metafassaden) zu überspringen.

AndroMDA wird in der Version 4 einen anderen Weg gehen, der sich noch mehr an der MDA-Spezifikation der OMG orientiert. Es transformiert dann in mehreren Stufen, die für Projekte wie KMS so aussehen werden, wie wir es in Tabelle 3 dargestellt haben.

Diese Vorgehensweise macht die Schritte für jede Transformation deutlich kleiner, da nicht mehr so große Unterschiede in den Abstraktionsebenen zu überwinden sind wie vorher. Für Projekte wie KMS ist die Architektur dadurch noch einfacher zu handhaben. Das Vorgehen dieser sukzessiven Verfeinerung über mehrere Metamodelle wurde in [Boh06] bereits ausführlich erläutert, sodass wir hier nicht näher darauf eingehen wollen.

Auf der Ebene des 3GL-Metamodells sind Änderungen des Modells zwischen zwei Generatorläufen leicht zu erkennen. AndroMDA wird durch Differenzbildung zwischen dem alten und dem aktuellen 3GL-Modell Änderungskommandos erzeugen können. Wird zum Beispiel im UML-Eingabemodell eine Klasse umbenannt, sodass sich die Namen dreier Klassen im 3GL-Zwischenmodell ändern, wird AndroMDA dies erkennen und drei Änderungskommandos an Eclipse absetzen – ein Vorgehen, das ohne Modell-zu-Modell-Transformation nicht so leicht machbar gewesen wäre.

Den Zugriff aufs Modell regeln

In allen Projekten ist klar festgelegt, wer zu welchem Zeitpunkt Zugriff auf den Quellcode hat. Das passiert üblicherweise in einem Versionskontrollsystem. In MDA-basierten Projekten gehört das Modell mit zum Quellcode. Daher ist es wichtig, dass das Modell genauso kontrolliert geändert wird wie jeder andere Quellcode auch.

Modelle werden in CASE-Tools bearbeitet. Das Tool ändert Text- oder Binärdateien, die intern recht komplex aufgebaut sind. Entwickler verwenden grundsätzlich das CASE-Tool, um die Modelle zu ändern, niemals einen Texteditor für die Modelldateien. Dies widerspricht in gewissem Maße der Philosophie des im Projekt verwendeten Open-Source-Versionskontrollsystems *Concurrent Versions System (CVS)*.

CVS geht davon aus, dass Entwickler bei Beginn der Arbeit Dateien aus dem System auschecken, dann verändern und wieder einchecken. Beim Auschecken wird keine Sperre gesetzt, andere Entwickler können dieselbe Datei ebenfalls auschecken, verändern und einchecken. Checken zwei Entwickler nacheinander dieselbe Datei ein, so hat der zweite die Aufgabe, eventuelle Konflikte (Änderungen an derselben Zeile in der Datei) von Hand aufzulösen.

Dieses Muster des nicht-exklusiven Auscheckens ist für Modelle ungeeignet, da ein Entwickler beim Einchecken nicht entscheiden kann, welche der Zeilen in den beiden Modelldateien die gültige ist. Das Problem entsteht erst recht beim Einsatz von CASE-Tools, die ihre Modelle in Binärdateien speichern.

Im Projekt haben wir den Zugriff auf das Modell deshalb durch organisatorische Maßnahmen geregelt. Das Modell war in *Packages* unterteilt und es gab klare Richtlinien, welche Gruppe von Entwicklern welche *Packages* im Modell bearbeiten darf. Dies war möglich, da das eingesetzte CASE-Tool „Together Designer 2005" (Borland) alle Modellelemente in separaten Dateien speichert.

Als Verbesserungspotenzial sehen wir hier den Einsatz eines anderen Versionskontrollsystems, das von vornherein beim Auschecken eine Sperre setzt. CVS erlaubt das zwar auch (mit Hilfe des Kommandos `cvs admin`), jedoch ist dieses Kommando in der Bedienung recht unhandlich.

Alternativ dazu könnte man die Nutzung eines speziell auf das eingesetzte UML-Tool zugeschnittenen Team-Servers erwägen. Derzeit gibt es solche Lösungen unter anderem für die Tools „Poseidon/UML" und „MagicDraw/UML". Durch die Nutzung einer solchen proprietären Lösung verliert man jedoch die Möglichkeit, in einem einzigen, gemeinsamen Repository alle Quellcodes versionieren zu können.

Ankopplung per XMI

CASE-Tool und Generator werden beim MDA-Vorgehen üblicherweise per *XML Metadata Interchange (XMI)* gekoppelt. Das CASE-Tool exportiert das Modell in Dateien im XMI-Format, der Generator liest diese ein und generiert den Code.

Es gibt zwei Typen von CASE-Tools: Einige speichern ihr Modell gleich nativ in XMI ab (oder als gezipptes XMI, um Platz auf der Platte zu sparen). Andere Tools spei-

chern in einem proprietären Format, das erst noch als XMI exportiert werden muss.

Werkzeuge vom ersten Typ verhalten sich im Entwicklungsprozess günstiger, da ein separater Schritt für den XMI-Export entfällt. Bei der interaktiven Arbeit auf dem Entwicklerplatz ist das eine reine Zeitfrage. Im Batch-Betrieb bei kontinuierlicher Integration mit CruiseControl ist es wichtig, dass das CASE-Tool auch in dieser Umgebung die XMI-Daten genauso gut exportiert wie im interaktiven Betrieb auf dem Entwicklerplatz. Im vorliegenden Projekt dauern das Starten des XMI-Exporters und das Exportieren des gesamten Modells zusammen zirka 60 bis 80 Sekunden – Zeit, die man besser nutzen könnte.

Die Erfahrung zeigt also, dass für das MDA-Vorgehen besser ein Tool verwendet werden sollte, das seine Modelle gleich in XMI ablegt.

Generatorläufe kurz halten

Codegenerierung aus Modellen wird in Büchern und Beispielen meist so beschrieben, dass Code aus *einem* Modell für *eine* Umgebung generiert wird. Übernimmt man dieses Schema naiv in die Praxis, so führt das zu langen Generatorläufen, da immer der gesamte Code aus dem gesamten Modell erzeugt wird. Im Projekt muss man also dafür sorgen, dass aus mehreren Teilmodellen für mehrere Umgebungen generiert werden kann.

Zunächst zu den Teilmodellen: Es ist notwendig, das Modell zu partitionieren, um beispielsweise immer nur komponentenweise zu generieren. Ein Entwickler generiert dann einmal komplett (um die Umgebung zu haben) und danach jeweils iterativ nur noch für die Komponente, an der er gerade arbeitet. Das spart Zeit.

Im Projekt wurden zwei Applikationsserver verwendet, einer für die Applikation, einer für die mathematischen Optimierungsverfahren. Auf dem ersten Server müssen die Anwendungskomponenten und das Domänenklassenmodell installiert werden und auf dem Optimierungsserver außerdem das Domänenklassenmodell und die Optimierungsverfahren. Die Zuordnung von Teilmodellen zu Umgebungen ist also im allgemeinen Fall *n:m*.

Der Generator muss so konfiguriert werden, dass er Code in wählbar großem Umfang generieren kann:

- alles auf einmal (wichtig für den nächtlichen Build),
- nur bestimmte Teilmodelle oder
- nur bestimmte Umgebungen.

Mit AndroMDA ist das möglich: Es existieren konfigurierbare *Package*-Filter, mit denen gesteuert werden kann, welche Teilmodelle generiert werden sollen. Für welche Umgebungen man generiert, wird über das Build-Skript gesteuert.

Die Erfahrung im Projekt hat gezeigt, dass es sich lohnt, genug Zeit in die Konfiguration einer „schlauen" Build-Umgebung zu investieren, da diese Zeit auf den Entwicklerplätzen vielfach wieder eingespart wird. Eine Generierung nach dem Motto „immer alles" ist hingegen ein Zeitfresser erster Klasse.

Ein weiterer Zeitfresser ist ein schlecht konfigurierter Virenscanner. Heutige Scanner müssen nicht mehr explizit gestartet werden, sondern können Dateien beim Zugriff auf Viren überprüfen. Leider sind die meisten Virenscanner aber so konfiguriert, dass sie ohne Ausnahme jede Datei analysieren – egal, ob es sich nun um eine ausführbare Datei oder um ein simples Textdokument handelt. Bei einem Generatorlauf werden potenziell sehr viele Dateien geschrieben – im hier geschilderten Projekt werden pro Generatorlauf bis zu 1.400 Dateien neu geschrieben und anschließend kompiliert. Durch einen naiv konfigurierten Virenscanner vervielfacht sich die Zeit für diesen Vorgang. Man sollte also alle vom Generator erzeugten Dateitypen im Virenscanner ausschließen.

Weiterentwicklung des Datenbankschemas

In MDA wird in der Regel ein vorwärts gerichteter Entwicklungsansatz verfolgt. Auch Datenbankschemata werden aus UML-Klassenmodellen vorwärts erzeugt. Stand der Technik ist dabei heute größtenteils noch die vollständige, nicht-inkrementelle Generierung des Schemas.

Einzelne Veränderungen an einer Klasse lassen sich noch nicht inkrementell im Schema nachvollziehen. Beispielsweise kann man ein Attribut hinzufügen und bekommt in der Anweisung `CREATE TABLE` des Schemas einfach eine neue Spalte hinzu generiert. Das Problem ist nun, dass dieses Schema nicht mehr auf die bisher im Projekt eingesetzte Testdatenbank passt.

Wir haben im Projekt Excel-Tabellen mit Testdaten gepflegt, die bei jeder Datenbankschema-Änderung zunächst von Hand angepasst wurden. Danach wurde aus den Excel-Tabellen automatisch ein SQL-Skript generiert, das die Daten nun erneut in die Testdatenbank einspielte. Damit war alles wieder konsistent: Modell, Schema und Testdaten.

Testdatenbanken existierten im Projekt in zwei Versionen: eine ältere, recht stabile und eine aktuelle, eher volatile. Entwickler können – je nach aktueller Aufgabe – entscheiden, auf welcher Datenbank sie testen wollen. Die Erfahrung zeigte, dass der Prozess für solche Änderungen genau eingehalten werden muss, damit nicht ein Entwickler irrtümlich mit neuem Code gegen ein altes Schema testet oder umgekehrt.

Werkzeugauswahl

Die Effizienz der Entwicklung wird maßgeblich durch die Wahl der Werkzeuge beeinflusst. Mit ungenügendem Handwerkszeug lassen sich nur schwer gute Ergebnisse erzielen. Das wichtigste Werkzeug in einem MDA-Entwicklungsprozess ist neben dem MDA-Generator das UML-Tool. Man sollte sein UML-Tool also sorgfältig auswählen, wobei vor allem folgende Punkte zu beachten sind:

- gute Unterstützung des XMI-Standards,
- ausgereifte Teamlösung und
- die Möglichkeit, Modelle zu partitionieren.

Die Unterstützung des XMI-Standards ist von zentraler Bedeutung, weil das in XMI gespeicherte Modell die Schnittstelle zwischen UML-Tool und Generator darstellt. Im Projekt hat sich herausgestellt, dass Werkzeuge sich unterschiedlich gut an den XMI-Standard halten. Testen Sie also unbedingt, ob das CASE-Tool Ihrer Wahl mit dem Generator Ihrer Wahl zusammenarbeitet.

Wenn das UML-Tool kein gültiges XMI produziert, muss man gegebenenfalls nachbessern bzw. einen eigenen XMI-Exporter schreiben, so wie es im Projekt geschah.

Weitere Erfahrungen

Hier einige weitere gelernte Lektionen in Kurzform:

- Eine gute Partitionierung des Modells ist essenziell für eine zügige Generierung. Dazu muss das Gesamtmodell fachlich und technisch angemessen in Module aufgeteilt werden.

- Wenn man während des Build-Prozesses erst noch XMI exportieren muss, raubt das kostbare Zeit.
- Modellvalidierung lohnt sich – Modellierungsfehler können zu einem frühen Zeitpunkt abgefangen werden. AndroMDA validiert Modelle zur Codegenerierungszeit; besser wäre noch ein direktes Feedback im UML-Werkzeug, was aber wiederum zu einer engeren Kopplung an dieses führt.
- Ein eigenes Profil im UML-Tool hilft den Entwicklern dabei, das Modell bereits richtig zu erstellen und nicht erst bei der Modellvalidierung auf Fehler aufmerksam gemacht zu werden.
- Die Entwickler müssen in den Gesamtprozess (vom Modell zum Code und zur Datenbank) eingewiesen werden. Das Modell ist der Quellcode – das muss in allen Köpfen verankert sein.
- Man sollte den Entwicklern klar machen, was alles aus den Modellelementen entsteht, d. h. die Abbildung vom plattformunabhängigen auf das plattformspezifische Modell sollte gut dokumentiert sein.
- Die Entwicklungsrechner sollten über ausreichend Hauptspeicher verfügen, um neben der Entwicklungsumgebung und dem UML-Werkzeug auch noch den Generator ausführen zu können. 1 GB Hauptspeicher ist unerlässlich, 2 GB sind ideal.

Resultat

Zu Ende des Projekts besteht das Kombiverkehr Kapazitätsmanagementsystem aus rund 350.000 Zeilen Quellcode, wovon etwa 100.000 von den Entwicklern geschrieben wurden. Der gesamte Architekturrahmen, wie z. B. die Konfigurationsdateien für Spring sowie die Rumpfcodes für Dienste und Datenzugriffsklassen, wurden vom Generator erzeugt. Werteobjekte wurden vollständig vom Generator erzeugt und mussten von den Entwicklern nie von Hand modifiziert werden. Auch die Datenzugriffsschicht wurde größtenteils vom Generator erzeugt: Keine einzige Konfigurationsdatei für das objektrelationale Mapping musste von Hand geschrieben werden, ca. 25% aller *HQL*-Abfragen (*Hibernate Query Language*) konnten vom Generator aus der Struktur der Entitäten abgeleitet werden. Die restlichen Abfragen wurden in HQL von Hand formuliert und im Modell abgelegt. Prinzipiell hätte die Möglichkeit bestanden, diese Abfragen in der *Object Constraint Language (OCL)* zu formulieren und durch die OCL-zu-HQL Transformations-*Cartridge* des Generators in HQL umwandeln zu lassen. Diese Idee wurde jedoch fallen gelassen, um die Lernkurve für die Entwickler nicht unnötig zu erhöhen. Insgesamt besteht das System aus knapp 170 Entitäten und 90 Diensten.

Fazit

Das von uns gewählte modellgetriebene Vorgehen hat eine zügige Entwicklung der Software bei gleich bleibend hoher Qualität ermöglicht. Durch die mittels MDA herbeigeführte Abstraktion konnten wir die Komplexität des Vorhabens gut eingrenzen. Die Entwickler konnten sich voll und ganz auf die Implementierung der Geschäftslogik konzentrieren und brauchten sich nicht mit den Detailaspekten der technischen Architektur des Systems auseinanderzusetzen. Sie konnten somit den Vorteil genießen, dass das Know-how der Architekten in Form des des konfigurierten Generators auf allen Entwicklerplätzen präsent war. Gleichzeitig konnten die Architekten sich jederzeit darauf verlassen, dass die von ihnen vorgegebene Architektur eingehalten wurde, da der architektonische Code einzig und allein durch den Generator erzeugt wurde.

Die Entwicklung der MDA-Werkzeuge geht weiter in Richtung höherer Abstraktion und besserer Benutzbarkeit. Zukünftige Projekte werden davon noch weiter profitieren und komplexere Aufgaben bewältigen können, als dies heute bereits möglich ist. Die Erfahrungen aus dem vorliegenden Projekt haben gezeigt, dass Unternehmen heute vom Einsatz der MDA profitieren können und dass der Einsatz dieser Technologie kein Risiko darstellt, sondern neue Chancen eröffnet. ■

Literatur & Links

[And] AndroMDA, Open Source Code Generator Framework, siehe: www.andromda.org

[Boh06] M. Bohlen, QVT und Multi-Metamodell-Transformationen in MDA, in: OBJEKTspektrum 2/06

[Che] Checkstyle 4.1, siehe: checkstyle.sourceforge.net

[Fin] FindBugs, Find Bugs in Java Programms, siehe: findbugs.sourceforge.net

[Kom] Kombiverkehr Homepage, siehe: www.kombiverkehr.de

[PMD] PMD, siehe: pmd.sourceforge.net

[Sei97] C. Seidelmann, Der Kombinierte Verkehr – Ein Überblick, in: Internationales Verkehrswesen (49) 6/1997 S. 321-324 (siehe: www.sgkv.de/deutsch/KV-Grundsatzartikel.pdf)

[Spr] Spring Framework, siehe: www.springframework.org