Grundlegende Konzepte und Einordnung der Model Driven Architecture (MDA)

von Peter Roßbach, Thomas Stahl, Wolfgang Neuhaus

Model Driven Architecture

Model Driven Architecture (MDA) – ein Begriff, der sich zunehmend in der Fachpresse unter gänzlich unterschiedlichen Interpretationen wiederfindet. Schenkt man den vielen Versprechungen um MDA Glauben, so lassen sich aus technologieunabhängigen, einheitlichen Modellen problemlos Anwendungen für verschiedenste Plattformen wie J2EE, CORBA oder .NET automatisch generieren. Mit dem Titelthema in dieser Ausgabe versuchen wir, etwas Licht in das Dunkel zu bringen. Dazu werden wir in diesem Artikel Ziele und Potenziale beschreiben, die wichtigsten Konzepte von MDA aufzeigen und eine Abgrenzung und Einordnung vornehmen. Die weiteren Artikel diskutieren eine heute nutzbare Interpretation des OMG-Standards und zeigen Möglichkeiten und Grenzen auf.

Ziele und Potenziale

MDA ist ein junger Standard der Object Management Group (OMG), welche 1989 gegründet wurde und heute ein offenes Konsortium aus ca. 800 Firmen weltweit ist. Die OMG erstellt herstellerneutrale Spezifikationen zur Verbesserung der Interoperabilität und Portabilität von Softwaresystemen. Bekannte Ergebnisse sind CORBA (Common Object Request Broker Architecture), IDL, UML (Unified Modeling Language), XMI oder MOF (Meta Object Facility). Das neue Flaggschiff der OMG heißt MDA [1]. Mit MDA soll eine erhebliche Steigerung der Entwicklungsgeschwindigkeit erreicht werden. Das Mittel dazu heißt „Automation durch Formalisierung". Aus formal eindeutigen Modellen soll durch Generatoren automatisch Code erzeugt werden. Durch den Einsatz der Generatoren und die formal eindeutig definierten Modellierungssprachen soll darüber hinaus die Softwarequalität gesteigert werden. Fehler in den generierten Codeanteilen können an einer Stelle – in den Generatorschablonen – beseitigt werden. Die Qualität des generierten Sourcecodes ist gleichbleibend, was zu einem höheren Grad der Wiederverwendung führen soll.

Ein weiteres wesentliches Ziel ist die bessere Handhabbarkeit von Komplexität durch Abstraktion. Mit den Modellierungssprachen soll „Programmierung" auf einer abstrakteren Ebene möglich werden, die klare Trennung von fachlichen und technischen Anteilen zielt auf eine bessere Wartbarkeit durch Trennung von Verantwortlichkeiten ab. Die abstraktere, technologieunabhängige Beschreibung von Schlüsselkonzepten mit eindeutigen Modellierungssprachen verspricht eine verbesserte Handhabbarkeit des Technologiewandels. Und nicht zuletzt soll eine verbesserte Interoperabilität durch Standardisierung erreicht werden.

Grundsätzlicher Ansatz

Mit welchem Ansatz will die OMG diese hehren Ziele erreichen? Fachliche Spezifikationen werden in pattformunabhängigen Modellen (PIM = Platform Independent Model) definiert. Dazu wird eine formal eindeutige Modellierungssprache – eine erweiterte UML-Notation (UML-Profil) – verwendet. Die damit spezifizierte Fachlichkeit ist vollständig unabhängig von der späteren Zielplattform. Plattformen können zum Beispiel CORBA, J2EE oder .NET sein. Durch eine in der Regel mit Werkzeugen automatisierte Modelltransformation werden aus den plattformunabhängigen fachlichen Spezifikationen plattformabhängige Modelle (PSM = Platform Specific Model) gewonnen. Diese plattformabhängigen Modelle enthal-

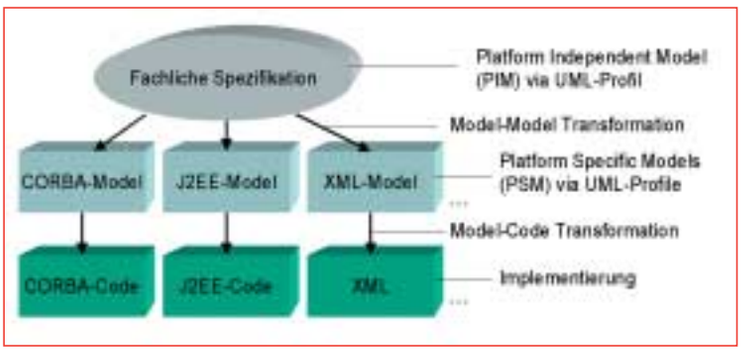

Abb. 1: MDA Grundprinzip

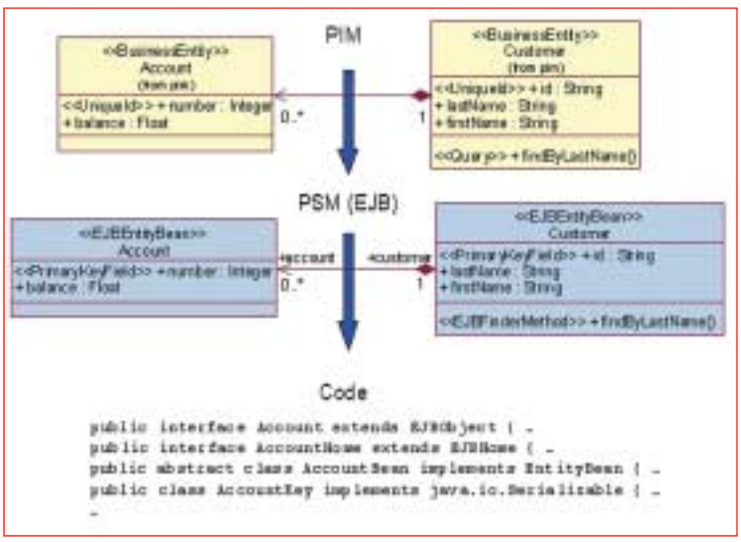

Abb. 2: Ausschnitt aus einem PIM

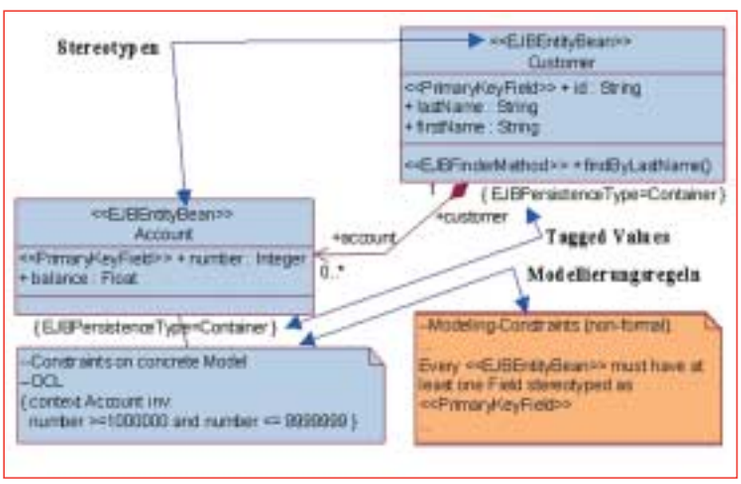

Abb. 3: Verwendung eines UML-Profils

ten die spezifischen Konzepte der Zielplattform. Mit einer weiteren werkzeugunterstützten Transformation auf der Basis eines PSM wird dann für eine konkrete Zielplattform die Implementierung gewonnen (Abb. 1). Die Vision der OMG geht allerdings noch einen Schritt weiter: Plattformspezifische Modelle sollen ausführbar werden, sodass die Code-Generierung vollständig entfallen würde. Dazu müssten Compiler oder Interpreter für Plattformen existieren, die plattformspezifische Modelle ausführbar machen. Solche Compiler oder Interpreter sind heute nur in Nischenbereichen vorhanden bzw. in der Diskussion, wie beispielsweise bei der Entwicklung von Embedded-Systemen [2], [3].

Wichtig: PIM und PSM sind immer relativ zu einer Plattform definiert. Damit kann ein Modell beispielsweise spezifisch für die Plattform J2EE sein, jedoch unabhängig von der Plattform BEA Weblogic Server.

In Abbildung 2 ist ein kleiner Ausschnitt aus einem PIM (Platform Independent Model) zu sehen. Dargestellt ist ein Klassenmodell mit zwei fachlichen Klassen: *Customer* und *Account*. Beide Klassen sind mit dem Stereotyp <<*BusinessEntity*>> ausgezeichnet. Weiterhin besitzen beide ein Attribut, das mit dem Stereotyp <<*UniqueId*>> ausgezeichnet ist. Zusätzlich finden wir beim Customer noch eine Methode *findByLastName*, die mit dem Stereotyp <<*Query*>> versehen ist. Die Erweiterung des Standardsprachumfangs der UML durch Stereotypen ist ein Standardmechanismus, der zu diesem Zweck von der OMG spezifiziert wurde. Hier wird er genutzt, um eine formal eindeutige Modellierungssprache zu definieren. Erst diese Eindeutigkeit macht ein UML-Modell zu einem MDA-Modell. Die im PIM verwendeten Konzepte <<*BusinessEntity*>>, <<*UniqueId*>> und <<*Query*>> sind vollständig unabhängig von der Zielplattform. Diese Abhängigkeit entsteht erst durch die Transformation des PIM auf das PSM. Hier finden wir die für J2EE spezifischen Konzepte <<*EJBEntityBean*>>, <<*PrimaryKeyField*>> und <<*EJBFinderMethod*>>. Durch die dann folgende Transformation wird aus dem PSM letztlich Sourcecode, in dem sich die aufgeführten Konzepte dann in ihrer konkreten Ausprägung wiederfinden.

MDA-Konzepte im Überblick

Modell: Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems. MDA-Modelle werden in der Regel in UML definiert. Aber auch klassische Programmiersprachen sind MDA-Modellierungssprachen. Ihre Programme sind MDA-Modelle. Diese sind wiederum unabhängig von der darunter liegenden Plattform – beispielsweise Bytecode oder Maschinencode. UML-Diagramme sind jedoch nicht per se MDA-Modelle. Der wichtigste Unterschied zwischen allgemeinen UML-Diagrammen (z.B. Analyse-Modellen) und MDA-Modellen liegt darin, dass die Bedeutung von MDA-Modellen formal definiert ist. Dies wird

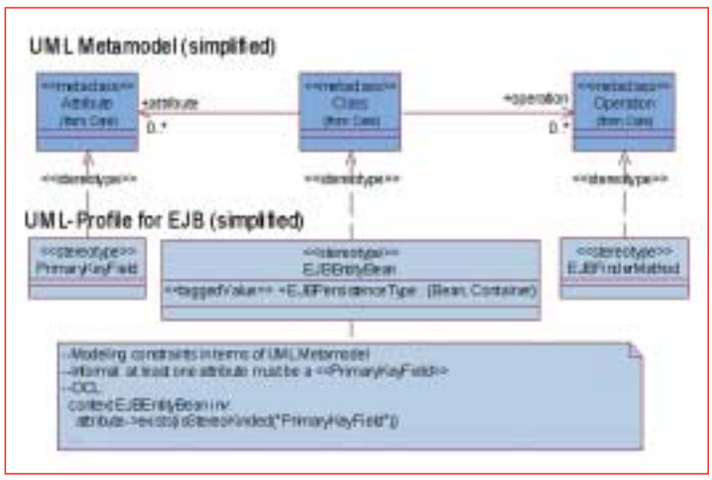

Abb. 4: UML-Metamodell und UML-Profil für EJB (Ausschnitt)

durch die Verwendung einer formal eindeutigen Modellierungssprache sicher gestellt, die in einem UML-Profil (s.u.) festgelegt wird. Damit wird erreicht, dass das erzeugte Modell eindeutig durch Generatoren auf ein anderes Modell oder auf Sourcecode abgebildet bzw. tranformiert werden kann.

Plattform: MDA sagt (zunächst) nichts über den Abstraktionsgrad von Plattformen aus. Plattformen können aufeinander aufbauen. Beispielsweise ist ein Intel PC eine Plattform für Linux. CORBA, J2EE oder Web Services sind mögliche Plattformen für ein eBusiness-System und C++ ist eine mögliche Plattform für CORBA. Auch eine wohldefinierte Anwendungsarchitektur mit dazugehörigem Laufzeitsystem kann eine Plattform für Applikationen sein.

UML-Profile: UML-Profile sind der Standardmechanismus zur Erweiterung des Sprachumfangs der UML. Sie enthalten Sprachkonzepte, die über Basis-UML-Konstrukte wie Klassen und Assoziationen, Stereotypen, Tagged Values und Modellierungsregeln (Constraints) festgelegt werden (Abb. 3).

Definiert wird ein UML-Profil als Erweiterung des UML-Metamodells. Dieser Zusammenhang wird in Abbildung 4 anhand eines vereinfachten Beispiels – ein UML-Profil für Enterprise JavaBeans (EJB) – verdeutlicht.

Die Standard-UML-Konzepte *Attribute, Class* und *Operation* werden im UML-Profil um die spezifischen Konzepte *PrimaryKeyField*, *EJBEntityBean* und *EJBFinderMethod* erweitert. Zusätzliche Erweiterungen werden durch Tagged Values und Modellierungsrichtlinien in Form von Constraints definiert. UML-Profile sind also Metamodelle und definieren formale Modellierungssprachen als Erweiterung der UML.

PIM und PSM: Die Trennung von Platform Independent Model (PIM) und Platform Specific Model (PSM) ist ein Schlüsselkonzept von MDA im Sinne der OMG. Der Hintergrund: Konzepte sind stabiler als Technologien und formale Modelle besitzen Potenzial für automatisierte Transformationen. Das PIM abstrahiert von technologischen Details, während das PSM die Konzepte einer Plattform verwendet, um ein System zu beschreiben (Abb. 5). Der „Rückweg" – die Gewinnung eines PIMs aus einem PSM ist kaum automatisierbar. Dieser Weg erfordert manuelle, intellektuelle Arbeit in Form eines Refactorings, da mit dem PIM eine echte Abstraktion erreicht wird. Somit legt MDA einen Forward Engineering Prozess nahe und Tool-unterstütztes Roundtrip Engineering ist kaum möglich.

Transformationen: Transformationen bilden Modelle auf die jeweils nächste Ebene, weitere Modelle oder Sourcecode ab. Im Sinne von MDA müssen Transformationen flexibel und formal auf Basis eines gegebenen Profils definiert werden können. Dies ist eine Voraussetzung für die gewünschte Automatisierung der Transformation über Generatoren. Zurzeit existiert allerdings noch keine OMG-Spezifikation für eine einheitliche Transformationssprache. Es liegt jedoch ein Re-

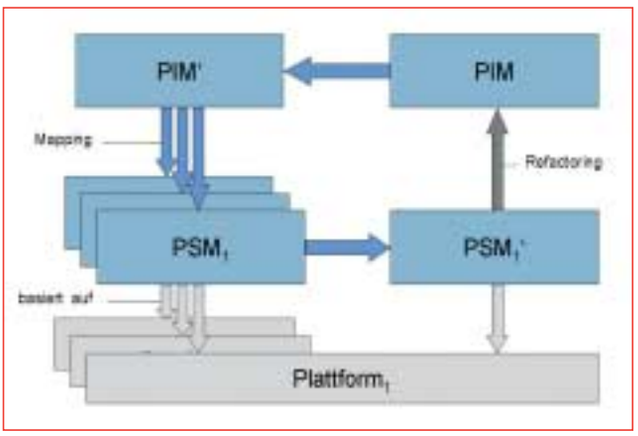

Abb. 5: Zusammenhang PIM – PSM – Plattform

quest für ein Proposal dazu vor („MOF 2.0 Query/Views/Transformations RFP", OMG Document ad/2002-04-10). Darin soll eine Transformation zwischen Metamodellen beschrieben werden. Metamodelle können in UML-Profilen definiert sein. Nehmen wir an, uns läge ein UML-Profil für EJB und ein UML-Profil für BEA WebLogic Server vor. Die Transformation würde dann eine Abbildung zwischen den Modellierungssprachen in diesen beiden UML-Profilen beschreiben. Als Resultat könnten dann Modelle aus der einen Modellierungssprache in die andere überführt werden.

Heutige Generatoren lösen dieses Problem auf andere Weise, indem sie proprietäre Transformationssprachen verwenden. Hier kommen beispielsweise jPhyton, TCL, JSP, XSLT oder zweckoptimierten Skriptsprachen zum Einsatz. Die mit diesen Sprachen definierten Generatorschablonen funktionieren nach dem Prinzip von Makros und verwenden die Modelle als Eingabeparameter. Derzeit ist demzufolge (noch) keine Interoperabilität für Modell-Transformationen gegeben.

Abgrenzung und Einordnung

An dieser Stelle fragen Sie sich möglicherweise, wo denn nun der „bahnbrechende" Unterschied zu bisherigen Ansätzen der Code-Generierung liegt. Daher hier eine kurze Abgrenzung: MDA ist kein CASE-Ansatz. Klassische CASE-Tools zeichnen sich dadurch aus, dass häufig sowohl Metamodell wie auch Transformation fix sind. Darüber hinaus besitzen sie in der Regel proprietäre Modellierungssprachen. Damit fehlt die angestrebte Interoperabilität und insbesondere lassen sich Modellierungssprache und der generierte Anteil nicht auf projektspezifische Bedürfnisse anpassen.

MDA ist keine 4GL. Ähnlich wie bei den CASE-Tools sind bei den 4GL-Tools häufig Transformation und Metamodell fest. Zusätzlich beruhen sie auf festen Laufzeitsystemen, die durch den Entwickler nicht modifiziert werden können. MDA macht hier dem gegenüber keine Einschränkungen.

Und MDA ist kein Code-Wizard oder Pattern-Expander. Jeder kennt sie: Die nützlichen Code-Wizards in Entwicklungsumgebungen oder die Pattern-Expansion in den UML-Tools. Sie erlauben die automatische Erzeugung eines Klassenrumpfes beispielsweise eines Entity Beans bzw. die Erzeugung mehrerer Klassenrümpfe bei der Pattern-Expansion. Dieser Schritt ist im Unterschied zu MDA jedoch einmalig. Die mit MDA erreichbare wie- derholte Generierung bei Erhaltung der vorgenommenen Individualisierungen fehlt. Außerdem geht die Abstraktion „verloren", die mit MDA im Modell vorgenommen wird.

Praktische Interpretation

Soweit die „reine OMG Lehre". Damit MDA praktisch anwendbar wird, muss man diesen jungen OMG-Standard allerdings in geeigneter Weise interpretieren und entsprechende Werkzeuge einsetzen, die zu der Interpretation passen. Heute existieren im Markt unterschiedliche Interpretationen, sowohl den MDA/D-Prozess wie auch die dazu gehörigen Werkzeuge betreffend. Die weiteren Artikel zum MDA-Titelthema vertiefen eine mögliche Interpretation, die durch die Mitglieder der Arbeitsgemeinschaft Architecture Management Group [4] diskutiert und unterstützt wird. Als Richtschnur dazu dient der *Generative Development Process GDP* [5]. Dieser Prozessleitfaden ist aus der Praxis zahlreicher MDA-Projekte bei der b+m Informatik AG [6] entstanden und berücksichtigt die besonderen Bedürfnisse beim praktischen Einsatz von MDA.

Der erste Artikel wird dazu den GDP und die im Prozess benötigten Rollen beschreiben [7]. Der zweite Artikel beschäftigt sich mit der „frühen Phase" in einem MDA-Projekt – dem Übergang zwischen Geschäftsprozessmodellierung und objektorientierter Analyse. Hier erfahren Sie, welche Möglichkeiten durch den Einsatz der UML-Notation zur Geschäftsprozessmodellierung erschlossen werden können und wie dies in MDA-Projekten genutzt werden kann [8]. Der dritte Artikel betrachtet die besonderen Aspekte aus der Entwicklersicht [9]. In der nächsten Ausgabe des *Java Magazins* beleuchten wir dann ein MDA-Projekt aus Architektursicht, denn wie der Name „Model Driven Architect" schon vermuten lässt, kommt der Architektur eine besondere Bedeutung zu [10]. Und schließlich nimmt ein fünfter Artikel zum Themenkomplex MDA eine Einordnung in weitere generative und generische Techniken vor [11].

Links & Literatur

[1] MDA Specifications, 2003, www.omg.org/mda/

[2] Statemate – ausführbare UML-Diagramme für Embedded Systems, 2003, www.ilogix.com/products/magnum/index.cfm

[3] M. Völter: „A Generative Component Infrastructure for Embedded Systems", 2003, www.voelter.de/data/pub/SmallComponents.pdf

[4] Achitecture Management Group: www.a-m-group.de/

[5] Generative Development Process, 2003, www.architectureware.de/

[6] b+m Informatik AG: www.bmiag.de/

[7] W. Neuhaus, C. Robitzki: „Eine praktische Interpretation", *Java Magazin* 9.2003

[8] T. Weilkiens: „Vom Modell zum Code – ein kurzer Weg mit MDA", *Java Magazin* 9.2003

[9] M. Schepe, W. Neuhaus, P. Roßbach: „Praktische Entwicklung mit MDA/D", *Java Magazin* 9.2003

[10] T. Stahl, M. Schepe, C. Robitzki: „Model Driven Architect", *Java Magazin* (noch nicht veröffentlicht)

[11] M. Völter: „Codegenerierung – Ein Überblick", *Java Magazin* (noch nicht veröffentlicht)

Der MDA-Standard muss in der Praxis geeignet interpretiert werden, um anwendbar zu sein

von Wolfgang Neuhaus und Carsten Robitzki

Eine praktische Interpretation

Ob sich die aktuell vorliegenden Spezifikationen zu MDA „in Reinform“ als Standard durchsetzen, ist fraglich [1]. Denn dazu lassen sie noch zu viele Punkte offen und verfolgen vielleicht auch nicht immer den richtigen Weg. Wir wollen mit diesem Artikel aufzeigen, wie dennoch ein praxistauglicher MDA/D-Entwicklungsprozess auf dieser Basis gestaltet werden kann. Denn wir glauben, dass generative Softwareentwicklung auf Basis der MDA-Konzepte eine kommende Schlüsseltechnologie ist. Und ihre Einführung wird in der Softwareentwicklung eine ähnliche Bedeutung erlangen, wie die Einführung der Hochsprachen als Antwort auf die zunehmende Komplexität der Assemblerprogramme.

Eine gewagte Aussage, finden Sie? Natürlich. Aber vielleicht können Sie diese These besser nachvollziehen, wenn Sie das Titelthema in dieser Ausgabe gelesen haben.

Soll der MDA-Standard in der Praxis angewendet werden, so stellt man schnell fest, dass es heute kaum praxistaugliche Ansätze gibt, die die OMG-Vision bereits vollständig umsetzen [2]. Ausführbare Modelle erfordern entsprechende Compiler oder Interpreter für die jeweilige Plattform, die heute kaum zu finden sind, und auch die Transformation von Metamodellen hat in der Praxis bislang keine Bedeutung. Die Diskussion, die wir in der Praxis erleben, dreht sich vielmehr um Ansätze, die einen pragmatischeren Weg gehen: Aus einem plattformunabhängigen Modell wird unmittelbar Sourcecode generiert. Dieser Sourcecode enthält alle schematischen Elemente und wird mit einem Programmeditor oder einer Entwicklungsumgebung um individuellen fachlichen Code ergänzt. Für die Transformation werden häufig Schablonen verwendet, die das Modell als Eingangsparameter erhalten und in Skriptsprachen wie TCL, JPhyton, XSLT oder eigens dafür entwickelten Spezialsprachen geschrieben sind [3]. Aber diese Mittel lassen sich nur effektiv einsetzen, wenn auch der Entwicklungsprozess adäquat angepasst wird. Gerade dazu schweigt sich die OMG jedoch aus. Unsere Erfahrungen mit der von uns gewählten Interpretation wollen wir im Folgenden schildern.

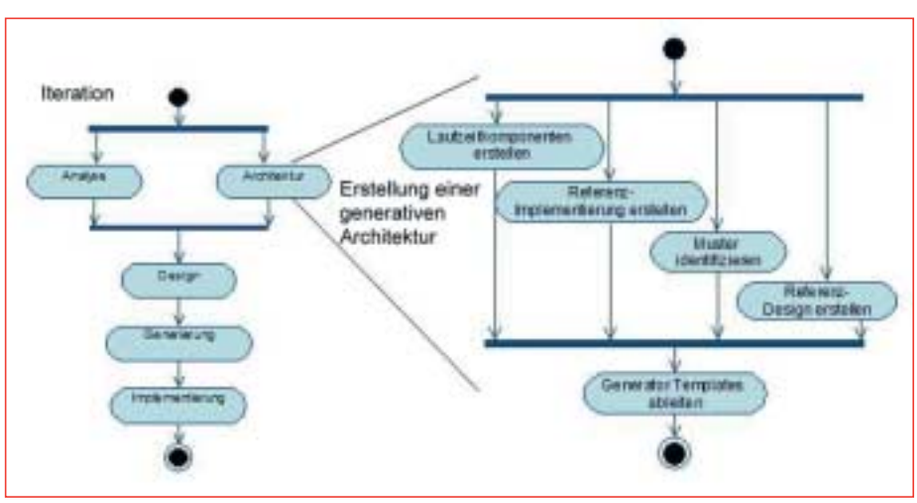

Abb. 1: Generative Development Process (GDP)

Praktische Interpretation

Als Richtschnur dazu dient uns der Generative Development Process GDP (Abb. 1). Dieser Prozessleitfaden ist aus der Praxis zahlreicher MDA-Projekte bei der b+m Informatik AG entstanden und berücksichtigt die besonderen Bedürfnisse beim praktischen Einsatz von MDA [4].

Der GDP ist nicht als Konkurrenz zu etablierten OO-Entwicklungsprozessen zu sehen, sondern als Verfeinerung im Hinblick auf die Anwendung architekturzentrierter und generativer Techniken. Eine der wesentlichen Eigenschaften des GDP ist, dass nicht generative aber bzgl. ihres Generats starre Entwicklungsumgebungen verwendet oder gar vorausgesetzt werden, sondern beliebige Ziel-Architekturen, Sprachen, Schnittstellen und Laufzeitkomponenten unterstützt werden können. Die Praxis zeigt, dass ein generativer Ansatz nur auf Basis spezifischer und durchgängig ausgearbeiteten Architekturen Sinn macht. Dies bedeutet die explizite Abkehr von „One-Fits-All" CASE-Ansätzen hin zu generativen Ansätzen, die individuelle architekturelle Anforderungen unterstützen. Es steht also klar der Entwicklungsprozess und nicht eine integrierte Entwicklungsumgebung (CASE-Tool / IDE) im Vordergrund.

Mit anderen Worten: Es wird nichts generiert, was nicht vorher in Form einer Referenzimplementierung verifiziert wurde. Damit erübrigen sich insbesondere die mit generativen Ansätzen häufig gestellten Fragen wie z.B.: Wie hoch ist die Laufzeitperformance des Generats? Welche Qualität/Lesbarkeit hat der generierte Sourcecode? Das Generat bzw. der Sourcecode ist genauso gut (oder schlecht) wie die Referenzimplementierung, aus der man die Transformation ableitet. Das fundierte Spezialwissen von Software-Architekten wird im GDP parallel zur Analyse in Form einer generativen Architektur erfasst [5] und so einer breiten Schicht von Anwendungsentwicklern zur Verfügung gestellt [6]. Diese generative Architektur besteht aus der Definition einer Designsprache (UML-Profil), einer Anzahl von Generator-Schablonen (Transformation) und ggf. Laufzeitkomponenten wie Frameworks oder Bibliotheken. Generiert werden alle schematischen Anteile einer Anwendung. Spezielle Fachlogik wird nach wie vor individuell in der Zielsprache in geschützten Code-Bereichen programmiert, die bei mehrfacher Generierung erhalten bleiben.

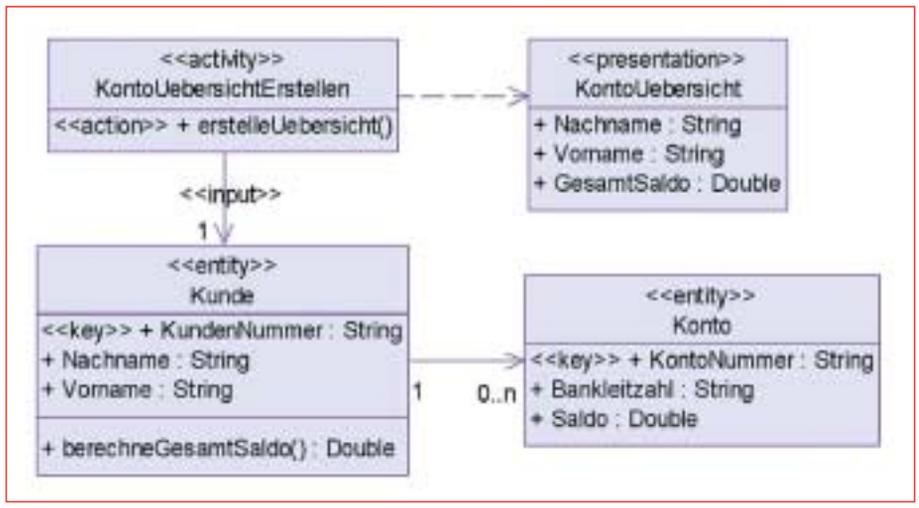

Abb. 2: Architekturzentriertes Design

Architekturzentriertes Design

Die definierte Designsprache enthält die Architekturkonzepte der Anwendungsfamilie in einer plattformunabhängigen Abstraktion. Mit dieser Designsprache erstellen die Designer das Anwendungsdesign in Form von PIMs. Auf die Trans- formation dieser PIMs in plattformabhängige UML-Modelle (PSMs) wird im Unterschied zur reinen OMG-Lehre bewusst verzichtet. Ein PSM stellt einen Zwischenschritt der Transformation vom PIM zum Sourcecode dar und wird nur benötigt, wenn inhaltlich in diese Transformation eingegriffen werden muss. Aber wenn dies geschieht, entstehen die üblichen Konsistenzprobleme, wie sie beim klassischen Roundtrip Engineering auftreten: Änderungen lassen sich nicht automatisch in abstraktere Modelle (PIM) zurück propagieren. Weiterhin sind manuelle Änderungen auf der PSM-Ebene aufwändig, zumal diese Modelle in der Regel bereits deutlich expandiert (näher am Sourcecode) sind. Bei einer direkten Transformation kann dieser Aufwand vermieden werden. Gegenüber der Variante mit PSM findet hier eine bewusste Beschränkung aber damit verbunden auch eine deutliche Vereinfachung und Optimierung statt. Die Praxis

Besonderheiten unserer MDA-Interpretation

Architekturzentriertes Design: Wir verzichten auf den Einsatz plattformspezifischer Modelle. Stattdessen setzen wir auf plattformunabhängige Modelle in architekturzentriertem Design. Diese Beschränkung auf der einen Seite führt zu einer deutlichen Optimierung auf der anderen Seite. Pflegeaufwände werden reduziert und Konsistenzprobleme vermieden.

Forward Engineering: Der Verzicht auf Roundtrip Engineering in unserer Interpretation geschieht bewusst. Da in unseren Modellen echte Abstraktionen vorgenommen werden, ist ein Reverse Engineering nicht sinnvoll.

Transformation mit Schablonen: Die Beschreibung von Abbildungen zwischen Metamodellen als Vorschrift für Modelltransformationen, wie sie derzeit bei der OMG diskutiert wird, halten wir nicht für praxistauglich. Der Einsatz von Generatorschablonen hingegen hat sich in der Praxis bewährt und kann leicht verbreitet werden, da der Mechanismus vielen aus der Entwicklung von Code-Wizards oder Makros vertraut ist.

Keine 100%-Generierung: Wir generieren in der Regel nicht 100% einer Anwendung und halten dies auch nur in Ausnahmefällen für möglich und sinnvoll. Generiert werden die schematischen bzw. technischen Aspekte einer Anwendung. Die individuellen bzw. fachlichen Aspekte werden in der Zielsprache „manuell" ergänzt.

in bisherigen Projekten hat gezeigt, dass diese Vereinfachung in der Regel einen höheren Nutzen hat, als die zusätzlichen Freiheitsgrade beim Einsatz von PSMs. Betrachten wir ein Beispiel eines PIM (Abb. 2). Das Modell verrät nichts über verwendete Basistechnologien – die technologische Umsetzung solcher Modelle ist erst im Kontext einer konkreten Architekturbindung definiert. Durch die semantische Anreicherung des Modells mit Stereotypen, tagged Values und Constraints entsteht eine formale UML-Designsprache. Im GDP liegt das Abstraktionsniveau dieser Sprache auf der Ebene der Architekturkonzepte, weshalb wir auch von architekturzentriertem Design sprechen.

Die fachliche Bedeutung des Diagramms in Abbildung 2 ist nahezu offensichtlich: Im Zentrum steht eine Aktivität – ein Baustein für übergeordnete Prozessmodelle, die in der Lage ist, eine Aktion zur Erstellung einer kundenspezifischen Konto-Übersicht auszuführen. Die Kunden-Entität wird der Aktivität als Eingabe vermittelt. Erstere besitzt neben zwei fachlichen Attributen ein identifizierendes Merkmal und ist selbst in der Lage, den Gesamtsaldo zu berechnen, in dem sie die Saldi der assoziierten Konten aufaddiert. Die Aktivität bzw. ihre Aktion nutzt eine Präsentation mit drei fachlichen Attributen zur Darstellung des Ergebnisses. Ein Standard Java-Code-Generator würde die annotierten Stereotypen ignorieren und die Signaturen vier einfacher Java-Klassen erzeugen. Im GDP wird die programmiersprachliche Umsetzung der Designsprache und damit der konkreten Modelle architekturspezifisch durchgeführt. Dazu zwei Beispiele:

In einer EJB-basierten Architektur mit HTML-Clients: Activity-Klassen sind Stateless Session Beans, die die Schnittstellen einer serverseitigen Ablaufsteuerung implementieren. Jede *action* ist deklarativ transaktional. Die *entity*-Klassen sind Entity Beans mit entsprechenden Remote-Interfaces. Attribute des Typs *key* bilden die PK-Klassen. Für öffentliche Attribute gibt es *Getter*- und *Setter*-Methoden. Es wird Container Managed Persistence (CMP) für die Persistenz verwendet. Die Deskriptoren für ein spezifisches objektrelationales Mapping-Tool lassen sich aus dem Modell ableiten. Für Assoziationen gibt es Zugriffsmethoden, die sich auf *Finder*-Methoden der assoziierten Klasse stützen. Die *presentation*-Klassen spezifizieren JSP-Models, mit deren Hilfe JSP/HTML-Seiten befüllt werden. Jede *presentation* ist mit einem spezifischen Request-Analyser (Web-Framework) gekoppelt.

In einer C++/CORBA-basierten Client-Server-Architektur: Zu jeder *activity*-Klasse gibt es ein IDL-Interface. Alle Attribut- und Parameter-Typen im Design werden auf entsprechende IDL-Typen abgebildet.

Es gibt ein entsprechendes C++-Skeleton. Die *activitiy*-Klassen implementieren die Schnittstellen zu einem spezifischen Workflow-System. Aktionen (*action*-Operationen) sind Transaktionen auf einem Objekt Transaktions-Monitor (OTM). Alle *entity*-Klassen sind nicht verteilbare C++-Klassen, die Instanzen werden mittels objektrelationalem Mapping in einem RDBMS abgelegt. Attribute des Typs *key* dienen als Primärschlüssel. Die *presentation*-Klassen sind Swing-GUIs, welche die Schnittstellen eines spezifischen Client-Frameworks implementieren.

Anhand dieser beiden einfachen Beispiele lassen sich gut die wesentlichen Vorteile des Ansatzes erkennen: Modelle in architekturzentriertem Design sind kompakt, ausreichend informationsreich und enthalten noch keine Architekturbindung. Sie sind dadurch übersichtlicher und leichter zu warten. Außerdem eignen sie sich besser zur Diskussion mit anderen Projektbeteiligten, da technische Details abstrahiert sind. Durch den Verzicht auf Roundtrip Engineering ist das Modell immer konsistent zum Sourcecode.

Rollen in unserem MDA/D-Entwicklungsprozess

Durch die Aufteilung des Prozesses in Architektur- und Entwicklungsstrang und damit in die Implementierung der generativen Architektur und in die Implementierung der eigentlichen Anwendung mit Hilfe der generativen Architektur entstehen unterschiedliche Rollen im Projekt. Diese können (müssen aber nicht) je nach Projektgröße, -organisation und Skill der Projektmitglieder von unterschiedlichen Personen besetzt werden. Die Architekten definieren im Architekturstrang die Architektur der Anwendung und erarbeiten die generative Architektur. Außerdem sind sie während des gesamten Projekts für die Pflege und Weiterentwicklung aller Bestandteile der generativen Architektur verantwortlich, denn das Feedback der Entwickler führt meistens (je nach Reifegrad der generativen Architektur und speziellen architektonischen Anforderungen der Projekte) zu notwendigen Änderungen bzw. Erweiterungen der generativen Architektur [5]. Die Designer erstellen auf Basis der Designsprache der generativen Architektur das Design der zu entwickelnden Anwendung [7]. Die Entwickler können sich auf die Umsetzung der spezifischen Fachlogik in den geschützten Code- Bereichen konzentrieren, da der schematische/technische Code automatisch generiert wird [6]. In der Praxis werden die Rollen Designer und Entwickler häufig von den gleichen Personen besetzt. Bei der Einführung generativer Softwareentwicklung kann es jedoch auch sinnvoll sein, diese Rollen personell zu trennen, wenn die Entwickler zunächst noch an die Arbeit mit UML-Werkzeugen herangeführt werden müssen.

Einfacher designen ohne PSM

Fazit

MDA ist heute praktisch einsetzbar, wenn man den OMG-Standard geeignet interpretiert. Es entstehen keine bis geringe Abhängigkeiten zu Toolherstellern (solange sie Standards wie z.B. XMI als Austauschformat für Modelle unterstützen), da bis auf die Transformation (aufgrund der noch nicht vorhandenen Standardisierung) die Interoperabilität gegeben ist. Damit lassen sich die aufgeführten Potenziale (in unterschiedlichem Ausmaß) bereits heute mit MDA erreichen. Erstmals wird eine echte Kontrolle von Architekturänderungen im Projekt möglich. Entwickler und Designer stellen notwendige Architekturänderungen fest, da die Architektur exakt definiert ist. Damit unterstützt der GDP die Durchgängigkeit der Architektur und die Nachvollziehbarkeit von architektonischen Änderungen. Dies bedeutet, dass auch am Ende eines Projekts sämtliche Details der Architektur bekannt sind. Und das ist leider heute noch keine Selbstverständlichkeit! Schematische Anteile werden hundertprozentig sicher umgesetzt und auch die Änderungen an der Architektur fließen mit gleicher Sicherheit sofort in die Umsetzung ein. In heutigen eBusiness-Anwendungen mit mehrschichtigen Architekturen können nach unserer Erfahrung ca. 60% Lines of Code automatisch generiert werden. Zusätzlich wird das Bewusstsein für architekturbezogene Themen im Team gestärkt und das Wissen über die Architektur nimmt zu, da der MDA-Ansatz eine explizite Auseinandersetzung mit architektonischen Aspekten erzwingt. Technische und fachliche Aspekte sind eindeutig voneinander getrennt und identifizierbar. Um diese Vorteile zu erreichen, ist es allerdings notwendig, den Entwicklungsprozess zur Berücksichtigung der spezifischen „MDA-Bedürfnisse“ anzupassen.

Ausblick

Wir konzentrieren uns hier auf architekturzentrierte Software-Entwicklung mit entsprechenden UML-Profilen. Das bedeutet, die Domäne unserer Modellierungssprache ist „Architektur für Business-Software“. Man kann aber auch für fachliche Domänen mit dem gleichen Vorgehen eine generative Unterstützung durch entsprechende domänenspezifische Modellierungssprachen erreichen. Diese domänenspezifischen Profile dienen dann der Beschreibung von fachlich motivierten Konfigurationen von vorhandenen Komponenten (die wiederum architekturzentriert entwickelt sein können) für z.B. Versicherungswesen, Automobilbau oder Embedded-Systeme. Dies stellt dann die Königsdisziplin dar, weil das Potenzial für die Generierung dort noch deutlich höher ist. Allerdings hat die architekturzentrierte Entwicklung in der Regel einen größeren Wiederverwendungsgrad.

Links & Literatur

[1] MDA Specifications, 2003, www.omg.org/mda/

[2] P. Roßbach, T. Stahl, W. Neuhaus: „Model Driven Architecture“, *Java Magazin* 9.2003

[3] M. Völter: „Codegenerierung – Ein Überblick“, *Java Magazin* 9.2003

[4] Generative Development Process, 2003, www.architectureware.de

[5] T. Stahl, M. Schepe, C. Robitzki: „Model Driven Architect“, *Java Magazin* (noch nicht veröffentlicht)

[6] M. Schepe, W. Neuhaus, P. Roßbach: „Praktische Entwicklung mit MDA/D“, *Java Magazin* 9.2003

[7] T. Weilkiens: „Vom Modell zum Code – ein kurzer Weg mit MDA“, *Java Magazin* 9.2003

Vom Geschäftsprozess zum Code – ein kurzer Weg mit MDA

von Tim Weilkiens, Bernd Oestereich, Thomas Stahl

Vom Modell zum Code

Geschäftsprozessmodellierung, Systemanalyse, Systemdesign, Implementierung. Vier Domänen, die eng aufeinander aufbauen, zwischen denen aber oft tiefe Gräben liegen. Der Entwickler muss die eigentlichen Softwareanforderungen häufig aufwändig in diversen Dokumenten mit unterschiedlichen Notationen und Detaillierungen suchen. Das führt in den meisten Fällen zu Inkonsistenzen und unerwünschtem Systemverhalten. Mit der UML als gemeinsame Beschreibungssprache, der objektorientierten Geschäftsprozessmodell (OOGPM)-Methodik und der MDA/D-Technologie können diese Gräben überwunden werden. Dieser Artikel zeigt einen durchgängigen Weg vom Geschäftsprozess zum MDA-Design und erläutert ihn anhand eines praktischen Beispiels.

Geschäftsprozessmodellierung

Die Geschäftsprozessmodellierung (GPM) erfreut sich seit nunmehr einem Jahrzehnt großer Aufmerksamkeit in Praxis und Theorie. Im Rahmen der Anforderungsanalyse für ein Systementwicklungsprojekt wird meist eine begrenzte Modellierung betrieben, um den Kontext des zu entwickelnden Systems zu analysieren und darzustellen. Wird in dieser Domäne gleich konsequent die Unified Modeling Language (UML) eingesetzt, erzielt man den Vorteil gegenüber spezifischen Notationen der GPM (z.B. in ARIS – Architektur integrierter Informationssysteme), dass eine einheitliche Beschreibungssprache vom Geschäftsprozessmodell bis zum Systemdesignmodell verwendet wird. Ein weiterer Vorteil ist die formale Fundierung der UML. Das ist eine wichtige Grundvoraussetzung, um MDA überhaupt möglich werden zu lassen. Somit sind bereits in Geschäftsprozessmodellen potenzielle Ansatzpunkte für MDA gegeben. Die Vorgehensweise der objektorientierten GPM mit der UML wird in [1] detailliert beschrieben.

Geschäftsanwendungsfälle und Geschäftsprozesse

Dieser Artikel zeigt den Weg vom Geschäftsprozess zum Code anhand eines Fallbeispiels (siehe Kasten „Fallbeispiel“). Zunächst ein kurzer Blick auf die Modellierung der Geschäftsprozesse.

Ein Geschäftsanwendungsfall beschreibt einen geschäftlichen Ablauf, d.h. er beinhaltet die Schritte, die manuell und nicht systemgestützt durchgeführt werden. Fachlich zusammengehörige Geschäftsanwendungsfälle werden zusammengefasst und bilden gemeinsam einen Geschäftsprozess. Die sachlogische Reihenfolge, in der die Geschäftsanwendungsfälle eines Geschäftsprozesses ausgeführt werden dürfen, wird in einem Ablaufmodell in Form eines Aktivitätsdiagramms ausgedrückt. Abbildung 1 zeigt das Aktivitätsdiagramm des gefundenen Geschäftsprozesses *Kfz-Vermietung* für ein Car-Sharing- Unternehmen aus unserem Fallbeispiel. Die einzelnen Schritte entsprechen den identifizierten Geschäftsanwendungsfällen. Der Ablauf jedes einzelnen Geschäftsanwendungsfalls wird

Fallbeispiel

Das Fallbeispiel ist ein fiktives Car-Sharing-Unternehmen. Die Kunden sind nur registrierte Teilnehmer. In der Sprache des Unternehmens sind es die Mitglieder. Als Mitglied zahlt man monatlich einen Mitgliedsbeitrag und ist berechtigt, die Kfz des Fuhrparks zu nutzen. Hierzu muss vorab über das Call-Center eine Reservierung durchgeführt werden. Die Kfz stehen an verschiedenen Stationen im Stadtgebiet verteilt. Jedes Mitglied ist im Besitz einer Chipkarte, die den Zugang zu von ihnen reservierten Kfz über ein eingebautes Bordcomputersystem ermöglicht. Nach der Nutzung muss das Kfz wieder an derselben Station abgegeben werden, an der es auch ausgeliehen wurde. Mit der Chipkarte wird es dann wieder verschlossen. Die Abrechnung der einzelnen Fahrten erfolgt monatlich und basiert auf Zeit- und Kilometertarifen.

Das Unternehmen arbeitet mit verschiedenen veralteten Systemen, die untereinander nicht kompatibel sind, viele manuelle Schritte erfordern und häufig zu Fehlern und Inkonsistenzen führen. Um die Mitgliederzufriedenheit und die Effizienz in diesem Geschäftsbereich zu verbessern, strebt das Unternehmen die Entwicklung eines einheitlichen Systems an.

wiederum mit einem eigenen Aktivitätsdiagramm beschrieben.

Weitere Aktivitäten der OOGPM-Methodik führen beispielsweise zu einem Glossar, einem Geschäftsklassenmodell sowie zu nicht-funktionalen Anforderungen wie Geschäftsregeln.

Systemanwendungsfälle definieren?

Anhand der GPM-Ergebnisse werden nun die Systemanwendungsfälle abgeleitet. Ein Systemanwendungsfall beschreibt das gewünschte externe Systemverhalten aus der Sicht der Anwender und somit Anforderungen, die das System zu erfüllen hat. Sein Ablauf beinhaltet im Gegensatz zum Geschäftsanwendungsfall nur Schritte, die vom System ausgeführt werden.

Das Management des Car-Sharing-Unternehmens beschließt, die Geschäftsanwendungsfälle im Geschäftsprozess *Kfz-Vermietung*, die die Reservierung betreffen, durch ein EDV-System zu unterstützen, um diesen Geschäftsbereich zu optimieren. Abbildung 2 zeigt einen Ausschnitt der somit identifizierten Systemanwendungsfälle. Auch ihre Abläufe werden mit Aktivitätsdiagrammen beschrieben. Größtenteils können sie von den Aktivitätsdiagrammen der entsprechenden Geschäftsanwendungsfälle abgeleitet werden. Unterschiede entstehen beispielsweise dadurch, dass manuelle Schritte wie *Kfz fahren* nicht übernommen werden.

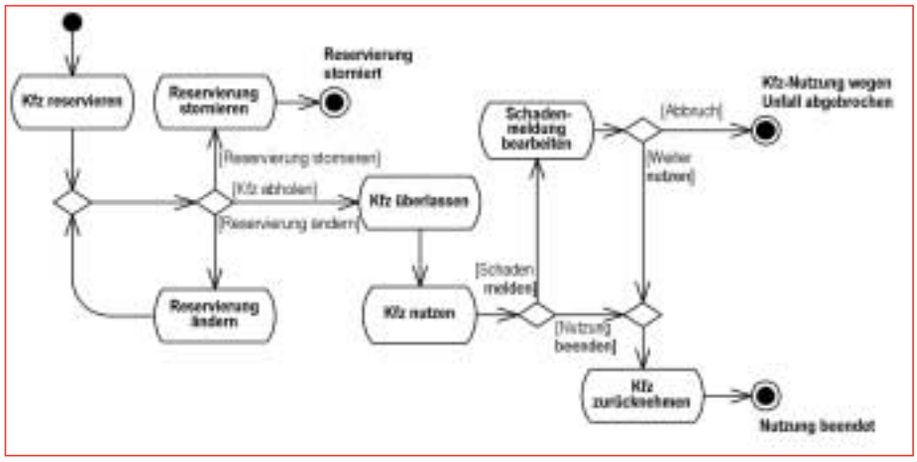

Abb. 1: Ablaufmodell Geschäftsprozess Kfz-Vermietung

Weitere Ergebnisse der Systemanforderungsanalyse sind ein Fachklassenmodell, abgeleitet aus dem Geschäftsklassenmodell, und nicht-funktionale Anforderungen, die ebenfalls aus den Ergebnissen der GPM abgeleitet werden können.

Früher MDA-Ansatz

Im Design ist es das Ziel, möglichst früh ein Detaillierungsniveau zu finden, auf dem der in dieser Artikelreihe vorgestellte Model Driven Architecture/Development (MDA/D)-Ansatz effektiv aufsetzen kann. Das heißt, das MDA-Designmodell (PIM) sollte sich eng an dem Systemanalysemodell orientieren und der Zusammenhang explizit über Modellbeziehungen oder implizit über Namenskonventionen vorliegen. Die enge Bindung der beiden Modelle gewährleistet, dass die im Systemanalysemodell spezifizierten Anforderungen auch in der gewünschten Ausprägung umgesetzt werden. Ebenso können Änderungen und ihre Auswirkungen in beide Richtungen gut verfolgt werden.

Hinsichtlich MDA ist vor allem die Generierung dynamischer Aspekte spannend. Ein guter Ausgangspunkt hierfür sind die Aktivitätsdiagramme der Systemanwendungsfälle. Um die Abbildung auf geeignete Elemente im Sourcecode zu spezifizieren, müssen noch entsprechende Stereotypen der Designsprache (dazu mehr im nächsten *Java Magazin*) ergänzt werden. Bis auf diesen Zusatz würde man bereits aus einem reinen Systemanalysemodell heraus eine Transformation durchführen können. Das führt aber leider zu Ergebnissen, die noch nicht befriedigen können, da sich nur wenige Informationen sinnvoll übertragen lassen. Die Ursache liegt vor allem in einem entscheidenden Unterschied zwischen unseren vorhandenen Systemanalysemodellen und dem gewünschten Sourcecode. Die Aktivitätsdiagramme der Systemanwendungsfälle beschreiben das Verhalten des Systems mit dem Fokus auf der sachlogischen Reihenfolge (Ablaufmodelle). Der zugehörige Sourcecode muss aber das reaktive Verhalten des Systems auf eintreffende Ereignisse umsetzen. Das ist eine andere Sichtweise. Für die Transformation zur Reaktion fehlen in den Aktivitätsdiagrammen die Ereignisse. Ohne Verlust der Einfachheit kann diese Information nicht

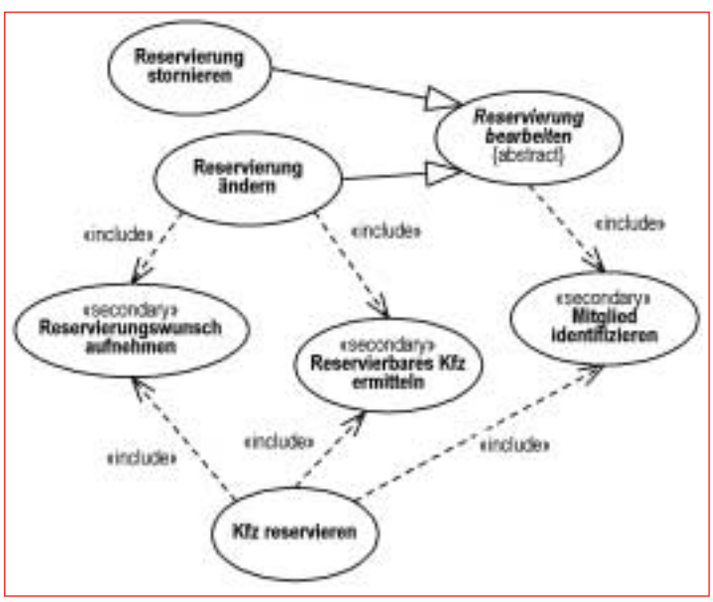

Abb. 2: Systemanwendungsfall-Struktur

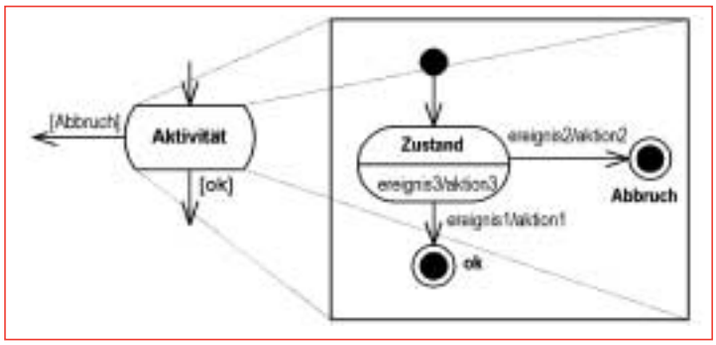

Abb. 3: Zusammenhang zwischen Aktivitäts- und Zustandsdiagramm

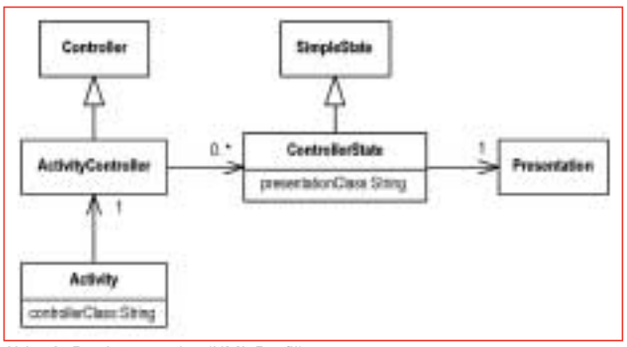

Abb. 4: Designsprache (UML-Profil)

ergänzt werden. Zudem würde man den Abstand zum ursprünglichen Systemanalysemodell wesentlich vergrößern, was nicht erwünscht ist. Für die Modellierung des reaktiven Verhaltens eignen sich die Zustandsdiagramme der UML.

Zustandsmodelle für die MDA-Transformation

Beschreibt man also das Verhalten eines Systemanwendungsfalls zusätzlich mit einem Zustandsdiagramm, gewinnt man ausreichend Informationen, um es in Sourcecode transformieren zu können. In einem Zustandsdiagramm können Ereignisse und die unmittelbare Reaktion darauf an den Transitionen (Zustandsübergänge) bzw. in den Zuständen selbst spe- zifiziert werden. Allerdings hat man mit einem Aktivitätsdiagramm und einem Zustandsdiagramm je Anwendungsfall ein hohes Maß an Redundanz, was bekanntlich zu vielen Problemen und Fehlern führt. Beide Modelle enthalten beispielsweise die sachlogische Reihenfolge der einzelnen Systemanwendungsfall-Schritte.

Das Aktivitätsdiagramm hat sich in der bekannten Form hervorragend bewährt, um einen Anwendungsfallablauf zu beschreiben. Hier sollten somit keine Änderungen vorgenommen werden. Also modelliert man das Zustandsdiagramm derart, dass keine Redundanz zum Aktivi-

Abb. 5: Ablaufmodell *Kfz reservieren*

tätsdiagramm entsteht. Das erzielt man durch eine geeignete Definition des Kontextes. Statt des gesamten Systemanwendungsfalls wählt man einen Schritt im Aktivitätsdiagramm als Kontext für das Zustandsdiagramm. D.h. jeder einzelne Schritt wird durch ein eigenes Zustandsdiagramm beschrieben. Dort werden dann die Ereignisse und die entsprechenden Reaktionen spezifiziert, sodass sie automatisch in Sourcecode mit dem gewünschten Verhalten transformiert werden können.

Abbildung 3 zeigt den Zusammenhang zwischen einer Aktivität und dem zugehörigen Zustandsdiagramm. In der Abbildung hat das Zustandsdiagramm zwei Endknoten: *ok* und *Abbruch*. Für jeden Endknoten im Zustandsdiagramm sind im Aktivitätsdiagramm passende Bedingungen an den ausgehenden Kanten.

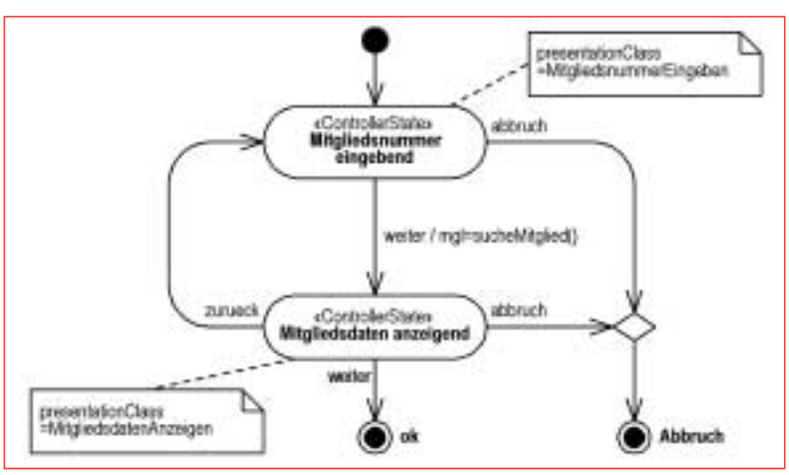

Abb. 6: Zustandsmodell *Mitglied identifizieren*

MDA Transformation

Der Artikel „Model Driven Architect“ im nächsten *Java Magazin* beschreibt die Entwicklung einer Referenzimplementierung, um daraus eine Designsprache (UML-Profil) abzuleiten. Abbildung 4 zeigt einen kleinen Ausschnitt aus diesem Profil. Die Klassen sind die Stereotypen, die Attribute die zugehörigen Eigenschaftswerte. Ein Stereotyp erweitert das vorhandene UML-Vokabular um ein weiteres Element, das auf einem Standardelement der UML basiert und dieses um spezielle Eigenschaften ergänzt („Stereo“-typ). Mit der UML und der Designsprache haben wir nun das Vokabular, mit dem das PIM modelliert werden kann.

Die Modellierung des PIM wird anhand des Systemanwendungsfalls *Kfz reservieren* gezeigt. In Abbildung 5 sieht man das Aktivitätsdiagramm angereichert mit Stereotypen aus der Designsprache. Das Aktivitätsdiagramm des Geschäftsanwendungsfalls *Kfz reservieren* sieht identisch aus, da der Geschäftsanwendungsfall vollständig vom System unterstützt werden soll. Jede Aktivität, die mit dem Stereotyp *Activity* gekennzeichnet ist, wird bei der Transformation berücksichtigt. Der Eigenschaftswert *controllerClass* stellt den Bezug des Aktivitätsdiagramms (dynamisches Modell) zum Klassendiagramm (statisches Modell) her. Der Wert gibt den Klassennamen der Kontrollklasse (*ActivityController*) aus der Präsentationsschicht an.

Wie oben bereits festgestellt, werden Ereignisse und Reaktionen in Zustandsdiagrammen spezifiziert. Das Aktivitätsdiagramm gibt nur den Gesamtablauf vor. Abbildung 6 zeigt das Zustandsdiagramm der ersten Aktivität *Mitglied identifizieren*. Die Zustände sind mit dem Stereotypen *ControllerState* versehen. Die zugehörige Präsentationsklasse wird im Eigenschaftswert *presentationClass* angegeben. Es gibt zwei Endzustände *ok* und *Abbruch*. Entsprechend findet man im Aktivitätsdiagramm (Abb. 5) aus der Aktivität zwei ausgehende Transitionen mit den Bedingun- gen *[ok]* und *[Abbruch]*. Die Transition von *Mitgliedsnummer eingebend* zu *Mitgliedsdaten anzeigend* zeigt deutlich das Ereignis/Reaktion-Verhalten. Wenn der Anwender

nach der Eingabe der Mitgliedsnummer auf den WEITER-Knopf in der Applikation drückt, löst dies das entsprechende *weiter*-Ereignis in dem Zustandsdiagramm aus. Das führt dazu, dass die zugehörige Transition feuert. Die Angabe *mgl=sucheMitglied()* an der Transition ist eine Aktion, die während des Zustandübergangs ausgeführt wird. Es ist die unmittelbare Reaktion auf das Ereignis *weiter*. Die Syntax der Aktion ist abhängig von der gewählten *Action Language* für das Modell. Im einfachsten Fall wählt man hier eine Programmiersprache, z.B. Java. Der hier im Modell angegebene Sourcecode wird dann direkt an die entsprechende Stelle im Zielcode transformiert.

Die Eigenschaftswerte der Stereotypen im Aktivitäts- und Zustandsdiagramm stellen den Bezug zu den Klassen her, die das zugehörige Verhalten umsetzen. Abbildung 7 zeigt einen Ausschnitt aus dem Klassendiagramm.

Aus den Aktivitäts- und Zustandsdiagrammen (Dynamik) und den Klassendiagrammen (Statik) findet die Transformation in den Sourcecode statt. Die Aufrufreihenfolge der einzelnen Methoden wird je nach Zielplattform z.B. in Interpreter code oder Konfigurationsdateien wie *strutsconfig* transformiert.

Entitäten

Bisher haben wir nur die Abläufe und ihre dynamische und statische Modellierung betrachtet. Werfen wir nun einen Blick auf die Modellierung der Entitäten (Informationsklassen). Die Klassen und ihre Beziehungen untereinander können aus dem Geschäfts- bzw. Fachklassenmodell abgeleitet werden. Eine Entity erhält aus der Designsprache den Stereotyp *EntityObject*. Basierend auf dem Model View Control (MVC)-Entwurfsmuster wird die Beziehung der Entitäten (Model) mit dem ActivityController (View) über eine Prozessklasse (Control) hergestellt. Die Prozessklasse wird mit dem Stereotyp *ProcessObject* versehen und enthält Operationen, die vom *ActivityController* aufgerufen werden. Damit die View unabhängig von den spezifischen Entitäten ist, werden die Daten zwischen View und Control über Value-Objekte transportiert. Sie

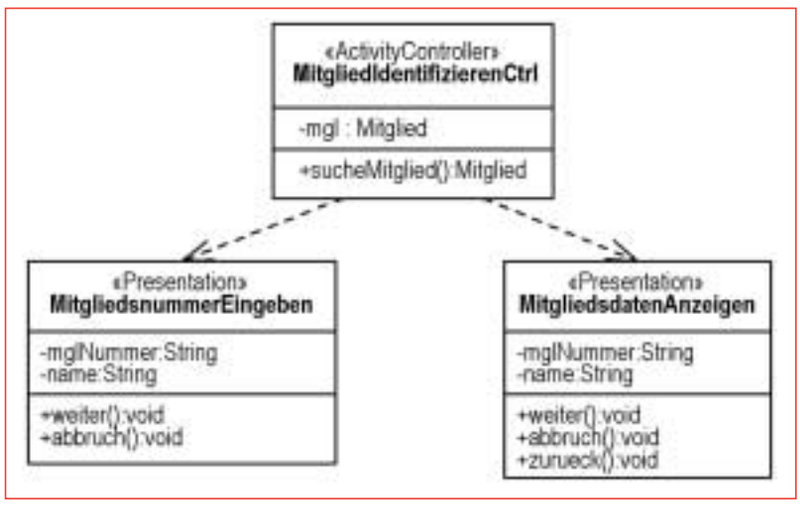

Abb. 7: Klassenmodell *ActivityController*

werden mit dem Stereotyp *ValueObject* gekennzeichnet. Insgesamt ergibt sich dann das in Abbildung 8 dargestellte Klassenmodell.

Fazit und Ausblick

Was haben wir nun erreicht? Wir haben ein PIM modelliert, das sehr eng mit dem Systemanalysemodell verbunden ist und wegen der Durchgängigkeit der Methodik auch eine nahe Beziehung zu dem Geschäftsprozessmodell hat. Aus dem PIM kann automatisch der Sourcecode generiert werden. Somit ist insgesamt sichergestellt, dass fachlich gestellte Anforderungen – sei es in derGeschäftsprozesswelt oder in der Systemanforderungsanalyse – in der gewünschten Ausprägung im späteren System enthalten sind. Neue oder veränderte Anforderungen an das System können leicht in die Entwicklung einfließen, da zwischen den Modellen keine Gräben, sondern klar definierte Brücken sind.

Bisher können noch nicht alle gestellten Systemanforderungen in den MDA-Ansatz einbezogen werden. Die kürzlich verabschiedete UML Version 2.0 erweitert die Möglichkeiten auf diesem Gebiet, da die Neuauflage der Modellierungssprache sehr viel formaler ist und um interessante Elemente erweitert wurde. Das Potenzial, das hinter dem GPM/MDA-Ansatz steckt, ist also noch lange nicht ausgeschöpft.

Links & Literatur

[1] B. Oestereich, C. Weiss, C. Schröder, T. Weilkiens, A. Lenhard: „Objektorientierte Geschäftsprozessmodellierung mit der UML“, dpunkt.verlag, 2003

[2] OMG Unified Modeling Language, 2003, www.omg.org/uml

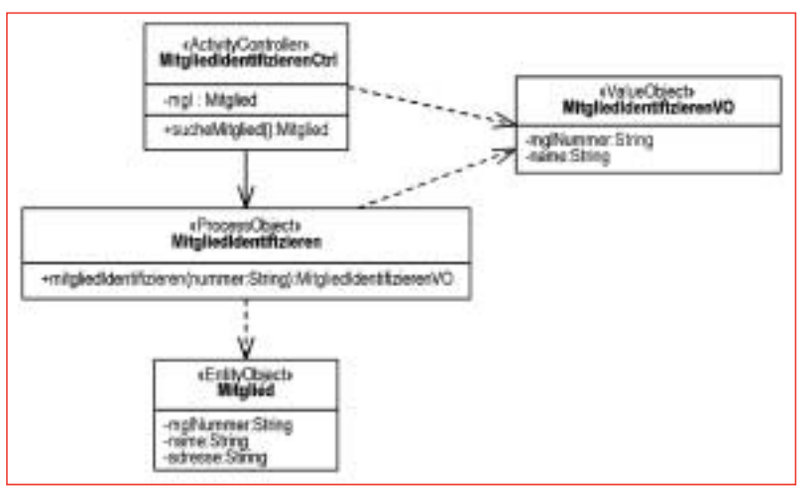

Abb. 8: Klassenmodell für den Anwendungsfallschritt *Mitglied identifizieren*

Anwendungsentwicklung in MDA/D-basierten Projekten unter Verwendung einer existierenden generativen Architektur

von Martin Schepe, Wolfgang Neuhaus, Peter Roßbach

Praktische Entwicklung mit MDA/D

Die Softwareentwicklung mit Hilfe der Model Driven Architecture (MDA) erfordert unterschiedliche Rollen. Eine Schlüsselrolle – wie könnte es auch anders sein – ist die des Entwicklers. Wir beschreiben die wesentlichen Aspekte der Softwareerstellung mit einem MDA/D-basierten Entwicklungsprozess aus Sicht des Entwicklers bei vorhandener generativer Architektur. Anhand eines Anwendungsbeispiels werden wir die wesentlichen wiederkehrenden Schritte exemplarisch erläutern und abschließend die Beziehung zwischen Entwicklung und Architektur beschreiben.

Zuvor jedoch eine kurze Beleuchtung der Ausgangssituation. Wie sieht heute der Alltag in der Rolle als Softwareentwickler aus? Darauf kann selbstverständlich nur eine recht allgemeine Antwort gegeben werden. In der Regel erhalten wir eine – mehr oder weniger gute – Spezifikation für die zu realisierenden Programmmodule. Die Umsetzung der dort aufgestellten Anforderungen erfolgt gemäß einer vorgegebenen Architektur und weiterer Sourcecode-Konventionen mit Hilfe einer integrierten Entwicklungsumgebung oder eines Programm- editors mit Ant-basierten Build-Scripten. Es entsteht zum einen technischer Code zur Unterstützung der verwendeten Architektur, wie z.B. Local- und Remote-Interfaces bei J2EE, und zum anderen fachlicher Code zur Umsetzung der Business Logik, z.B. Reservierung eines Kfz. Häufig entfernt sich dabei die Realisierung vom Anwendungsdesign, wenn vorhanden, da das Spezifikationsdokument bei Änderungen auf Implementierungsebene nicht ständig abgegli- chen wird. Dabei bleibt also jede Menge Spielraum, um Spezifikationen, Architekturvorgaben und Coding-Styleguides zu interpretieren und nachhaltig unabgesprochen zu ändern. Während der Anwendungsentwicklung entstehen so zwangsweise gewisse Unschärfen bei der Interpretation der oben genannten Vorgaben, was oft zu sehr individualisiertem Code führt. Dies ist zur Identifikation mit dem eigenen Code vertretbar, führt aber aufgrund der unterschiedlichen Skills und Erfahrungen der Teammitglieder zu Einzellösungen unterschiedlichster Schematik, Qualität und unterschiedlicher Wartbarkeit. Zusätzlich erschwert gerade die fehlende einheitliche Schematik die Einarbeitung in die Sourcen der anderen Teammitglieder und die Wartbarkeit der Gesamtanwendung leidet zunehmend. Dieses Verfahren führt dann schnell zu dem Unspruch: Never touch a runnig system!

Entwicklung mit einem MDA/D-basiertem Vorgehen

Bei der Entwicklung mit einer praktischen Interpretation der MDA-Standards [1] der OMG wie dem Generative Development Process [2] ist natürlich eine Spezifikation die Grundlage für die Erstellung der gewünschten Anwendung. Zusätzlich wird eine generative Architektur, die im

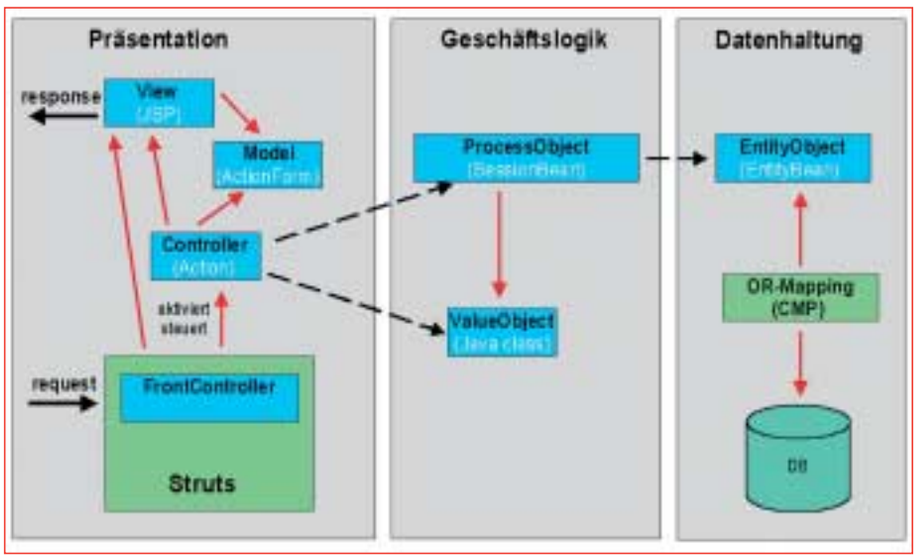

Abb. 1: Car-Sharing Basisarchitektur

Architekturstrang parallel zur Analyse erarbeitet worden ist, zur Verfügung gestellt [3]. Diese generative Architektur liefert eine auf die Projektbedürfnisse abgestimmte Designsprache, um das Anwendungsdesign eindeutig und formal erstellen zu können. Darüber hinaus stellt sie Transformationsvorschriften in Form von Templates zu Verfügung. Mit Hilfe der Templates generieren wir aus dem Anwendungsdesign den Sourcecode für alle schematischen Anteile, dem technischen Code. Wiederverwendbare Komponenten und Frameworks zur Ergänzung des Laufzeitsystems werden oft als zusätzliche Dienste der generativen Architektur zur Verfügung gestellt.

Begonnen wird bei der Anwendungsentwicklung mit der Erstellung des Anwendungsdesigns unter Zuhilfenahme eines UML-Tools. Die aus dem UML-Tool exportierte XMI-Repräsentation des Anwendungsdesigns wird durch den MDA/D-Generator in einen Implementierungsrahmen transformiert. Die eigentliche Fachlogik, als fachlicher Code, wird manuell mittels Programmeditor oder integrierter Entwicklungsumgebung in die im Sourcecode generierten geschützten Bereiche programmiert. Die Arbeit erfolgt konsequent im Forward Engineering [3]. Bei notwendigen Änderungen der generativen Architektur wird vom Kontext der Anwendungsentwicklung in den Kontext der Architektur gewechselt, um jetzt in der Rolle des Architekten benötigte Anpassungen an Designsprache, Transformationsvorschriften und Laufzeitsystem vorzunehmen.

Die unterschiedlichen Rollen im GDP sind natürlich nicht fest an Personen gebunden. Je nach Projektsituation, Skills und Neigungen der Teammitglieder können sie unterschiedliche oder feste Rollen einnehmen. Bei bestimmten Gegebenheiten, z.B. Entwicklung einer neuen Anwendungsarchitektur, ist es sinnvoll, alle Rollen von jedem Teammitglied ausüben zu lassen. Die wesentlichen Neuerungen für Programmerstellung beim Einsatz des GDP bestehen damit in der Modellierung des Anwendungsdesigns mit einer formalen Designsprache, dem konsequenten Forward Engineering und den exakten Vorgaben für geschützte Code-Bereiche, in denen Fachlogik ergänzt wird. Manch einer verbindet mit diesen Neuerungen eine Einschränkung persönlicher Freiheiten oder gar die „Gängelung“ durch den Architekten. Solche Vorbehalte entstehen häufig auf Grund fehlender oder falscher Informationen. Das Vorgehen selbst erwartet keine Besetzung der Rollen mit festen Personen, sondern beschreibt, wie bei der J2EE-Entwicklung lediglich die Aufgaben der einzelnen Rollen wie z.B. Entwickler und Architekt [4]. Die Rollenbesetzung selbst liegt in der Verantwortung des Teams. Das Team bestimmt, ob die Rolle vollständig agil, starr oder in einer Mischform eingenommen werden soll. Somit kann eine „Gängelung“ nicht durch den Wechsel des Vorgehens, sondern durch eine Umorganisation der Rollenverteilung oder -nutzung im Projektteam beseitigt werden.

Das Anwendungsbeispiel

Die Basis für alle weiteren Betrachtungen bildet ein fiktives Produkt Car-Sharing 1.0, das erstmals auf der JAX2003 in einer Session präsentiert wurde. Die Anwendung wurde erstellt, um einen ganzheitlichen Ansatz zur Entwicklung von Software, angefangen bei der Analyse und dem Design über die MDA/D-basierte Codegenerierung bis hin zur Implementierung der Fachlogik anhand eines einfachen Beispiels demonstrieren zu können. Der fachliche Hintergrund der Anwendung basiert auf einem Fallbeispiel zur Erstellung eines Informationssystems für ein Car-Sharing Unternehmen (siehe Artikel „Vom Modell zum Code“, S. 30). Car-Sharing setzt in der Version 1.0 den Systemanwendungsfall *KFZ reservieren* um und erlaubt die Reservierung und Verwaltung von KFZ. Die Mitglieder der Car-Sharing-Gemeinschaft sind zur Autorisierung und späteren Abrechnung im System erfasst. Der Hauptzweck des Systems ist die elektronische Durchführung von Fahrzeugreservierungen durch Call-Center-Agenten.

Die Architektur der Car-Sharing-Anwendung ist eine klassische 3-Tier-Architektur bestehend aus einer Präsentations-, Prozess- und Datenhaltungsschicht. Sie basiert auf der J2EE-Rahmenarchitektur (Abb. 1).

Die Präsentationsschicht verwendet das MVC-Pattern gemäß der Servlet-Model 2 Architektur à la Struts [5]. Alle vom Browser kommenden HTTP-Anfragen werden zentral von einem *FrontController* entgegen genommen, ausgewertet und an die entsprechend anzuzeigende View weitergeleitet. Die Evaluierung und Abarbeitung von ausgelösten GUI-Aktionen und die Auswertung von bedingten Verzweigungen (Guards) in der Navigationsfolge delegiert der *FrontController* an entsprechende *SubController*. Der *SubController* stellt der View die zur Anzeige benötigten Daten in einem *ViewModel*

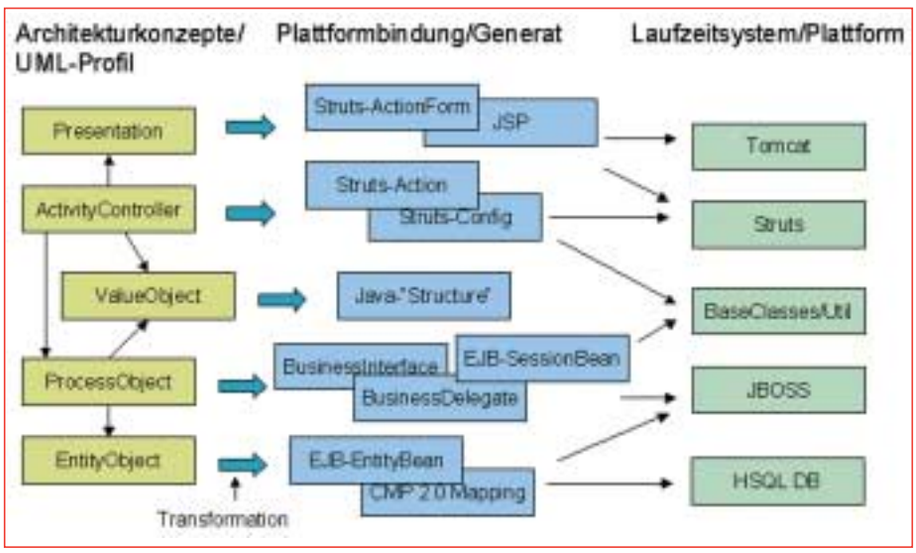

Abb. 2: Generative Architektur und Laufzeitumgebung

zur Verfügung. Zur Ablaufsteuerung stützt sich die Schicht auf das beliebte Struts-Framework 1.1. Der Datenaustausch mit der Prozessschicht erfolgt über *ValueObjects*. Die Prozessschicht bietet den Controllern der Präsentationsschicht zustandslose transaktionale Dienste in Form von Methoden auf *ProcessObjects* an. Die Controller können über diese *ProcessObjekte* zur Anzeige relevante Daten lesen und neu erfasste Daten persistieren. Gleichzeitig werden die in der Datenhaltungsschicht liegenden Entitäten gemäß des Process-Entitiy-Patterns vor direkten Zugriffen durch Objekte der Präsentationsschicht geschützt.

Die Datenhaltungsschicht verfügt über ein persistentes Business Object Model (BOM), welches durch Entity Beans realisiert ist. Die Persistierung erfolgt über den Container Managed Persistence (CMP)-Mechanismus in eine SQL-Datenbank. Die Plattform und Laufzeitumgebung (Zielplattform) der Car-Sharing-Anwendung stützt sich ausschließlich auf Open Source-Software. Als Plattform dient ein Tomcat Webserver, der EJB 2.0 konforme Applikations-Server JBoss und die Datenbank HyperSonic SQL. Das zentrale Element der Laufzeitumgebung ist das Struts-Framework zur Steuerung der Abläufe und Prozesse in der Anwendung. Zusätzlich ergänzen exemplarisch einige Basis- und Hilfsklassen, die den Anwendungsrahmen der Car-Sharing-Anwendung bilden, die Laufzeitumgebung.

MDA-Werkzeuge

Um MDA praktisch zu nutzen, benötigen wir ein UML-Modellierungswerkzeug (Rational Rose, Together, MagicDraw, MID Innovator, ObjectIF, etc.) und einen MDA/D-Generator. Das UML-Modellierungswerkzeug muss in der Lage sein, mit UML-Profilen zur UML-Spracherweiterung zu arbeiten. Zurzeit findet sich kein UML-Tool, das Modellierungs-constraints, d.h. die Prüfung von Zusicherungen, die auf der Metaebene festgelegt worden sind, unterstützen kann, sodass diese Prüfung der Zusicherungen durch den MDA/D-Generator unterstützt werden sollte. Das Generator-Werkzeug muss die vom ausgewählten UML-Werkzeug erstellten Modelle lesen und als Eingabe für die Generierung nutzen können. Der MDA/D-Generator sollte zusätzlich eine Unterstützung zur Prüfung der Modellierungs-constraints bereitstellen. Die meisten UML-Tools sind heute in der Lage, Modelle im XMI-Format abzuspeichern. Allerdings ist XMI noch lange nicht gleich XMI. Hier bieten viele Generatorhersteller spezielle Adapter für die unterschiedlichen Modellierungswerkzeuge an. Der MDA/D-Generator sollte eine flexible Transformation und ein anpassbares Metamodell besitzen. Die OMG bietet eine Aufstellung von MDA/D-Tools [8]. Allerdings sollte man hier genau hinsehen, denn nicht alle dort gelisteten Tools erfüllen die oben genannten Anforderungen.

Listing 1

```
...
<html:form action="<%= (String)
request.getAttribute("FormAction") %>" method="Post">
 <table border="0" cellspacing="0" cellpadding="0" >
 <tr>
  <td><bean:message key="de.amg.stattauto.benutzer.
              presentation.BenutzerAnmelden.kennung"/>
                                           </td>
  <td>
   <html:text property="kennung"/>
  </td>
 </tr>
 <tr>
  <td><bean:message key="de.amg.stattauto.benutzer.
              presentation.BenutzerAnmelden.passwort"/>
                                           </td>
  <td>
   <html:password property="passwort"/>
  </td>
 </tr>
 <tr>
  <td>
   <input type="hidden" name="registration.jsp.
                                  Event" value="Weiter">
   <input type="submit" name="Event" value="Weiter"/>
  </td>
  <td>
   <input type="hidden" name="registration.jsp.
                                 Event" value="Beenden">
   <input type="submit" name="Event" value=
                                          "Beenden"/>
  </td>
 </tr>
</table>

</html:form>
...
```

Listing 2

```
package de.amg.stattauto.benutzer.presentation;

import org.apache.struts.action.ActionForm;
public class BenutzerAnmeldenForm extends ActionForm
{
 private String kennung;
 private String passwort;
 public String getKennung()
 {
  return kennung;
 }
 public void setKennung(String aKennung)
 {
  kennung = aKennung;
 }
 public String getPasswort()
 {
  return passwort;
 }
 public void setPasswort(String aPasswort)
 {
  passwort = aPasswort;
 }
}
```

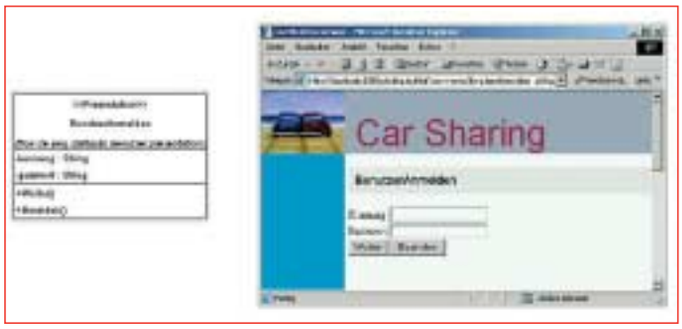

Abb. 3: Transformation vom Modell *BenutzerAnmelden* auf den konkreten Dialog *BenutzerAnmeldenView*

Als UML-Modellierungswerkzeug kommt Poseidon UML Community Edition von Gentleware zum Einsatz [6]. Dessen XMI-Output wird vom b+m GeneratorFramework über die Generatorschablonen in Sourcecode transformiert [7]. Dieser Sourcecode wird dann in Eclipse weiterverarbeitet.

Die Erstellung des für Car-Sharing verwendeten Designs in Form eines PIMs erfolgt mit Hilfe einer Designsprache (UML-Profils), welche auf Architekturkonzepte abzielt. Vereinfacht finden wir im UML-Profil die in der konzeptionellen Architekturübersicht skizzierten Aspekte als Konstrukte der Designsprache wieder (z.B. EntityObject, ValueObject,...). Die Transformation (Plattformbindung) auf die Zielplattform erfolgt über einen Satz von Generatorschablonen, welche aus den Modellinformationen den entsprechenden Sourcecode für die oben skizzierte Zielumgebung generieren. Die Designsprache und die Plattformbindung in Form der Templates bilden die in Abbildung 2 dargestellte generative Architektur für Car-Sharing 1.0 (siehe Artikel „Model Driven Architect“ in der nächsten Ausgae des *Java Magazins*).

Die folgenden Beispiele skizzieren die Tätigkeiten des Entwicklers auf den verschiedenen Ebenen der Anwendungserstellung beim Design/Generate/Build-Zyklus.

Beispiel 1: Einfache Modelländerungen

Das erste Beispiel skizziert eine einfache Änderung im statischen Klassenmodell der Car-Sharing-Anwendung und einen entsprechenden Durchlauf der Tätigkeiten des Zyklus von Design, Generate und Build. Da für die Car-Sharing-Anwendung JSPs vollständig aus den Informationen der als <<*Presentation*>> ausgezeichneten Klasse generiert werden, bietet sich eine Änderung auf Ebene einer Presentation an.

Der obere Bereich der Abbildung 3 zeigt die Präsentations-Klasse *BenutzerAnmelden* und den aus ihr entsprechend generierten Dialog als HTML-Ausgabe einer JSP. Die Methoden der *Presentation* finden eine Entsprechung in den SUBMIT-Buttons der JSP-Präsentation im Browser. Die Umbenennung der Methode *Ende* in *Beenden* der Klasse *BenutzerAnmelden* resultiert in der JSP Präsentation im Browser in einer entsprechenden Beschriftung des SUBMIT-Buttons in *Beenden*, wie im Dialog zu sehen. Neben der JSP wird die Struts-ActionForm, die ein ViewModel darstellt, aus der Präsentations-Klasse vollständig

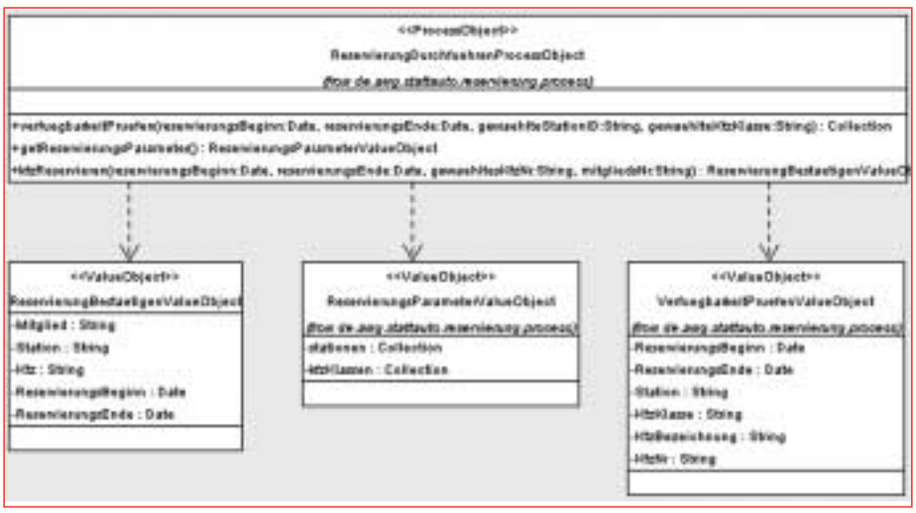

Abb. 4: ReservierungDurchfuehrenProcessView

generiert. Beide Artefakte sind sind Ergebnisse der Transformation (Listings 1 und 2).

Bei einfachen Änderungen arbeiten wir bei der Entwicklung also ausschließlich auf Modellebene. Nach vollzogenen Änderungen erfolgt der Export des Modells ins XMI-Format. Der Generator interpretiert das XMI und generiert die entsprechenden Sourcen. Sowohl das Build als auch das Deployment finden anschließend über die Entwicklungsumge-bung statt. Hier wird erkennbar, dass das Modell die Stellung von Sourcecode ein- nimmt. Alle Informationen über die durch- geführte Änderung werden im Modell gehalten. So ist es naheliegend, dass das Modell neben den eigentlichen Sourcen in das Release-Management der Anwendung ein- bezogen wird.

Beispiel 2: Modelländerungen und geschützte Bereiche

Das zweite Beispiel verdeutlicht, wie individuelle Fachlogik in geschützten Bereichen ergänzt werden kann. Dazu sehen wir uns an, wie die für eine Reser- vierung notwendigen Parameter in der Prozessschicht bestimmt und der Präsentationsschicht zur Verfügung gestellt werden.

Abbildung 4 zeigt den in der Prozessschicht benötigten Modellausschnitt. Das *ReservierungDurchfuehrenProcessObject* erhält dazu die *getReservierungs-Parameter()*-Methode die ein *ReservierungsParameterValueObject* zurückliefert. Das *ReservierungsParameterValueObject* dient als Datencontainer, um die Daten aus der Prozessschicht an die Präsentationsschicht weitergeben zu können. Bei der Generierung werden aus diesem Modell alle notwendigen Klassen, Java-Interfaces und Deployment Deskriptoren generiert, die für die Umsetzung des *ReservierungDurchfuehrenProcessObject* in ein SessionBean notwendig sind. Zusätzlich wird ein *ReservierungDurchfuehrenBusinessDelegate* gemäß des Business Delegate-Patterns aus den J2EE Core Patterns generiert [9]. Das *ReservierungsParameterValueObject* wird zu 100% generiert. Für die Methode *getReservierungs Parameter()* wird jedoch „nur“ die Methodensignatur generiert. Die Implementierung muss manuell in der IDE ergänzt werden. Dies geschieht in geschützten Code-Bereichen (Listing 3).

Die Entscheidung über mögliche geschützte Code-Bereiche ist auf Architekturebene bei der Erstellung der Generator-Schablonen zu treffen. Alle Ergänzungen außerhalb dieser geschützten Bereiche gehen bei wiederholten Generierungen verloren. Die Festlegung der geschützten Bereiche ist von enormer Bedeutung, deren Tragfähigkeit sich bei der Anwendungsentwicklung beweisen muss [10].

Beispiel 3: Arbeiten mit dynamischen Modellen

Neben der in den vorangegangenen Beispielen aufgezeigten Möglichkeit der Generierung von Sourcecode auf Basis statischer

Listing 3: Methode des *ReservierungDurchfuehrenProcessObject-Bean*

```
public ReservierungsParameterValueObject
                                    getReservierungsParameter()
  throws RemoteException
 {
  // PROTECTED REGION
ID(127a0a0a1ae33bb8bf23a87384Operation_
                                    MethodBody) START
  ReservierungsParameterValueObject vo = null;

  try
  {
   StattAutoModuleComponent component =
                    new StattAutoModuleComponentImpl();
   StationHome home = component.getStationHome();
   Collection stationen = home.findByAll();

   Collection colStationen = new ArrayList();

   for (Iterator i = stationen.iterator(); i.hasNext();)
   {
    Station station = (Station) i.next();

    colStationen.add(station.getName());
   }

   Collection colKfzKlassen = new ArrayList();

   colKfzKlassen.add(KfzKlasse.KOMPAKT);
   colKfzKlassen.add(KfzKlasse.KLEINWAGEN);
   colKfzKlassen.add(KfzKlasse.MITTELKLASSE);
   colKfzKlassen.add(KfzKlasse.OBERKLASSE);

   vo = new ReservierungsParameterValueObject
                              (colStationen, colKfzKlassen);
  }
  catch (Exception e)
  {
   e.printStackTrace();
   throw new RemoteException("Fehler beim Suchen
                          der Reservierungsparameter", e);
  }

  return vo;

  // PROTECTED REGION END
 }
```

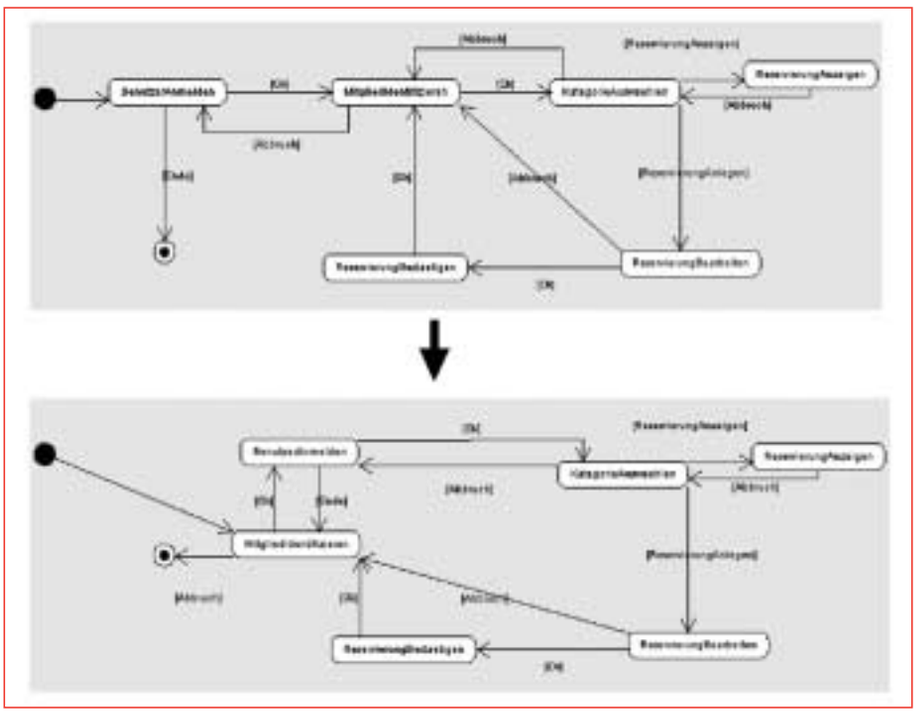

Abb. 5: Änderung der Navigationsfolge

Modelle können auch dynamische Modelle wie Aktivitätsdiagramme und Zustandsdiagramme zur Codegenerierung genutzt werden [11]. Wie so etwas aussehen kann, wollen wir mit diesem Beispiel aufzeigen. Abbildung 5 zeigt ein Aktivitätsdiagramm vor und nach einer Änderung der Navigationsfolge in der resultierenden Applikation. Gegenüber dem Ursprungszustand ist nun, bevor eine Benutzeranmeldung erfolgt, der Schritt zur Identifikation des anrufenden Mitglieds vorzunehmen (über die Sinnhaftigkeit dieser Änderung darf gestritten werden).

Da wir uns für Struts als Rahmenwerk zur Ablaufsteuerung in diesem Beispielprojekt entschieden haben, müssen auf Basis des Aktivitätsdiagramms die notwendigen Konfigurationen zur Ablaufsteuerung generiert werden. Wie dies konkret aussieht, zeigt der Ausschnitt der Struts-Konfiguration (Listing 4).

Damit kann die Ablaufsteuerung zu 100% aus dem Anwendungsdesign generiert werden. Manuelle Eingriffe in die Struts-Konfiguration sind nicht mehr notwendig. Nach unserer Erfahrung bietet dies insbesondere den Vorteil, dass die Aktivitätsdiagramme gleichzeitig sehr gut die Navigationsfolge dokumentieren und zur Diskussion auch mit fachseitigen Ansprechpartnern genutzt werden können [11].

Zusammenspiel Entwicklung und Architektur

Ein funktionierendes Zusammenspiel zwischen Entwicklung und Architektur ist ein wichtiger Schlüssel zum Erfolg in MDA/D-basierten Projekten. Denn mit der Bereitstellung der generativen Architektur ist die Arbeit an ihr keineswegs abgeschlossen. Durch die in den Beispielen aufgeführten Schritte der Anwendungsentwicklung entstehen im Rahmen des Projekts meistens auch Änderungsanforderungen an die generative Architektur. Auf der einen Seite werden zusätzliche geschützte Code-Bereiche für individuelle Ergänzungen benötigt auf der anderen Seite neue Architekturmuster identifiziert, die generativ unterstützt werden sollen. Nur wenn es im Team gelingt, dass aus der Entwicklung stammende Feedback aufzunehmen und in die generative Architektur einfließen zu lassen, entsteht am Ende eine möglichst tragfähige Lösung zur Erfüllung der gestellten Anforderungen. Somit entwickelt sich die generative Architektur ähnlich eines Frameworks evolutionär weiter und muss dem-

entsprechend versioniert sowie den Projekten, in denen sie verwendet wird, zur Verfügung gestellt werden.

Um bei der Entwicklung eigene Ideen zur Verbesserung der verwendeten generativen Architektur zu untersuchen, kann dies natürlich ohne weiteres lokal – gewissermaßen in einer „Sandbox" – ausprobiert werden. Der Code außerhalb der geschützten Bereiche bleibt bis zu einem erneuten Generierungslauf erhalten. Führt die Änderung zum gewünschte Ergebnis, kann sie dem gesamten Projekt durch Anpassung der generativen Architektur zur Verfügung gestellt werden. Dies hat den Vorteil, dass die generative Architektur innerhalb eines Projekts immer in einem klar definierten und konsistenten Zustand vorliegt.

Listing 4: Auszug aus der generierten struts-config.xml

```
<!-- ControllerState "BenutzerAnmelden" --i>
<action
 path="/BenutzerAnmelden_a64aa88aa2aa37ad0cdd0bf506f30162ba7ffb_Init"
 type="de.amg.stattauto.benutzer.presentation.BenutzerAnmeldenController"
 name="BenutzerAnmeldenForm"
 input="/BenutzerAnmelden.jsp"
 scope="request"
 parameter="BenutzerAnmelden_Init,a64aa88aa2aa37ad0cdd0bf506f30162ba7ffb">
 <forward name="Ok"
  path="/BenutzerAnmelden.jsp" contextRelative="true"
 />
</action>

<action
 path="/BenutzerAnmelden_a64aa88aa2aa37ad0cdd0bf506f30162ba7ffb_Exit"
 type="de.amg.stattauto.benutzer.presentation.BenutzerAnmeldenController"
 name="BenutzerAnmeldenForm"
 input="/BenutzerAnmelden.jsp"
 scope="request"
 parameter="BenutzerAnmelden_Exit,a64aa88aa2aa37ad0cdd0bf506f30162ba7ffb">
 <forward name="BenutzerAnmelden_To_Beenden"
  path="/MitgliedsnummerAngeben_a64aa88aa2aa37ad0cdd0bf506f30162ba7ffa_Init.do"
 />
 <forward name="Weiter"
  path="/KategorieAuswaehlen_a64aa88aa2aa37ad0cdd0bf506f30162ba7ff9_Init.do"
 />
 <forward name="Fehler"
  path="/BenutzerAnmelden_a64aa88aa2aa37ad0cdd0bf506f30162ba7ffb_Init.do"
 />
</action>

<!-- ControllerState „MitgliedsnummerAngeben" -->
<action
 path="/MitgliedsnummerAngeben_a64aa88aa2aa37ad0cdd0bf506f30162ba7ffa_Init"
 type="de.amg.stattauto.mitglied.presentation.MitgliedIdentifizierenController"
 name="MitgliedIdentifizierenForm"
 input="/MitgliedIdentifizieren.jsp"
 scope="request"

parameter="MitgliedsnummerAngeben_Init,a64aa88aa2aa37ad0cdd0bf506f30162ba7ffa">
 <forward name="Ok"
  path="/MitgliedIdentifizieren.jsp" contextRelative="true"
 />
</action>
```

Fazit

Mit der skizzierten praktischen Interpretation ist MDA/D heute produktiv bei der Anwendungsentwicklung einsetzbar. In der Rolle des Entwicklers gewinnt man mehr Zeit für die „wesentlichen Aufgaben", wie der Umsetzung der projektspezifischen Fachlogik.

Die lästige Copy & Paste-Arbeit zur Erstellung von technischem Infrastrukturcode, der für die fachliche Programmierung von keinerlei Bedeutung ist, wird von einem MDA/D-Generator übernommen. Z.B. ist die manuelle Programmierung des gesamten technischen Codes einer EJB Entity Bean (Interfaces, Deployment Deskriptoren, Primary Keys,...) bei den ersten Beans ja noch interessant, um die Zusammenhänge zu begreifen, macht aber spätestens bei der dritten Bean ohne generative Unterstützung keinen Spaß mehr. Zusätzlich sorgt die Generierung des technischen Codes dafür, dass dieser immer exakt die gleiche Qualität besitzt und von gleicher Systematik ist, was die Wartbarkeit der Software und den Einarbeitungsaufwand in Programmteile anderer Teammitglieder erleichtert. Letztendlich ist eine Fehlerbehebung im technischen Code, im Vergleich zu nicht generativen Ansätzen, sehr viel schneller und effizienter durchzuführen. Ein Bug im Infrastrukturcode muss ausschließlich an einer Stelle, nämlich in der Transformationsvorschrift gefixt werden (ähnlich dem Fixen von Fehlern in einem Framework). Nach erneuter Generierung werden sämtliche fehlerhaften Codefragmente durch korrigierte ersetzt , sodass eine Behebung des Fehlers im gesamten Projekt sichergestellt ist.

Für die Zukunft aus Entwicklungssicht wäre eine bessere Unterstützung zum verteilten Arbeiten an UML Modellen durch die UML-Tool Hersteller (Modellversionierung, Modularisierung,...) wünschenswert. Vielversprechend erscheinen die zu erwartenden Verbesserungen in Richtung der Generierung automatischer Tests, sowie der Verfügbarkeit von Basisbausteinen zur Verwendung bekannter Architekturmuster in der generativen Architektur.

Links & Literatur

[1] MDA Specifications, 2003, www.omg.org/mda/

[2] Generative Development Process,2003, www.architectureware.de/gdp/

[3] T. Stahl, M. Schepe, C. Robitzki: „Model Driven Architect", *Java Magazin* 9.2003

[4] J2EE Specification 1.4 Proposed Final Draft, 2003, java.sun.com/j2ee/download.html#platformspec

[5] Jakarta Struts Framework, 2003, jakarta.apache.org/struts/

[6] Poseidon UML Community Edition, 2003, www.gentleware.de/products/poseidonCE.php3

[7] b+m GeneratorFramework, 2003, www.architectureware.de/genfw/

[8] MDA/D Tools bei der OMG, 2003, www.omg.org/mda/committed-products.htm

[9] J2EE Core Patterns, 2003, java.sun.com/blueprints/patterns/index.html

[10] M. Völter: „Codegenerierung – Ein Überblick", *Java Magazin* (noch nicht veröffentlicht)

[11] T. Weilkiens, B. Oestereich, T.Stahl: „Vom Modell zum Code – ein kurzer Weg mit MDA", *Java Magazin* 9.2003